17 Triggers

This chapter discusses triggers, which are procedures written in PL/SQL, Java, or C that run (fire) implicitly whenever a table or view is modified or when some user actions or database system actions occur. You can write triggers that fire whenever one of the following operations occurs: DML statements on a particular schema object, DDL statements issued within a schema or database, user logon or logoff events, server errors, database startup, or instance shutdown.

This chapter includes:

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

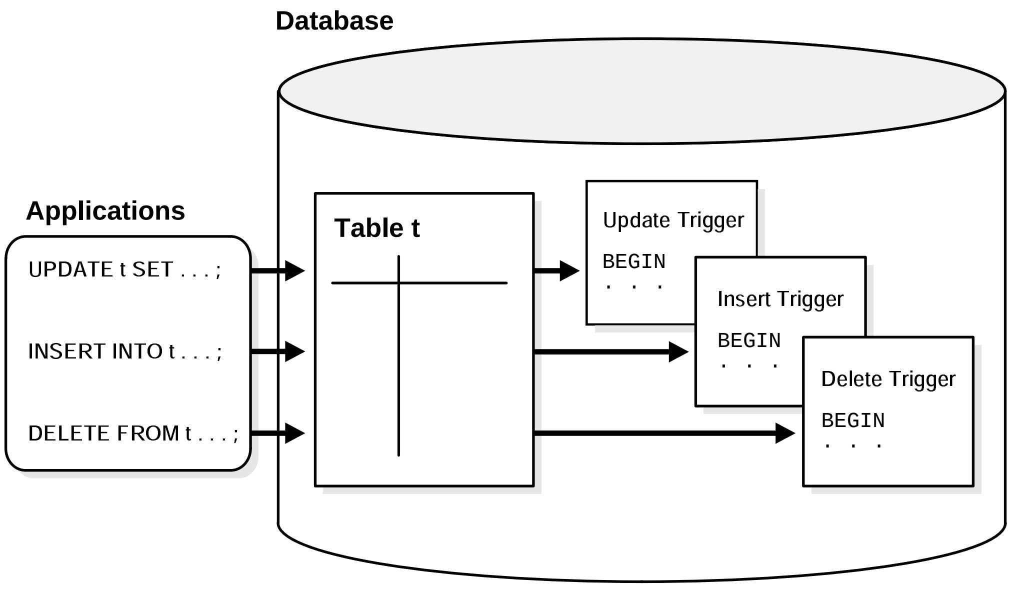
Introduction to Triggers

Oracle lets you define procedures called **triggers** that run implicitly when an `INSERT`, `UPDATE`, or `DELETE` statement is issued against the associated table or, in some cases, against a view, or when database system actions occur. These procedures can be written in PL/SQL or Java and stored in the database, or they can be written as C callouts.

Triggers are similar to stored procedures. A trigger stored in the database can include SQL and PL/SQL or Java statements to run as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by Oracle when a triggering event occurs, no matter which user is connected or which application is being used.

Figure 17- 1 shows a database application with some SQL statements that implicitly fire several triggers stored in the database. Notice that the database stores triggers separately from their associated tables.

Figure 17-1 Triggers



A trigger can also call out to a C procedure, which is useful for computationally intensive operations.

The events that fire a trigger include the following:

- DML statements that modify data in a table (`INSERT`, `UPDATE`, **or** `DELETE`)
- DDL statements
- System events such as startup, shutdown, and error messages
- User events such as logon and logoff

Note: Oracle Forms can define, store, and run triggers of a different sort. However, do not confuse Oracle Forms triggers with the triggers discussed in this chapter.

