Database Triggers

You may fire when you are ready, Gridley.

George Dewey: *at the battle of Manila Bay*

This chapter discusses database triggers; that is, procedures that are stored in the database and implicitly executed ("fired") when a table is modified. This chapter includes:

- An Introduction to Triggers
- Parts of a Trigger
- Triggers Execution

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide*.

An Introduction to Triggers

Oracle allows you to define procedures that are implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table. These procedures are called database triggers.

Triggers are similar to stored procedures, discussed in Chapter 14, "Procedures and Packages". A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. While a procedure is explicitly executed by a user, application, or trigger, one or more triggers are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter which user is connected or which application is being used.

For example, Figure 15 - 1 shows a database application with some SQL statements that implicitly fire several triggers stored in the database.

Figure 15 - 1. Triggers

Notice that triggers are stored in the database separately from their associated tables.

Triggers can be defined only on tables, not on views. However, triggers on the base table(s) of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against a view.

How Triggers Are Used

In many cases, triggers supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can permit DML operations against a table only if they are issued during regular business hours. The standard security features of Oracle, roles and privileges, govern which users can submit DML statements against the table. In addition, the trigger further restricts DML operations to occur only at certain times during weekdays. This is just one way that you can use triggers to customize information management in an Oracle database.

In addition, triggers are commonly used to

- automatically generate derived column values
- prevent invalid transactions
- enforce complex security authorizations
- enforce referential integrity across nodes in a distributed database

- enforce complex business rules
- provide transparent event logging
- provide sophisticated auditing
- maintain synchronous table replicates
- gather statistics on table access

Examples of many of these different trigger uses are included in the *Oracle7 Server Application Developer's Guide*.

A Cautionary Note about Trigger Use

When a trigger is fired, a SQL statement within its trigger action potentially can fire other triggers, as illustrated in Figure 15 - 2. When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*.

Figure 15 - 2. Cascading Triggers

While triggers are useful for customizing a database, you should only use triggers when necessary. The excessive use of triggers can result in complex interdependences, which may be difficult to maintain in a large application.

Database Triggers vs. Oracle Forms Triggers

Oracle Forms can also define, store, and execute triggers. However, do not confuse Oracle Forms triggers with the database triggers discussed in this chapter.

Database triggers are defined on a table, stored in the associated database, and executed as a result of an INSERT, UPDATE, or DELETE statement being issued against a table, no matter which user or application issues the statement.

Oracle Forms triggers are part of an Oracle Forms application and are fired only when a specific trigger point is executed within a specific Oracle Forms application. SQL statements within an Oracle Forms application, as with any database application, can implicitly cause the firing of any associated database trigger. For more information about Oracle Forms and Oracle Forms triggers, see the *Oracle Forms User's Guide*.

Triggers vs. Declarative Integrity Constraints

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

A declarative integrity constraint is a statement about the database that is never false while the constraint is enabled. A constraint applies to existing data in the table and any statement that manipulates the table.

Triggers constrain what transactions can do. A trigger does not apply to data loaded before the definition of the trigger. Therefore, it does not guarantee all data in a table conforms to its rules.

A trigger enforces transitional constraints; that is, a trigger only enforces a constraint at the time that the data changes. Therefore, a constraint such as "make sure that the delivery date is at least seven days from today" should be enforced by a trigger, not a declarative integrity constraint.

In evaluating triggers that contain SQL functions that have NLS parameters as arguments (for example, TO_CHAR, TO_DATE, and TO_NUMBER), the default values for these parameters are taken from the NLS parameters currently in effect for the session. You can override the default values by specifying NLS parameters explicitly in such functions when you create a trigger.

For more information about declarative integrity constraints, see Chapter 7, "Data Integrity".

Parts of a Trigger

A trigger has three basic parts:

- a triggering event or statement
- a trigger restriction
- a trigger action

Figure 15 - 3 represents each of these parts of a trigger and is not meant to show exact syntax. Each part of a trigger is explained in greater detail in the following sections.

Figure 15 - 3. The REORDER Trigger

Triggering Event or Statement

A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement on a table.

For example, in Figure 15 - 3, the triggering statement is

```
. . . UPDATE OF parts_on_hand ON inventory . . .
```

which means that when the PARTS_ON_HAND column of a row in the INVENTORY table is updated, fire the trigger. Note that when the triggering event is an UPDATE statement, you can include a column list to identify which columns must be updated to fire the trigger. Because INSERT and DELETE statements affect entire rows of information, a column list cannot be specified for these options.

