10

Creating Triggers

Objectives

After completing this lesson, you should be able to do the following:

- **Describe the different types of triggers**
- **Describe database triggers and their uses**
- **Create database triggers**
- **Describe database trigger-firing rules**
- **Remove database triggers**

Lesson Aim

In this lesson, you learn how to create and use database triggers.

Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or database
- Executes implicitly whenever a particular event takes place
- Can be either of the following:
  - Application trigger: Fires whenever an event occurs with a particular application
  - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

ORACLE

Types of Triggers

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is an application developed with Oracle Forms Developer.

Database triggers execute implicitly when any of the following events occur:

- DML operations on a table
- DML operations on a view, with an `INSTEAD OF` trigger
- DDL statements, such as `CREATE` and `ALTER`

This is the case no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur (for example, when a user logs on or the DBA shuts down the database).

Note: Database triggers can be defined on tables and on views. If a DML operation is issued on a view, then the `INSTEAD OF` trigger defines what actions take place. If these actions include DML operations on tables, then any triggers on the base tables are fired.

Database triggers can be system triggers on a database or a schema. For databases, triggers fire for each event for all users; for a schema, they fire

Guidelines for Designing Triggers

- You can design triggers to:
  - Perform related actions
  - Centralize global operations
- You must not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

Guidelines for Designing Triggers

- Use triggers to guarantee that related actions are performed for a specific operation.
- Use database triggers for centralized, global operations that should be fired for the triggering statement, independent of the user or application issuing the statement.
- Do not define triggers to duplicate or replace the functionality already built into the Oracle database. For example, implement integrity rules using declarative constraints, not triggers. To remember the design order for a business rule:
  - Use built-in constraints in the Oracle server, such as primary key, and so on.
  - Develop a database trigger or an application, such as a servlet or Enterprise JavaBeans (EJB) on your middle tier.
  - Use a presentation interface, such as Oracle Forms, HTML, JavaServer Pages (JSP) and so on, for data presentation rules.
- Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain. Use triggers when necessary, and be aware of recursive and cascading effects.

Creating DML Triggers

Create DML statement or row type triggers by using:

```
CREATE [OR REPLACE] TRIGGER trigger_name
 timing
 event1 [OR event2 OR event3]
ON object_name
[[REFERENCING OLD AS old | NEW AS new]
 FOR EACH ROW
 [WHEN (condition)]]
trigger_body
```

- A statement trigger fires once for a DML statement.
- A row trigger fires once for each row affected.

Note: Trigger names must be unique with respect to other triggers in the same schema.

Creating DML Triggers

The components of the trigger syntax are:

- *trigger_name* uniquely identifies the trigger.
- *timing* indicates when the trigger fires in relation to the triggering event. Values are **BEFORE**, **AFTER**, and **INSTEAD OF**.
- **event** identifies the DML operation causing the trigger to fire. Values are **INSERT**, **UPDATE [OF column]**, and **DELETE**.
- *object_name* indicates the table or view associated with the trigger.
- For row triggers, you can specify:
  - A **REFERENCING** clause to choose correlation names for referencing the old and new values of the current row (default values are **OLD** and **NEW**)
  - **FOR EACH ROW** to designate that the trigger is a row trigger
  - A **WHEN** clause to apply a conditional predicate, in parentheses, which is evaluated for each row to determine whether or not to execute the trigger body

- The *trigger_body* is the action performed by the trigger, implemented as

Types of DML Triggers

The trigger type determines whether the body executes for each row or only once for the triggering statement.

- **A statement trigger:**
  - **Executes once for the triggering event**
  - **Is the default type of trigger**
  - **Fires once even if no rows are affected at all**
- **A row trigger:**
  - **Executes once for each row affected by the triggering event**
  - **Is not executed if the triggering event does not affect any rows**
  - **Is indicated by specifying the `FOR EACH ROW` clause**

ORACLE

Types of DML Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row `UPDATE`) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself (for example, a trigger that performs a complex security check on the current user).

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of rows that are affected or on data provided by the triggering event itself.

