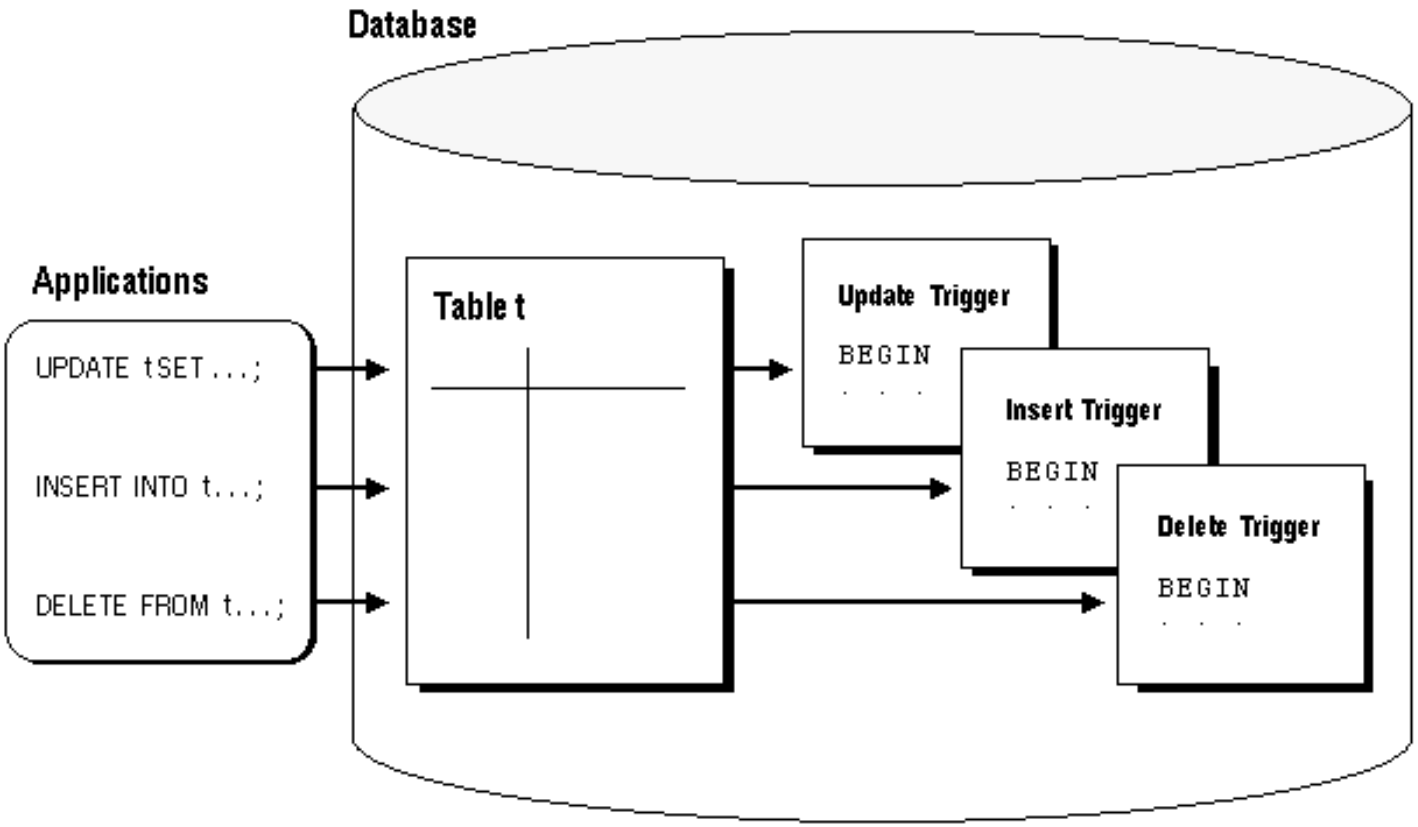
Database Systems

Outline

- Describe different types of triggers
- Describe database triggers and their use
- Create database triggers
- Describe database trigger firing rules
- Remove database triggers

Introduction



Types of Triggers

- A trigger:
  - Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database
- Executes implicitly whenever a particular event takes place
- Can be either:
  - 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

Guidelines for Designing Triggers

- Design triggers to:
  - Perform related actions
  - Centralize global operations
- Do not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers
- 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

Database Trigger: Example

Couse_Id	Marks	…	…

Check_Marks Trigger

Creating DML Triggers

- A triggering statement contains:
- Trigger timing
  - For table: before, after
  - For view: instead of
- Triggering event: insert, update, or delete
- Table name: On table, view
- Trigger type: Row or statement
- WHEN clause: Restricting condition
- Trigger body: PL/SQL block

DML Trigger Components: 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.