A triggering event can specify multiple DML statements, as in

```
. . . INSERT OR UPDATE OR DELETE OF inventory . . .
```

which means that when an INSERT, UPDATE, or DELETE statement is issued against the INVENTORY table, fire the trigger. When multiple types of DML statements can fire a trigger, conditional predicates can be used to detect the type of triggering statement. Therefore, a single trigger can be created that executes different code based on the type of statement that fired the trigger.

Trigger Restriction

A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire. The trigger action is not executed if the trigger restriction evaluates to FALSE or UNKNOWN.

A trigger restriction is an option available for triggers that are fired for each row. Its function is to control the execution of a trigger conditionally. You specify a trigger restriction using a WHEN clause. For example, the REORDER trigger in Figure 15 - 3 has a trigger restriction. The trigger is fired by an UPDATE statement affecting the PARTS_ON_HAND column of the INVENTORY table, but the trigger action only fires if the following expression is TRUE:

```
new.parts_on_hand < new.reorder_point
```

Trigger Action

A trigger action is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE.

Similar to stored procedures, a trigger action can contain SQL and PL/SQL statements, define PL/SQL language constructs (variables, constants, cursors, exceptions, and so on), and call stored procedures. Additionally, for row trigger, the statements in a trigger action have access to column values (new and old) of the current row being processed by the trigger. Two correlation names provide access to the old and new values for each column.

Types of Triggers

When you define a trigger, you can specify the number of times the trigger action is to be executed: once for every row affected by the triggering statement (such as might be fired by an UPDATE statement that updates many rows), or once for the triggering statement, no matter how many rows it affects.

Row Triggers A row trigger is fired each time the table is affected by the triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all.

Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected. For example, Figure 15 - 3 illustrates a row trigger that uses the values of each row affected by the triggering statement.

Statement Triggers A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected). For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.

Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected. For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

BEFORE vs. AFTER Triggers

When defining a trigger, you can specify the *trigger timing*. That is, you can specify whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

BEFORE Triggers BEFORE triggers execute the trigger action before the triggering statement. This type of trigger is commonly used in the following situations:

- BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By using a BEFORE trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.

- BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER Triggers AFTER triggers execute the trigger action after the triggering statement is executed. AFTER triggers are used in the following situations:

- AFTER triggers are used when you want the triggering statement to complete before executing the trigger action.

- If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

Combinations

Using the options listed in the previous two sections, you can create four types of triggers:

- **BEFORE statement trigger** Before executing the triggering statement, the trigger action is executed.
- **BEFORE row trigger** Before modifying each row affected by the triggering statement and before checking appropriate integrity constraints, the trigger action is executed provided that the trigger restriction was not violated.
- **AFTER statement trigger** After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
- **AFTER row trigger** After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row provided the trigger restriction was not violated. Unlike BEFORE row triggers, AFTER row triggers lock rows.

You can have multiple triggers of the same type for the same statement for any given table. For example you may have two BEFORE STATEMENT triggers for UPDATE statements on the EMP table. Multiple triggers of the same type permit modular installation of applications that have triggers on the same tables. Also, Oracle snapshot logs use AFTER ROW triggers, so you can design your own AFTER ROW trigger in addition to the Oracle-defined AFTER ROW trigger.