Note: Row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

Trigger Timing

When should the trigger fire?

- **BEFORE**: Execute the trigger body before the triggering DML event on a table.
- **AFTER**: Execute the trigger body after the triggering DML event on a table.
- **INSTEAD OF**: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

Note: If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.

Trigger Timing

The **BEFORE** trigger timing is frequently used in the following situations:

- To determine whether the triggering statement should be allowed to complete (This eliminates unnecessary processing and enables a rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing an **INSERT** or **UPDATE** statement
- To initialize global variables or flags, and to validate complex business rules

The **AFTER** triggers are frequently used in the following situations:

- To complete the triggering statement before executing the triggering action
- To perform different actions on the same triggering statement if a **BEFORE** trigger is already present

The **INSTEAD OF** triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because a view is not always modifiable. You can write appropriate DML statements inside the body of an **INSTEAD OF** trigger to perform actions directly on the underlying tables of views.

Note: If multiple triggers are defined for a table, then the order in which

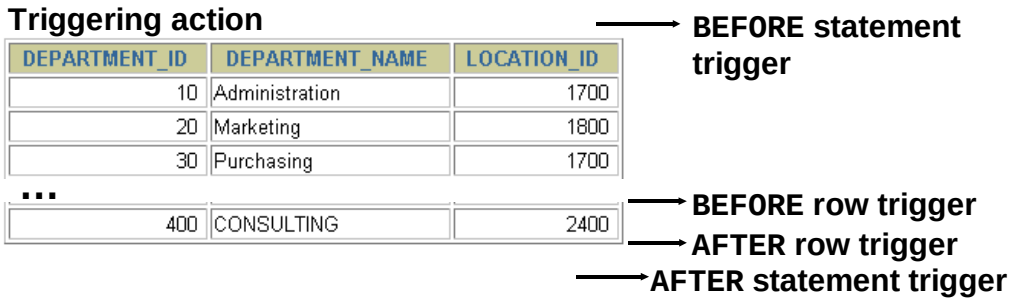
Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when a single row is manipulated:

DML statement

```
INSERT INTO departments
   (department_id,department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```

Triggering action



ORACLE

Trigger-Firing Sequence

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering DML statement affects a single row, both the statement trigger and the row trigger fire exactly once.

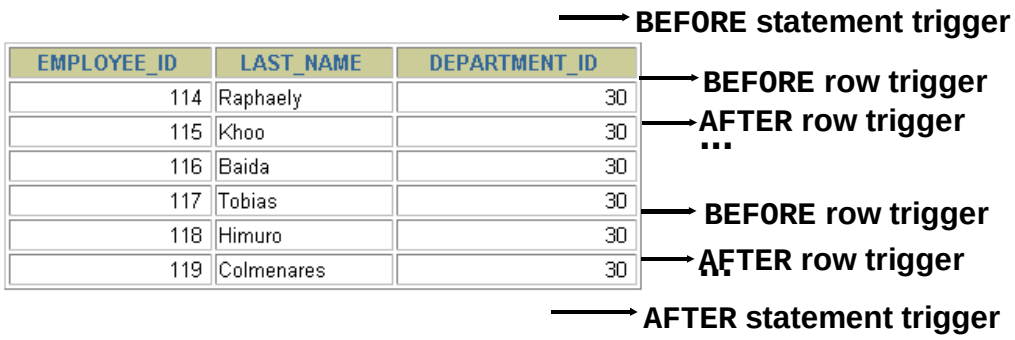
Example

The SQL statement in the slide does not differentiate statement triggers from row triggers because exactly one row is inserted into the table using the syntax for the `INSERT` statement shown in the slide.

Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```



ORACLE

Trigger-Firing Sequence (continued)

When the triggering DML statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Example

The SQL statement in the slide causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the `WHERE` clause (that is, the number of employees reporting to department 30).

Trigger Event Types and Body

A trigger event:

- **Determines which DML statement causes the trigger to execute**
- **Types are:**
  - **`INSERT`**
  - **`UPDATE [OF column]`**
  - **`DELETE`**

A trigger body:

- **Determines what action is performed**
- **Is a PL/SQL block or a `CALL` to a procedure**

ORACLE

Triggering Event Types

The triggering event or statement can be an `INSERT`, `UPDATE`, or `DELETE` statement on a table.

- **When the triggering event is an `UPDATE` statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an `INSERT` or for a `DELETE` statement because it always affects entire rows.**

  . . . UPDATE OF salary . . .

- **The triggering event can contain one, two, or all three of these DML operations.**

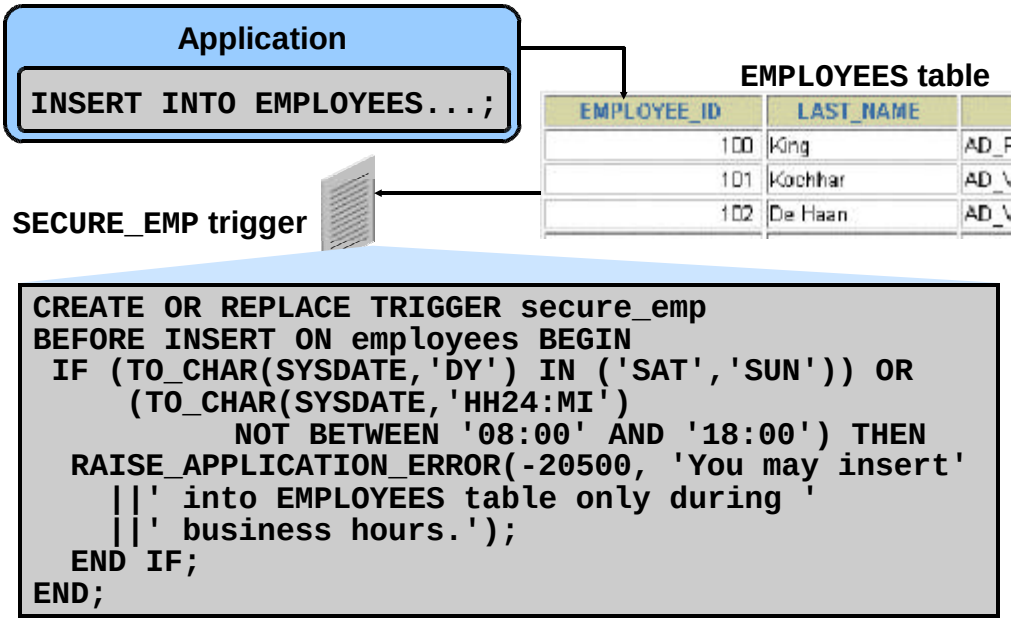
  . . . INSERT or UPDATE or DELETE

  . . . INSERT or UPDATE OF job_id . . .

The trigger body defines the action—that is, what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Note: The size of a trigger cannot be greater than 32 KB.

Creating a DML Statement Trigger



Creating a DML Statement Trigger

In this example, the SECURE_EMP database trigger is a BEFORE statement trigger that prevents the INSERT operation from succeeding if the business condition is violated. In this case, the trigger restricts inserts into the EMPLOYEES table during certain business hours, Monday through Friday.

If a user attempts to insert a row into the EMPLOYEES table on Saturday, then the user sees an error message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server

Testing SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
        first_name, email, hire_date,
        job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
       'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
      *
ERROR at line 1:
ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "PLSQL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'
```


Testing SECURE_EMP

Insert a row into the EMPLOYEES table during nonbusiness hours. When the date and time are out of the business timings specified in the trigger, you receive the error message shown in the slide.