DML Trigger Components: Triggering user event

- **Which DML statement causes the trigger to execute?**
- INSERT
- UPDATE (**specify a column list**)
- DELETE

DML Trigger Components: Trigger type

- **Should the trigger body execute for each row the statement affects or only once?**
- **Statement**
  - The trigger body executes once for the triggering event.
  - This is the default.
  - A statement trigger fires once, even if no rows are affected at all.
- **Row:**
  - The trigger body executes once for each row affected by the triggering event.
  - A row trigger is not executed if the triggering event affects no rows.

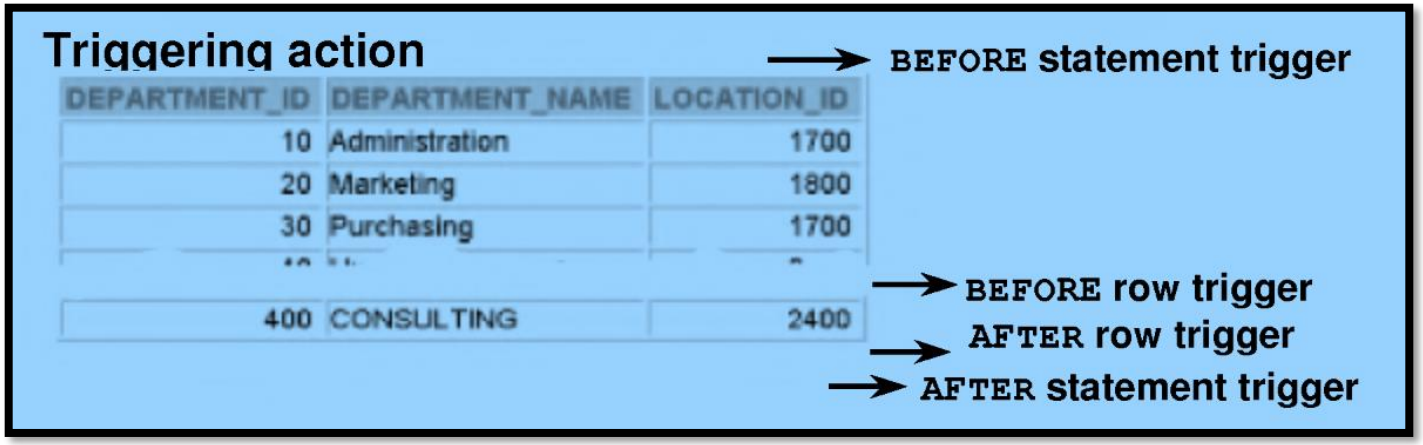
DML Trigger Components: Trigger body

- **What action should the trigger perform?**
- The trigger body is a PL/SQL block or a call to a procedure.
- Row triggers use correlation names to access the **old and new column** values of the row being processed by the trigger.

Firing Sequence

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

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



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;
```

Creating DML Statement Triggers

When the trigger fires

Name of the trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      event1 [OR event2 OR event3]
         ON table_name
trigger_body
```

Cause of Trigger Firing

Table/ View associated with the trigger

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

Example

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN (' SAT' , ' SUN' ) ) OR
    (TO_CHAR(SYSDATE,'HH2 4:MI') NOT BETWEEN '08:00' AND '18:00')
    THEN
        RAISE_APPLICATION_ERROR (-20500,'You may
                insert into EMPLOYEES table only
                during business hours.');
    END IF;
END;
```

```
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);
```

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;
```

Creating a DML Row Trigger

Specifies correlation names for the old and new values of the current row (The default values are OLD and NEW)

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

Designates that the trigger is a row trigger

Specifies the trigger restriction

Example

```
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 this amount');
     END IF;
END;
/
```

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

Create a trigger to allow only certain employees to be able to earn a salary of more than 15,000.