See Also:

- Chapter 14, "SQL, PL/SQL, and Java" for information on the similarities of triggers to stored procedures
- "The Triggering Event or Statement" on page 17-7

How Triggers Are Used

Triggers supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can restrict DML operations against a table to those issued during regular business hours. You can also use triggers to:

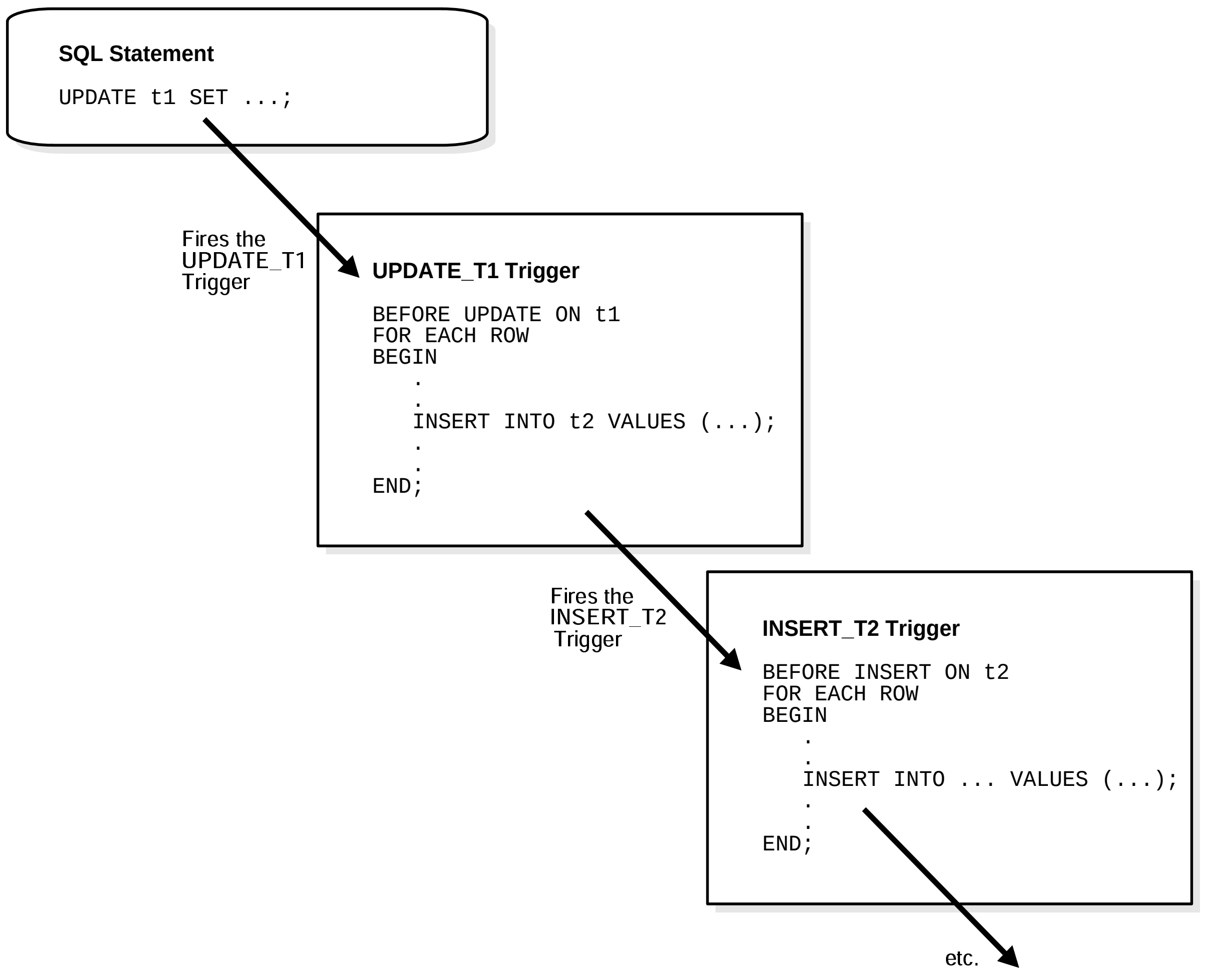
- Automatically generate derived column values
- Prevent invalid transactions
- Enforce complex security authorizations
- Enforce referential integrity across nodes in a distributed database
- Enforce complex business rules
- Provide transparent event logging
- Provide auditing
- Maintain synchronous table replicates
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Publish information about database events, user events, and SQL statements to subscribing applications

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for examples of trigger uses

Some Cautionary Notes about Triggers

Although triggers are useful for customizing a database, use them only when necessary. Excessive use of triggers can result in complex interdependencies, which can be difficult to maintain in a large application. For example, when a trigger fires, a SQL statement within its trigger action potentially can fire other triggers, resulting in **cascading triggers**. This can produce unintended effects. Figure 17- 2 illustrates cascading triggers.