You can create as many triggers of the preceding different types as you need for each type of DML statement (INSERT, UPDATE, or DELETE). For example, suppose you have a table, SAL, and you want to know when the table is being accessed and the types of queries being issued. Figure 15 - 4 contains a sample package and trigger that tracks this information by hour and type of action (for example, UPDATE, DELETE, or INSERT) on table SAL. A global session variable, STAT.ROWCNT, is initialized to zero by a BEFORE statement trigger, then it is increased each time the row trigger is executed, and finally the statistical information is saved in the table STAT_TAB by the AFTER statement trigger.

```
DROP TABLE stat_tab;
```

```
CREATE TABLE stat_tab(utype CHAR(8),
                       rowcnt INTEGER, uhour INTEGER);

CREATE OR REPLACE PACKAGE stat IS
 rowcnt INTEGER;
END;
/

CREATE TRIGGER bt BEFORE UPDATE OR DELETE OR INSERT ON sal
BEGIN
 stat.rowcnt := 0;
END;
/

CREATE TRIGGER rt BEFORE UPDATE OR DELETE OR INSERT ON sal
FOR EACH ROW BEGIN
 stat.rowcnt := stat.rowcnt + 1;
END;
/

CREATE TRIGGER at AFTER UPDATE OR DELETE OR INSERT ON sal
DECLARE
 typ  CHAR(8);
 hour NUMBER;
BEGIN
 IF updating
 THEN typ := 'update'; END IF;
 IF deleting  THEN typ := 'delete'; END IF;
 IF inserting THEN typ := 'insert'; END IF;

 hour := TRUNC((SYSDATE - TRUNC(SYSDATE)) * 24);
 UPDATE stat_tab
    SET rowcnt = rowcnt + stat.rowcnt
   WHERE utype = typ
      AND uhour = hour;
 IF SQL%ROWCOUNT = 0 THEN
   INSERT INTO stat_tab VALUES (typ, stat.rowcnt, hour);
 END IF;

EXCEPTION
 WHEN dup_val_on_index THEN
   UPDATE stat_tab
      SET rowcnt = rowcnt + stat.rowcnt
     WHERE utype = typ
       AND uhour = hour;
END;
/
```

Figure 15 - 4. Sample Package and Trigger for SAL Table

Trigger Execution

A trigger can be in either of two distinct modes:

enabled

An *enabled* trigger executes its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE.

disabled

A *disabled* trigger does not execute its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to TRUE.

For enabled triggers, Oracle automatically

- executes triggers of each type in a planned firing sequence when more than one trigger is fired by a single SQL statement
- performs integrity constraint checking at a set point in time with respect to the different types of triggers and guarantees that triggers cannot compromise integrity constraints
- provides read-consistent views for queries and constraints
- manages the dependencies among triggers and objects referenced in the code of the trigger action
- uses two-phase commit if a trigger updates remote tables in a distributed database
- if more than one trigger of the same type for a given statement exists, Oracle fires each of those triggers in an unspecified order

The Execution Model for Triggers and Integrity Constraint Checking

A single SQL statement can potentially fire up to four types of triggers: BEFORE row triggers, BEFORE statement triggers, AFTER row triggers, and AFTER statement triggers. A triggering statement or a statement within a trigger can cause one or more integrity constraints to be checked. Also, triggers can contain statements that cause other triggers to fire (cascading triggers).

Oracle uses the following execution model to maintain the proper firing sequence of multiple triggers and constraint checking:

```
1. Execute all BEFORE statement triggers that apply to the statement.
2. Loop for each row affected by the SQL statement.
```

```
a. Execute all BEFORE row triggers that apply to the
statement.
b. Lock and change row, and perform integrity constraint
checking (The lock is not released until the
transaction is committed.)
c. Execute all AFTER row triggers that apply to the
statement.
3. Complete deferred integrity constraint checking.
4. Execute all AFTER statement triggers that apply to the statement.
```

The definition of the execution model is recursive. For example, a given SQL statement can cause a BEFORE row trigger to be fired and an integrity constraint to be checked. That BEFORE row trigger, in turn, might perform an update that causes an integrity constraint to be checked and an AFTER statement trigger to be fired. The AFTER statement trigger causes an integrity constraint to be checked. In this case, the execution model executes the steps recursively, as follows:

```
1. Original SQL statement issued.
2. BEFORE row triggers fired.
3. AFTER statement triggers fired by UPDATE in
BEFORE row trigger.
4. Statements of AFTER statement triggers
executed.
5. Integrity constraint on tables changed by
AFTER statement triggers checked.
6. Statements of BEFORE row triggers executed.
7. Integrity constraint on tables changed by
BEFORE row triggers checked.
8. SQL statement executed.
9. Integrity constraint from SQL statement checked.
```

An important property of the execution model is that all actions and checks done as a result of a SQL statement must succeed. If an exception is raised within a trigger, and the exception is not explicitly handled, all actions performed as a result of the original SQL statement, including the actions performed by fired triggers, are rolled back. Thus, integrity constraints cannot be compromised by triggers. The execution model takes into account integrity constraints and disallows triggers that violate declarative integrity constraints.