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_table (user_name, timestamp,
     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

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

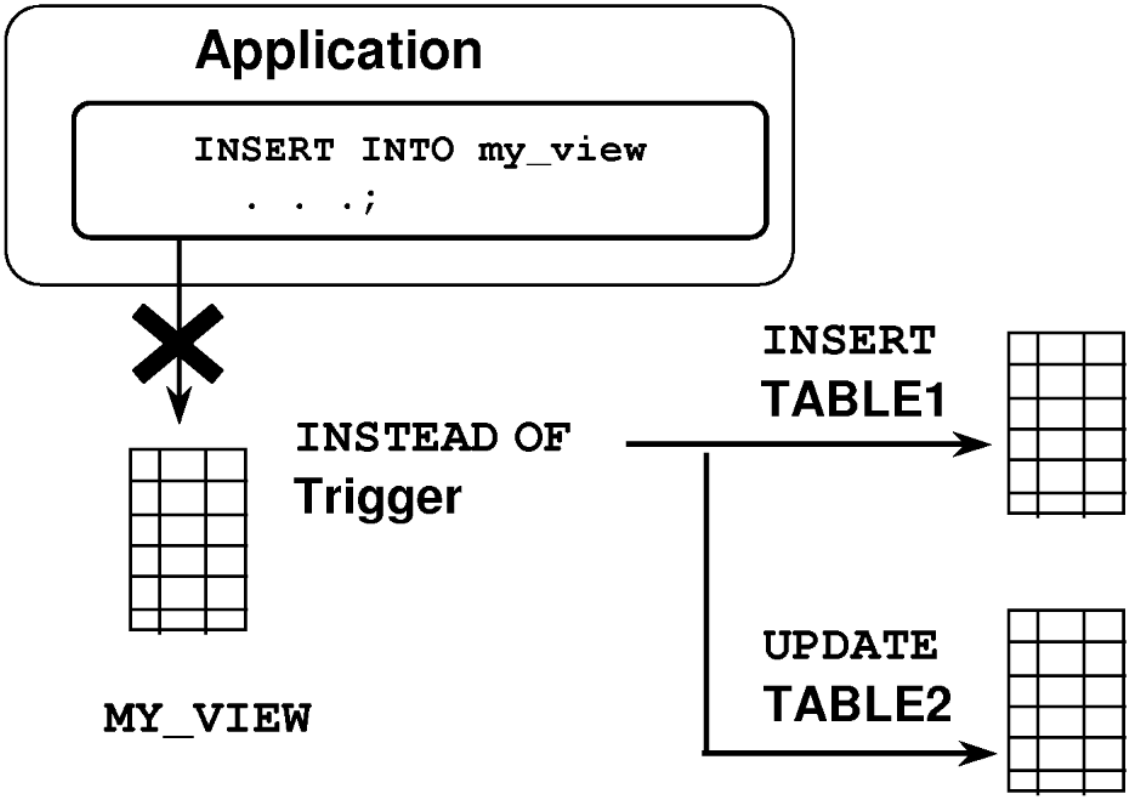
- The OLD and NEW qualifiers are available only in ROW triggers.
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.

Restricting a Row Trigger

```
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;
```

- To restrict the trigger action to those rows that satisfy a certain condition
- The NEW qualifier cannot be prefixed with a colon in the WHEN clause because the WHEN clause is outside the PL/SQL blocks.

INSTEAD of Triggers



Use INSTEAD OF triggers to modify data in which the DML statement has been issued against an inherently non-updatable view.

Creating an INSTEAD OF Trigger

Indicates that the trigger belongs to a view

```
CREATE [OR REPLACE] TRIGGER trigger_name
  INSTEAD OF
    event1 [OR event2 OR event3]
      ON view_name
   [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
trigger_body
```

Indicates the view associated with trigger

Designates the trigger to be a row trigger

Example

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

```
CREATE TABLE new_depts AS
    SELECT d.department_id, d.department_name, d.location_id,
       sum(e.salary) tot_dept_sal
    FROM employees e, departments d
    WHERE e.department_id = d.department_id
    GROUP BY d.department_id, d.department_name, d.location_id;
```