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
   (TO_CHAR(SYSDATE,'HH24')
       NOT BETWEEN '08' AND '18') THEN
   IF DELETING THEN RAISE_APPLICATION_ERROR(
    -20502,'You may delete from EMPLOYEES table'||
           'only during business hours.');
   ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
    -20500,'You may insert into EMPLOYEES table'||
           'only during business hours.');
   ELSIF UPDATING('SALARY') THEN
    RAISE_APPLICATION_ERROR(-20503, 'You may '||
     'update SALARY only during business hours.');
   ELSE RAISE_APPLICATION_ERROR(-20504,'You may'||
     ' update EMPLOYEES table only during'||
     ' normal hours.');
   END IF;
 END IF;
END;
```

ORACLE

Combining Triggering Events

You can combine several triggering events into one by taking advantage of the special conditional predicates `INSERTING`, `UPDATING`, and `DELETING` within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the `EMPLOYEES` table to certain business hours, Monday through Friday.

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
     AND :NEW.salary > 15000 THEN
    RAISE_APPLICATION_ERROR (-20202,
      'Employee cannot earn more than $15,000.');
  END IF;
END;
/
```

ORACLE

Creating a DML Row Trigger

You can create a **BEFORE** row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

In the example, a trigger is created to allow certain employees to be able to earn a salary of more than 15,000. Suppose that a user attempts to execute the following **UPDATE** statement:

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

The trigger raises the following exception:

```
UPDATE EMPLOYEES
  *
ERROR at line 1:
ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "PLSQL.RESTRICT_SALARY", line 5
ORA-04088: error during execution of trigger
"PLSQL.RESTRICT_SALARY"
```

Using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO audit_emp(user_name, time_stamp, id,
    old_last_name, new_last_name, old_title,
    new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.last_name, :NEW.last_name, :OLD.job_id,
    :NEW.job_id, :OLD.salary, :NEW.salary);
END;
/
```


Using OLD and NEW Qualifiers

Within a ROW trigger, reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifiers.

Data Operation	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

Usage notes:

- **The OLD and NEW qualifiers are available only in ROW triggers.**
- **Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.**
- **There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.**

Note: Row triggers can decrease the performance if you perform many updates on larger tables.

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

```
INSERT INTO employees
 (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA_REP', 6000,...);

UPDATE employees
 SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
```

```
SELECT user_name, timestamp, ...
FROM audit_emp;
```

USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
ORA25	31-MAR-06			Temp emp		SA_REP		6000
ORA25	31-MAR-06	999	Temp emp	Smith	SA_REP	SA_REP	6000	7000

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

Create a trigger on the EMPLOYEES table to add rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

There is an additional column named COMMENTS in AUDIT_EMP that is not shown in this slide.

Restricting a Row Trigger: Example

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
 IF INSERTING THEN
   :NEW.commission_pct := 0;
 ELSIF :OLD.commission_pct IS NULL THEN
   :NEW.commission_pct := 0;
 ELSE
   :NEW.commission_pct := :OLD.commission_pct+0.05;
 END IF;
END;
/
```


Restricting a Row Trigger: Example

To restrict the trigger action to those rows that satisfy a certain condition, provide a `WHEN` clause.

Create a trigger on the `EMPLOYEES` table to calculate an employee's commission when a row is added to the `EMPLOYEES` table, or when an employee's salary is modified.

The `NEW` qualifier cannot be prefixed with a colon in the `WHEN` clause because the `WHEN` clause is outside the PL/SQL blocks.

Summary of the Trigger Execution Model

1. Execute all `BEFORE STATEMENT` triggers.
2. Loop for each row affected:
   a. Execute all `BEFORE ROW` triggers.
   b. Execute the DML statement and perform integrity constraint checking.
   c. Execute all `AFTER ROW` triggers.
3. Execute all `AFTER STATEMENT` triggers.

Note: Integrity checking can be deferred until the `COMMIT` operation is performed.

ORACLE

Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers:

- `BEFORE` and `AFTER` statement triggers
- `BEFORE` and `AFTER` row triggers

A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. However, you can defer constraint checking until a `COMMIT` operation is performed.

Triggers can also cause other triggers—known as cascading triggers—to fire.

All actions and checks performed as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, then all actions performed because of the original SQL statement are rolled back (including actions performed by firing triggers). This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for the modified values that the trigger needs to read (query) or write (update).