Figure 17- 2 Cascading Triggers



```
UPDATE t1 SET ...;
```

```
BEFORE UPDATE ON t1
FOR EACH ROW
BEGIN
   .
   .
   INSERT INTO t2 VALUES (...);
   .
   .
END;
```

```
BEFORE INSERT ON t2
FOR EACH ROW
BEGIN
   .
   .
   INSERT INTO ... VALUES (...);
   .
   .
END;
```

Triggers Compared with Declarative Integrity Constraints

You can use both triggers and integrity constraints to define and enforce any type of integrity rule. However, Oracle Corporation strongly recommends that you use triggers to constrain data input only in the following situations:

- To enforce referential integrity when child and parent tables are on different nodes of a distributed database

- To enforce complex business rules not definable using integrity constraints
- When a required referential integrity rule cannot be enforced using the following integrity constraints:
  - `NOT NULL`, `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`
  - `DELETE CASCADE`
  - `DELETE SET NULL`

  See Also: "How Oracle Enforces Data Integrity" on page 21-4 for more information about integrity constraints

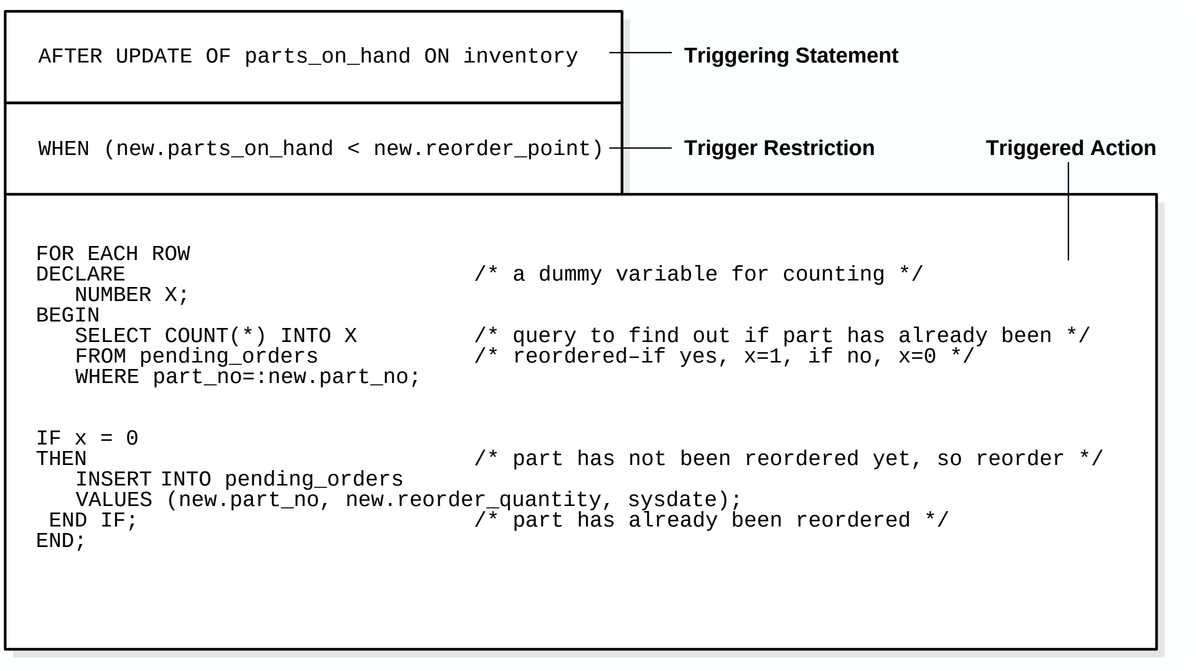
Parts of a Trigger

A trigger has three basic parts:

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

Figure 17- 3 represents each of these parts of a trigger and is not meant to show exact syntax. The sections that follow explain each part of a trigger in greater detail.