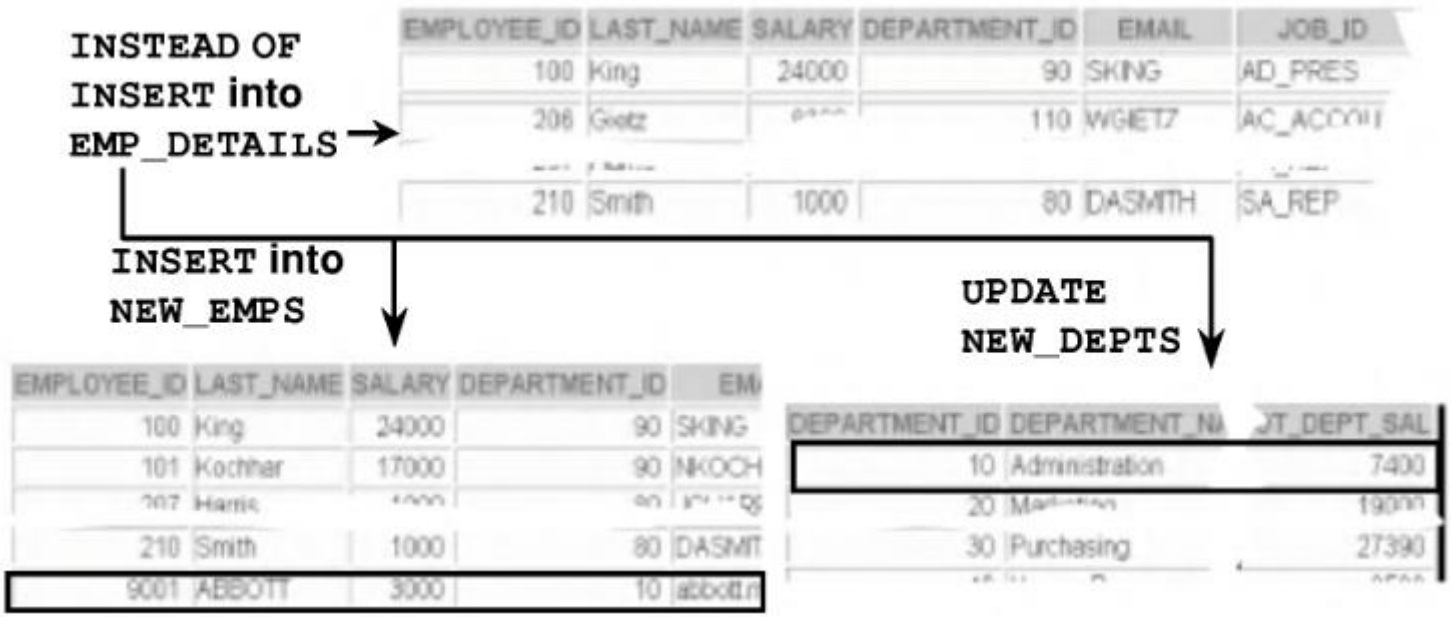
```
CREATE VIEW emp_details AS
     SELECT e.employee_id, e.last_name, e.salary, e.department_id,
            e.email, e.job_id, d.department_name, d.location_id
     FROM employees e, departments d
     WHERE e.department_id = d.department_id;
```

```
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, :NEW.email, :New.job_id, SYSDATE);
                UPDATE new_depts SET tot_dept_sal = tot_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 tot_dept_sal = tot_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 tot_dept_sal = tot_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 tot_dept_sal = tot_dept_sal - :OLD.salary WHERE
                department_id = :OLD.department_id;
                UPDATE new_depts SET tot_dept_sal = tot_dept_sal + :NEW.salary WHERE
                department_id = :NEW.department_id;
        END IF;
END;
```

Illustration: INSTEAD OF

Insert into **emp_details** that is based on employees and DEPARTMENTS tables

```
INSERT INTO emp_details(employee_id, ... )
VALUES(9001,'ABBOTT',3000,10,'abbott.mail.com','HR_MAN');
```



Database Triggers Vs. Stored Procedures

Triggers

- Defined with create trigger
- Data dictionary contains source code in **user_triggers**
- Implicitly invoked
- commit, savepoint, and rollback are not allowed

Procedures

- Defined with create procedure
- Data dictionary contains source code in **user_source**
- Explicitly invoked
- COMMIT, SAVEPOINT, and ROLLBACK are allowed

Managing Triggers

- Disable or reenable a database trigger:

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

- Disable or re-enable all triggers for a table:

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

- Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```

DROP TRIGGER Syntax

- To remove a trigger from the database, use the drop trigger syntax:

```
DROP TRIGGER trigger_name;
```

- Example:

```
DROP TRIGGER secure_emp;
```

All triggers on a table are dropped when the table is dropped.