Implementing an Integrity Constraint with a Trigger

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg
AFTER UPDATE OF department_id
ON employees FOR EACH ROW
BEGIN
 INSERT INTO departments VALUES(:new.department_id,
          'Dept '||:new.department_id, NULL, NULL);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
    NULL; -- mask exception if department exists
END;
/
```

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Successful after trigger is fired
```


Implementing an Integrity Constraint with a Trigger

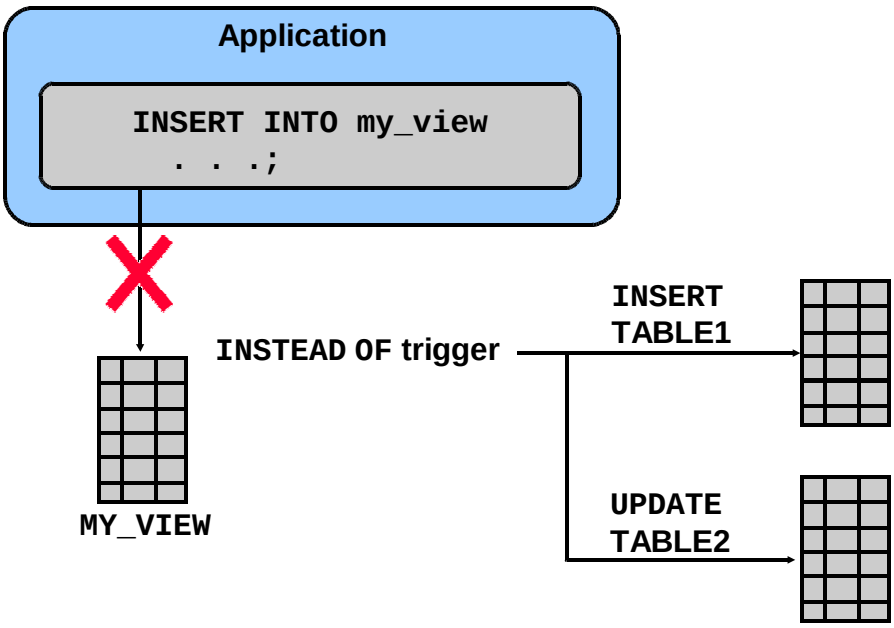
The example in the slide explains a situation in which the integrity constraint can be taken care of by using a trigger. The `EMPLOYEES` table has a foreign key constraint on the `DEPARTMENT_ID` column of the `DEPARTMENTS` table.

In the first SQL statement, the `DEPARTMENT_ID` of the employee 170 is modified to 999. Because department 999 does not exist in the `DEPARTMENTS` table, the statement raises exception –2292 for the integrity constraint violation.

The `EMPLOYEE_DEPT_FK_TRG` trigger is created that inserts a new row into the `DEPARTMENTS` table, using `:NEW.DEPARTMENT_ID` for the value of the new department's `DEPARTMENT_ID`. The trigger fires when the `UPDATE` statement modifies the `DEPARTMENT_ID` of employee 170 to 999. When the foreign key constraint is checked, it is successful because the trigger inserted the department 999 into the `DEPARTMENTS` table. Therefore, no exception occurs unless the department already exists when the trigger attempts to insert the new row. However, the `EXCEPTION` handler traps and masks the exception allowing the operation to succeed.

Note: This example works with Oracle8*i* and later releases but produces a run-time error in releases prior to Oracle8*i*.