For example, in the previously outlined scenario, suppose that Steps 1 through 8 succeed; however, in Step 9 the integrity constraint is violated. As a result of this violation, all changes made by the SQL statement (in Step 8), the fired BEFORE row trigger (in Step 6), and the fired AFTER statement trigger (in Step 4) are rolled back.

Note: Be aware that triggers of different types are fired in a specific order. However, triggers of the same type for the same statement are not guaranteed to fire in any specific order. For example, all BEFORE ROW triggers for a single UPDATE statement may not always fire in the same order. Design your applications not to rely on the firing order of multiple triggers of the same type.

Data Access for Triggers

When a trigger is fired, the tables referenced in the trigger action might be currently undergoing changes by SQL statements contained in other users' transactions. In all cases, the SQL statements executed within triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either needs to read (query) or write (update), the SQL statements in the body of the trigger being fired use the following guidelines:

- Queries see the current read-consistent snapshot of referenced tables and any data changed within the same transaction.

- Updates wait for existing data locks before proceeding.

The following examples illustrate these points.

Example

Assume that the SALARY_CHECK trigger (body) includes the following SELECT statement:

```
SELECT minsal, maxsal INTO minsal, maxsal
        FROM salgrade
        WHERE job_classification = :new.job_classification;
```

For this example, assume that transaction T1 includes an update to the MAXSAL column of the SALGRADE table. At this point, the SALARY_CHECK trigger is fired by a statement in transaction T2. The SELECT statement within the fired trigger (originating from T2) does not see the update by the uncommitted transaction T1, and the query in the trigger returns the old MAXSAL value as of the read-consistent point for transaction T2.

Example

Assume the following definition of the TOTAL_SALARY trigger, a trigger to maintain a derived column that stores the total salary of all members in a department:

```
CREATE TRIGGER total_salary
AFTER DELETE OR INSERT OR UPDATE OF deptno, sal ON emp
  FOR EACH ROW BEGIN
  /* assume that DEPTNO and SAL are non-null fields */
   IF DELETING OR (UPDATING AND :old.deptno != :new.deptno)
   THEN UPDATE dept
        SET total_sal = total_sal - :old.sal
        WHERE deptno = :old.deptno;
   END IF;
   IF INSERTING OR (UPDATING AND :old.deptno != :new.deptno)
   THEN UPDATE dept
    SET total_sal = total_sal + :new.sal
    WHERE deptno = :new.deptno;
   END IF;
   IF (UPDATING AND :old.deptno = :new.deptno AND
          :old.sal != :new.sal )
   THEN UPDATE dept
    SET total_sal = total_sal - :old.sal + :new.sal
   WHERE deptno = :new.deptno;
  END IF;
 END;
```

For this example, suppose that one user's uncommitted transaction includes an update to the TOTAL_SAL column of a row in the DEPT table. At this point, the TOTAL_SALARY trigger is fired by a second user's SQL statement. Because the **uncommitted** transaction of the first user contains an update to a pertinent value in the TOTAL_SAL column (in other words, a row lock is being held), the updates performed by the TOTAL_SALARY trigger are not executed until the transaction holding the row lock is committed or rolled back. Therefore, the second user waits until the commit or rollback point of the first user's transaction.

Storage for Triggers

For release 7.3, Oracle stores triggers in their compiled form, just like stored procedures. When a CREATE TRIGGER statement commits, the compiled PL/SQL code, called P code (for pseudocode), is stored in the database and the source code of a trigger is flushed from the shared pool.

For More Information

See "How Oracle Stores Procedures and Packages" .

Execution of Triggers

Oracle internally executes a trigger using the same steps used for procedure execution. The subtle and only difference is that a user automatically has the right to fire a trigger if he/she has the privilege to execute the triggering statement. Other than this, triggers are validated and executed the same way as stored procedures.

For More Information

See "How Oracle Executes Procedures and Packages" .

Dependency Maintenance for Triggers

Oracle automatically manages the dependencies of a trigger on the schema objects referenced in its trigger action. The dependency issues for triggers are the same as dependency issues for stored procedures. In releases earlier than 7.3, triggers were kept in memory. In release 7.3, triggers are treated like stored procedures; they are inserted in the data dictionary. Like procedures, triggers are dependent on referenced objects. Oracle automatically manages dependencies among objects.

For More Information

See Chapter 16, "Dependencies Among Schema Objects".