Trigger Test Cases

- Test each triggering data operation, as well as non-triggering 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 upon other triggers.
- Test the effect of other triggers upon the trigger.

Trigger Execution Model and Constraint Checking

1. Execute all before statement triggers.
2. Loop for each row affected:
    1. Execute all before row triggers.
    2. Execute all after row triggers.
3. Execute the DML statement and perform integrity constraint checking.
4. Execute all after statement triggers.

Example

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

```
CREATE OR REPLACE TRIGGER constr_emp_trig
 AFTER UPDATE ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO departments
    VALUES (999, 'dept999', 140, 2400);
END;
```

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

Creating Triggers on DDL Statements

Fire the trigger whenever a CREATE statement adds a new database object to the dictionary

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      [ddl_event1 [OR ddl_event2 OR ...]]
         ON {DATABASE|SCHEMA}
trigger_body
```

Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      [database_event1 [OR database_event2 OR ...]]
         ON {DATABASE|SCHEMA}
trigger_body
```

Database_event	Possible Values
AFTER SERVERERROR	Causes the Oracle server to fire the trigger whenever a server error message is logged
AFTER LOGON	Causes the Oracle server to fire the trigger whenever a user logs on to the database
BEFORE LOGOFF	Causes the Oracle server to fire the trigger whenever a user logs off the database
AFTER STARTUP	Causes the Oracle server to fire the trigger whenever the database is opened
BEFORE SHUTDOWN	Causes the Oracle server to fire the trigger whenever the database is shut down

CALL Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
      ON table_name
  [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
  [WHEN condition]
CALL procedure_name;
```

```
CREATE TRIGGER salary_check
    BEFORE UPDATE OF salary, job_id ON employees
    FOR EACH ROW
    WHEN (NEW.job_id <> 'AD_PRES')
        CALL check_sal(:NEW.job_id, :NEW.salary)
```

There is no semicolon at the end of the CALL statement.

Mutating Table

- A mutating table is a table that is currently being modified by an UPDATE, DELETE, or INSERT statement
- A table that might need to be updated by the effects of a declarative DELETE CASCADE referential integrity action.
- A table is not considered mutating for STATEMENT triggers.
- The triggered table itself is a mutating table, as well as any table referencing it with the FOREIGN KEY constraint.
- This restriction prevents a row trigger from seeing an inconsistent set of data.

Example

Guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO  v_minsalary, v_maxsalary
   FROM  employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR
     :NEW.salary > v_maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
```

When an employee record is updated, the **CHECK_SALARY** trigger is fired for each row that is updated. The trigger code queries the same table that is being updated, i.e. ***EMPLOYEES table is mutating table***.

Implementing Triggers

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Enforcing Referential Integrity within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
     REFERENCES departments(department_id)
ON DELETE CASCADE;
```

```
CREATE OR REPLACE TRIGGER cascade_updates
 AFTER UPDATE OF department_id ON departments
 FOR EACH ROW
BEGIN
  UPDATE employees
   SET employees.department_id=:NEW.department_id
   WHERE employees.department_id=:OLD.department_id;
  UPDATE job_history
   SET department_id=:NEW.department_id
   WHERE department_id=:OLD.department_id;
END;
```

Benefits of Database Triggers

- Improved data security:
  - Provide enhanced and complex security checks
  - Provide enhanced and complex auditing
- Improved data integrity:
  - Enforce dynamic data integrity constraints
  - Enforce complex referential integrity constraints
  - Ensure that related operations are performed together implicitly

Viewing Trigger Information

- You can view the following trigger information:
  - **user_objects** data dictionary view: Object information
  - **user_triggers** data dictionary view: The text of the trigger
  - **user_errors** data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger

Using USER_TRIGGERS

Column	Column Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	The type is BEFORE, AFTER, INSTEAD OF
TRIGGERING_EVENT	The DML operation firing the trigger
TABLE_NAME	Name of the database table
REFERENCING_NAMES	Name used for :OLD and :NEW
WHEN_CLAUSE	The when_clause used
STATUS	The status of the trigger
TRIGGER_BODY	The action to take

Thank you