INSTEAD OF Triggers



INSTEAD OF Triggers

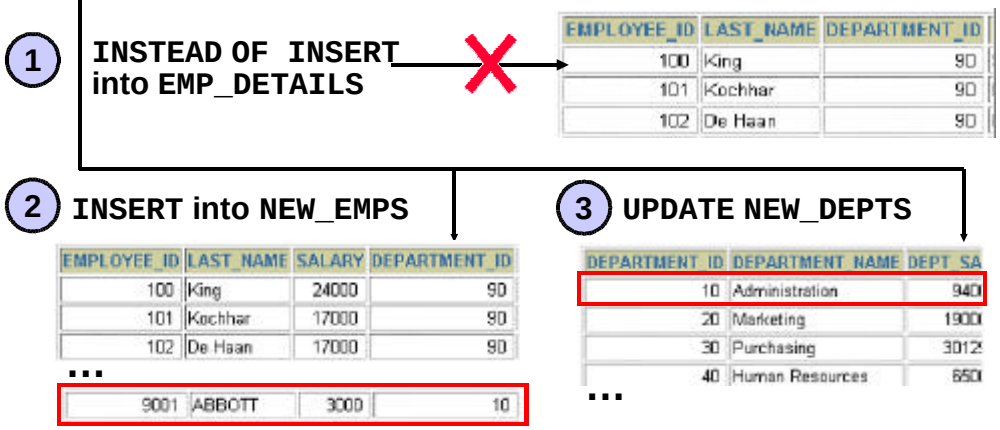
Use INSTEAD OF triggers to modify data in which the DML statement has been issued against an inherently nonupdatable view. These triggers are called INSTEAD OF triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. These triggers are used to perform INSERT, UPDATE, and DELETE operations directly on the underlying tables. You can write INSERT, UPDATE, and DELETE statements against a view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place. A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as GROUP BY, CONNECT BY, START, the DISTINCT operator, or joins. For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So you write an INSTEAD OF trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has INSTEAD OF triggers, then the triggers take precedence. INSTEAD OF triggers are row triggers. The CHECK option for views is not enforced when insertions or updates to the view are performed by using INSTEAD OF triggers. The INSTEAD OF trigger body must

Creating an INSTEAD OF Trigger

Perform the INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables:

```
INSERT INTO emp_details
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```



Creating an INSTEAD OF Trigger

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based. The example illustrates an employee being inserted into view EMP_DETAILS, whose query is based on the EMPLOYEES and DEPARTMENTS tables. The NEW_EMP_DEPT (INSTEAD OF) trigger executes in place of the INSERT operation that causes the trigger to fire. The INSTEAD OF trigger then issues the appropriate INSERT and UPDATE to the base tables used by the EMP_DETAILS view. Therefore, instead of inserting the new employee record into the EMPLOYEES table, the following actions take place:

1. The NEW_EMP_DEPT INSTEAD OF trigger fires.
2. A row is inserted into the NEW_EMPS table.
3. The DEPT_SAL column of the NEW_DEPTS table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

Note: The code for this scenario is shown in the next few pages.

Creating an INSTEAD OF Trigger

Use INSTEAD OF to perform DML on complex views:

```
CREATE TABLE new_emps AS
 SELECT employee_id,last_name,salary,department_id
 FROM employees;

CREATE TABLE new_depts AS
 SELECT d.department_id,d.department_name,
        sum(e.salary) dept_sal
 FROM employees e, departments d
 WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
 SELECT e.employee_id, e.last_name, e.salary,
        e.department_id, d.department_name
 FROM employees e, departments d
 WHERE e.department_id = d.department_id
GROUP BY d.department_id,d.department_name;
```


Creating an INSTEAD OF Trigger (continued)

The example creates two new tables, NEW_EMPS and NEW_DEPTS, based on the EMPLOYEES and DEPARTMENTS tables, respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables.

If a view has a complex query structure, then it is not always possible to perform DML directly on the view to affect the underlying tables. The example requires creation of an INSTEAD OF trigger, called NEW_EMP_DEPT, shown on the next page. The NEW_DEPT_EMP trigger handles DML in the following way:

- When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, using the data values supplied with the INSERT statement.
- When a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

Note: INSTEAD OF triggers can be written only for views, and the BEFORE and AFTER timing options are not valid.

Creating an INSTEAD OF Trigger (continued)

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO new_emps
    VALUES (:NEW.employee_id, :NEW.last_name,
          :NEW.salary, :NEW.department_id);
    UPDATE new_depts
      SET dept_sal = dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
  ELSIF DELETING THEN
    DELETE FROM new_emps
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal - :OLD.salary
      WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('salary') THEN
    UPDATE new_emps
      SET salary = :NEW.salary
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal +
                  (:NEW.salary - :OLD.salary)
      WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('department_id') THEN
    UPDATE new_emps
      SET department_id = :NEW.department_id
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal - :OLD.salary
```