Figure 17- 3 The REORDER Trigger



```
AFTER UPDATE OF parts_on_hand ON inventory

WHEN (new.parts_on_hand < new.reorder_point)

FOR EACH ROW
DECLARE                           /* a dummy variable for counting */
   NUMBER X;
BEGIN
   SELECT COUNT(*) INTO X         /* query to find out if part has already been */
   FROM pending_orders            /* reordered-if yes, x=1, if no, x=0 */
   WHERE part_no=:new.part_no;


IF x = 0
THEN                              /* part has not been reordered yet, so reorder */
   INSERT INTO pending_orders
   VALUES (new.part_no, new.reorder_quantity, sysdate);
 END IF;                          /* part has already been reordered */
END;
```

The Triggering Event or Statement

A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:

- An `INSERT`, `UPDATE`, or `DELETE` statement on a specific table (or view, in some cases)
- A `CREATE`, `ALTER`, or `DROP` statement on any schema object
- A database startup or instance shutdown
- A specific error message or any error message
- A user logon or logoff

For example, in Figure 17- 3, the triggering statement is:

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

This statement means that when the `parts_on_hand` column of a row in the `inventory` table is updated, fire the trigger. 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. You cannot specify a column list for INSERT and DELETE statements, because they affect entire rows of information.

A triggering event can specify multiple SQL statements:

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

This part means that when an INSERT, UPDATE, or DELETE statement is issued against the inventory table, fire the trigger. When multiple types of SQL statements can fire a trigger, you can use conditional predicates to detect the type of triggering statement. In this way, you can create a single trigger that runs different code based on the type of statement that fires the trigger.

Trigger Restriction

A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. The trigger action is not run if the trigger restriction evaluates to false or unknown. In the example, the trigger restriction is:

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

Consequently, the trigger does not fire unless the number of available parts is less than a present reorder amount.

Trigger Action

A trigger action is the procedure (PL/SQL block, Java program, or C callout) that contains the SQL statements and code to be run when the following events occur:

- A triggering statement is issued.
- The trigger restriction evaluates to true.

Like stored procedures, a trigger action can:

- Contain SQL, PL/SQL, or Java statements
- Define PL/SQL language constructs such as variables, constants, cursors, exceptions
- Define Java language constructs
- Call stored procedures

If the triggers are row triggers, the statements in a trigger action have access to column values of the row being processed by the trigger. Correlation names provide access to the old and new values for each column.

Types of Triggers

This section describes the different types of triggers:

- Row Triggers and Statement Triggers
- BEFORE and AFTER Triggers
- INSTEAD OF Triggers
- Triggers on System Events and User Events

Row Triggers and Statement Triggers

When you define a trigger, you can specify the number of times the trigger action is to be run:

- Once for every row affected by the triggering statement, such as a trigger fired by an UPDATE statement that updates many rows
- 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 run.

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 17- 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.

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, use a statement trigger to:

- Make a complex security check on the current time or user
- Generate a single audit record

BEFORE and AFTER Triggers

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

`BEFORE` and `AFTER` triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the base tables of a view are fired if an `INSERT`, `UPDATE`, or `DELETE` statement is issued against the view. `BEFORE` and `AFTER` triggers fired by DDL statements can be defined only on the database or a schema, not on particular tables.

See Also:

- "INSTEAD OF Triggers" on page 17-12
- "Triggers on System Events and User Events" on page 17-14 for information about how `BEFORE` and `AFTER` triggers can be used to publish information about DML and DDL statements

BEFORE Triggers

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

- When the trigger action determines whether the triggering statement should be allowed to complete. 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.
- To derive specific column values before completing a triggering `INSERT` or `UPDATE` statement.

AFTER Triggers

`AFTER` triggers run the trigger action after the triggering statement is run.

Trigger Type Combinations

Using the options listed previously, you can create four types of row and statement triggers:

- **BEFORE** *statement* **trigger**

  Before executing the triggering statement, the trigger action is run.

- **BEFORE *row* trigger**

  Before modifying each row affected by the triggering statement and before checking appropriate integrity constraints, the trigger action is run, if the trigger restriction was not violated.