Comparison of Database Triggers and Stored Procedures

Triggers	Procedures
Defined with CREATE TRIGGER	Defined with CREATE PROCEDURE
Data dictionary contains source code in USER_TRIGGERS.	Data dictionary contains source code in USER_SOURCE.
Implicitly invoked by DML	Explicitly invoked
COMMIT, SAVEPOINT, and ROLLBACK are not allowed.	COMMIT, SAVEPOINT, and ROLLBACK are allowed.

ORACLE

Comparison of Database Triggers and Stored Procedures

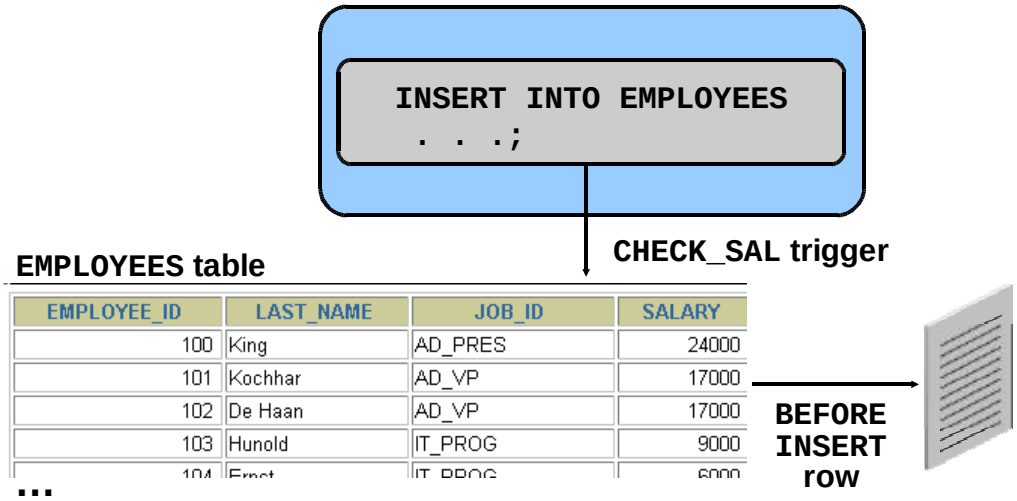
There are differences between database triggers and stored procedures:

Database Trigger	Stored Procedure
Invoked implicitly	Invoked explicitly
COMMIT, ROLLBACK, and SAVEPOINT statements are not allowed within the trigger body. It is possible to commit or roll back indirectly by calling a procedure, but it is not recommended because of side effects to transactions.	COMMIT, ROLLBACK, and SAVEPOINT statements are permitted within the procedure body.

Triggers are fully compiled when the CREATE TRIGGER command is issued and the executable code is stored in the data dictionary.

Note: If errors occur during the compilation of a trigger, the trigger is still created.

Comparison of Database Triggers and Oracle Forms Triggers



Comparison of Database Triggers and Oracle Forms Triggers

Database triggers are different from Forms Builder triggers.

Database Trigger	Forms Builder Trigger
Executed by actions from any database tool or application	Executed only within a particular Forms Builder application
Always triggered by a SQL DML, DDL, or a certain database action	Can be triggered by navigating from field to field, by pressing a key, or by many other actions
Is distinguished as either a statement or row trigger	Is distinguished as a statement or row trigger
Upon failure, causes the triggering statement to roll back	Upon failure, causes the cursor to freeze and may cause the entire transaction to roll back
Fires independently of, and in addition to, Forms Builder triggers	Fires independently of, and in addition to, database triggers
Executes under the security domain of the author of the trigger	Executes under the security domain of the Forms Builder user

Managing Triggers

- Disable or reenable a database trigger:

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

- Disable or reenable all triggers for a table:

```
ALTER TABLE table_name DISABLE | ENABLE
  ALL TRIGGERS
```

- Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```


Managing Triggers

A trigger has two modes or states: **ENABLED** and **DISABLED**. When a trigger is first created, it is enabled by default. The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.