- **AFTER *row* trigger**

  After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is run for the current row provided the trigger restriction was not violated. Unlike `BEFORE` *`row`* triggers, `AFTER` *`row`* triggers lock rows.

- **AFTER *statement* trigger**

  After executing the triggering statement and applying any deferred integrity constraints, the trigger action is run.

You can have multiple triggers of the same type for the same statement for any given table. For example, you can have two `BEFORE` *`statement`* triggers for `UPDATE` statements on the `employees` table. Multiple triggers of the same type permit modular installation of applications that have triggers on the same tables. Also, Oracle materialized view 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. The following example 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`. The global session variable `STAT.ROWCNT` is initialized to zero by a `BEFORE` *`statement`* trigger. Then it is increased each time the row trigger is run. Finally the statistical information is saved in the table `STAT_TAB` by the `AFTER` *`statement`* trigger.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for examples of trigger applications

INSTEAD OF Triggers

`INSTEAD OF` triggers provide a transparent way of modifying views that cannot be modified directly through DML statements (`INSERT`, `UPDATE`, and `DELETE`). These triggers are called `INSTEAD OF` triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.

You can write normal `INSERT`, `UPDATE`, and `DELETE` statements against the view and the `INSTEAD OF` trigger is fired to update the underlying tables appropriately. `INSTEAD OF` triggers are activated for each row of the view that gets modified.

Modify Views

Modifying views can have ambiguous results:

- Deleting a row in a view could either mean deleting it from the base table or updating some values so that it is no longer selected by the view.
- Inserting a row in a view could either mean inserting a new row into the base table or updating an existing row so that it is projected by the view.
- Updating a column in a view that involves joins might change the semantics of other columns that are not projected by the view.

Object views present additional problems. For example, a key use of object views is to represent master/detail relationships. This operation inevitably involves joins, but modifying joins is inherently ambiguous.

As a result of these ambiguities, there are many restrictions on which views are modifiable. An `INSTEAD OF` trigger can be used on object views as well as relational views that are not otherwise modifiable.

Even if the view is inherently modifiable, you might want to perform validations on the values being inserted, updated or deleted. `INSTEAD OF` triggers can also be used in this case. Here the trigger code performs the validation on the rows being modified and if valid, propagate the changes to the underlying tables.

`INSTEAD OF` triggers also enable you to modify object view instances on the client-side through OCI. 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 inherently modifiable. However, it is not necessary to define these triggers for just pinning and reading the view object in the object cache.

See Also:

- Chapter 13, "Object Datatypes and Object Views"
- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Application Developer's Guide - Fundamentals* for an example of an `INSTEAD OF` trigger

Views That Are Not Modifiable

A view is **inherently modifiable** if data can be inserted, updated, or deleted without using `INSTEAD OF` triggers and if it conforms to the restrictions listed as follows. If the view query contains any of the following constructs, the view is not inherently modifiable and you therefore cannot perform inserts, updates, or deletes on the view:

- Set operators
- Aggregate functions
- `GROUP BY`, `CONNECT BY`, or `START WITH` clauses
- The `DISTINCT` operator
- Joins (however, some join views are updatable)

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

See Also: "Updatable Join Views" on page 10-20

INSTEAD OF Triggers on Nested Tables

You cannot modify the elements of a nested table column in a view directly with the `TABLE` clause. However, you can do so by defining an `INSTEAD OF` trigger on the nested table column of the view. The triggers on the nested tables fire if a nested table element is updated, inserted, or deleted and handle the actual modifications to the underlying tables.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i SQL Reference* for information on the `CREATE TRIGGER` statement

Triggers on System Events and User Events

You can use triggers to publish information about database events to subscribers. Applications can subscribe to database events just as they subscribe to messages from other applications. These database events can include:

- System events
  - Database startup and shutdown
  - Server error message events
- User events
  - User logon and logoff
  - DDL statements (`CREATE`, `ALTER`, and `DROP`)
  - DML statements (`INSERT`, `DELETE`, and `UPDATE`)

Triggers on system events can be defined at the database level or schema level. For example, a database shutdown trigger is defined at the database level:

```
CREATE TRIGGER register_shutdown
  ON DATABASE
  SHUTDOWN
    BEGIN
    ...
    DBMS_AQ.ENQUEUE(...);
    ...
    END;
```

Triggers on DDL statements or logon/logoff events can also be defined at the database level or schema level. Triggers on DML statements can be defined on a table or view. A trigger defined at the database level fires for all users, and a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Event Publication

Event publication uses the publish-subscribe mechanism of Oracle Advanced Queuing. A **queue** serves as a message repository for subjects of interest to various subscribers. Triggers use the `DBMS_AQ` package to enqueue a message when specific system or user events occur.

See Also:

- *Oracle9i Application Developer's Guide - Advanced Queuing*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Event Attributes

Each event allows the use of attributes within the trigger text. For example, the database startup and shutdown triggers have attributes for the instance number and the database name, and the logon and logoff triggers have attributes for the username. You can specify a function with the same name as an attribute when you create a trigger if you want to publish that attribute when the event occurs. The attribute's value is then passed to the function or payload when the trigger fires. For triggers on DML statements, the `:OLD` column values pass the attribute's value to the `:NEW` column value.

System Events

System events that can fire triggers are related to instance startup and shutdown and error messages. Triggers created on startup and shutdown events have to be associated with the database. Triggers created on error events can be associated with the database or with a schema.

- `STARTUP` triggers fire when the database is opened by an instance. Their attributes include the system event, instance number, and database name.
- `SHUTDOWN` triggers fire just before the server starts shutting down an instance. You can use these triggers to make subscribing applications shut down completely when the database shuts down. For abnormal instance shutdown, these triggers cannot be fired. The attributes of `SHUTDOWN` triggers include the system event, instance number, and database name.
- `SERVERERROR` triggers fire when a specified error occurs, or when any error occurs if no error number is specified. Their attributes include the system event and error number.

User Events

User events that can fire triggers are related to user logon and logoff, DDL statements, and DML statements.

Triggers on LOGON and LOGOFF Events `LOGON` and `LOGOFF` triggers can be associated with the database or with a schema. Their attributes include the system event and username, and they can specify simple conditions on `USERID` and `USERNAME`.

- `LOGON` triggers fire after a successful logon of a user.
- `LOGOFF` triggers fire at the start of a user logoff.

Triggers on DDL Statements DDL triggers can be associated with the database or with a schema. Their attributes include the system event, the type of schema object, and its name. They can specify simple conditions on the type and name of the schema object, as well as functions like `USERID` and `USERNAME`. DDL triggers include the following types of triggers:

- `BEFORE CREATE` and `AFTER CREATE` triggers fire when a schema object is created in the database or schema.
- `BEFORE ALTER` and `AFTER ALTER` triggers fire when a schema object is altered in the database or schema.
- `BEFORE DROP` and `AFTER DROP` triggers fire when a schema object is dropped from the database or schema.

Triggers on DML Statements DML triggers for event publication are associated with a table. They can be either `BEFORE` or `AFTER` triggers that fire for each row on which the specified DML operation occurs. You cannot use `INSTEAD OF` triggers on views to publish events related to DML statements- instead, you can publish events using `BEFORE` or `AFTER` triggers for the DML operations on a view's underlying tables that are caused by `INSTEAD OF` triggers.

The attributes of DML triggers for event publication include the system event and the columns defined by the user in the `SELECT` list. They can specify simple conditions on the type and name of the schema object, as well as functions (such as `UID`, `USER`, `USERENV`, and `SYSDATE`), pseudocolumns, and columns. The columns can be prefixed by `:OLD` and `:NEW` for old and new values. Triggers on DML statements include the following triggers:

- `BEFORE INSERT` and `AFTER INSERT` triggers fire for each row inserted into the table.
- `BEFORE UPDATE` and `AFTER UPDATE` triggers fire for each row updated in the table.
- `BEFORE DELETE` and `AFTER DELETE` triggers fire for each row deleted from the table.