Disabling a Trigger

- By using the **ALTER TRIGGER** syntax, or disable all triggers on a table by using the **ALTER TABLE** syntax
- To improve performance or to avoid data integrity checks when loading massive amounts of data with utilities such as SQL*Loader. Consider disabling a trigger when it references a database object that is currently unavailable, due to a failed network connection, disk crash, offline data file, or offline tablespace.

Recompiling a Trigger

- By using the **ALTER TRIGGER** command to explicitly recompile a trigger that is invalid
- By issuing an **ALTER TRIGGER** statement with the **COMPILE** option

Removing Triggers

To remove a trigger from the database, use the `DROP TRIGGER` statement:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Note: All triggers on a table are removed when the table is removed.

ORACLE

Removing Triggers

When a trigger is no longer required, use a SQL statement in *i*SQL*Plus to remove it.

Testing Triggers

- Test each triggering data operation, as well as nontriggering data operations.
- Test each case of the `WHEN` clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.

Testing Triggers

Testing code can be a time-consuming process. Do the following when testing triggers:

- Ensure that the trigger works properly by testing a number of cases separately:
  - Test the most common success scenarios first.
  - Test the most common failure conditions to see that they are properly managed.
- The more complex the trigger, the more detailed your testing is likely to be. For example, if you have a row trigger with a `WHEN` clause specified, then you should ensure that the trigger fires when the conditions are satisfied. Or, if you have cascading triggers, you need to test the effect of one trigger on the other and ensure that you end up with the desired results.
- Use the `DBMS_OUTPUT` package to debug triggers.

Summary

In this lesson, you should have learned how to:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers

ORACLE

Summary

This lesson covered creating database triggers that execute before, after, or instead of a specified DML operation. Triggers are associated with database tables or views. The `BEFORE` and `AFTER` timings apply to DML operations on tables. The `INSTEAD OF` trigger is used as a way to replace DML operations on a view with appropriate DML statements against other tables in the database.

Triggers are enabled by default but can be disabled to suppress their operation until enabled again. If business rules change, triggers can be removed or altered as required.

Practice 10: Overview

This practice covers the following topics:

- **Creating row triggers**
- **Creating a statement trigger**
- **Calling procedures from a trigger**

Practice 10: Overview

You create statement and row triggers in this practice. You create procedures that are invoked from the triggers.

Practice 10

1. **The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.**
   a. **Write a procedure called CHECK_SALARY that accepts two parameters, one for an employee's job ID string and the other for the salary. The procedure uses the job ID to determine the minimum and maximum salary for the specified job. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>". Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.**
   b. **Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row. The trigger must call the CHECK_SALARY procedure to carry out the business logic. The trigger should pass the new job ID and salary to the procedure parameters.**
2. **Test the CHECK_SAL_TRG using the following cases:**
   a. **Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh to department 30. What happens and why?**
   b. **Update the salary of employee 115 to $2,000. In a separate update operation, change the employee job ID to HR_REP. What happens in each case?**
   c. **Update the salary of employee 115 to $2,800. What happens?**
3. **Update the CHECK_SALARY_TRG trigger to fire only when the job ID or salary values have actually changed.**
   a. **Implement the business rule using a WHEN clause to check whether the JOB_ID or SALARY values have changed.**
      Note: Make sure that the condition handles the NULL in the OLD.column_name values if an INSERT operation is performed; otherwise, an insert operation will fail.
   b. **Test the trigger by executing the EMP_PKG.ADD_EMPLOYEE procedure with the following parameter values: first_name='Eleanor', last name='Beh', email='EBEH', job='IT_PROG', sal=5000.**
   c. **Update employees with the IT_PROG job by incrementing their salary by $2,000. What happens?**

   d. **Update the salary to $9,000 for Eleanor Beh.**
      Hint: Use an UPDATE statement with a subquery in the WHERE