See Also:

- "Row Triggers" on page 17-9
- "BEFORE and AFTER Triggers" on page 17-10
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about event publication using triggers on system events and user events

Trigger Execution

A trigger is in either of two distinct modes:

Trigger Mode	Definition
Enabled	An enabled trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to `TRUE`.
Disabled	A disabled trigger does not run 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 performs the following actions:

- Runs 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 schema objects referenced in the code of the trigger action
- Uses two-phase commit if a trigger updates remote tables in a distributed database
- Fires multiple triggers in an unspecified order, if more than one trigger of the same type exists for a given statement

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
- `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. Run all `BEFORE` *`statement`* triggers that apply to the statement.
2. Loop for each row affected by the SQL statement.
   a. Run 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. Run all `AFTER` *`row`* triggers that apply to the statement.
3. Complete deferred integrity constraint checking.
4. Run 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 runs the steps recursively, as follows:

Original SQL statement issued.

1. `BEFORE` *`row`* triggers fired.
   a. `AFTER` *`statement`* triggers fired by `UPDATE` in `BEFORE` *`row`* trigger.

      i. Statements of `AFTER` *`statement`* triggers run.

      ii. Integrity constraint checked on tables changed by `AFTER` *`statement`* triggers.

   b. Statements of `BEFORE` *`row`* triggers run.

   c. Integrity constraint checked on tables changed by `BEFORE` *`row`* triggers.

2. SQL statement run.
3. Integrity constraint from SQL statement checked.

There are two exceptions to this recursion:

- When a triggering statement modifies one table in a referential constraint (either the primary key or foreign key table), and a triggered statement modifies the other, only the triggering statement will check the integrity constraint. This allows row triggers to enhance referential integrity.
- Statement triggers fired due to `DELETE CASCADE` and `DELETE SET NULL` are fired before and after the user `DELETE` statement, not before and after the individual enforcement statements. This prevents those statement triggers from encountering mutating errors.

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: Although triggers of different types are fired in a specific order, 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 so they do not 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 in other users' transactions. In all cases, the SQL statements run 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), then the SQL statements in the body of the trigger being fired use the following guidelines:

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

The following examples illustrate these points.

Data Access for Triggers Example 1 Assume that the `salary_check` trigger (body) includes the following `SELECT` statement:

```
SELECT min_salary, max_salary INTO min_salary, max_salary
  FROM jobs
  WHERE job_title = :new.job_title;
```

For this example, assume that transaction T1 includes an update to the `max_salary` column of the `jobs` 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 `max_salary` value as of the read-consistent point for transaction `T2`.

Data Access for Triggers Example 2 Assume that the `total_salary` trigger maintains 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 department_id, salary ON employees
 FOR EACH ROW BEGIN
  /* assume that department_id and salary are non-null fields */
   IF DELETING OR (UPDATING AND :old.department_id != :new.department_id)
   THEN UPDATE departments
  SET total_salary = total_salary - :old.salary
  WHERE department_id = :old.department_id;
   END IF;
   IF INSERTING OR (UPDATING AND :old.department_id != :new.department_id)
   THEN UPDATE departments
    SET total_salary = total_salary + :new.salary
    WHERE department_id = :new.department_id;
   END IF;
   IF (UPDATING AND :old.department_id = :new.department_id AND
    :old.salary != :new.salary )
   THEN UPDATE departments
    SET total_salary = total_salary - :old.salary + :new.salary
   WHERE department_id = :new.department_id;
  END IF;
 END;
```

For this example, suppose that one user's uncommitted transaction includes an update to the `total_salary` column of a row in the `departments` 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_salary` column (that is, a row lock is being held), the updates performed by the `total_salary` trigger are not run 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 of PL/SQL Triggers

Oracle stores PL/SQL triggers in 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 the trigger is flushed from the shared pool.

See Also: *PL/SQL User's Guide and Reference* for more information about compiling and storing PL/SQL code

Execution of Triggers

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

See Also: *PL/SQL User's Guide and Reference* for more information about stored procedures

Dependency Maintenance for Triggers

Like procedures, triggers depend on referenced objects. 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 those for stored procedures. Triggers are treated like stored procedures. They are inserted into the data dictionary.

See Also: Chapter 15, "Dependencies Among Schema Objects"