11

Applications for Triggers

Objectives

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

- Create additional database triggers
- Explain the rules governing triggers
- Implement triggers

Lesson Aim

In this lesson, you learn how to create more database triggers and learn the rules governing triggers. You also learn about the many applications of triggers.

Creating Database Triggers

- Triggering a user event:
  - CREATE, ALTER, or DROP
  - Logging on or off
- Triggering database or system event:
  - Shutting down or starting up the database
  - A specific error (or any error) being raised

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on data definition language (DDL) statements, or a user logging on or off, can also be defined at either the database level or schema level. Triggers on data manipulation language (DML) statements are defined on a specific table or a 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.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- Any error that occurs

Creating Triggers on DDL Statements

Syntax:

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


Creating Triggers on DDL Statements

DDL_Event	Possible Values
CREATE	Causes the Oracle server to fire the trigger whenever a CREATE statement adds a new database object to the dictionary
ALTER	Causes the Oracle server to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary
DROP	Causes the Oracle server to fire the trigger whenever a DROP statement removes a database object in the data dictionary

The trigger body represents a complete PL/SQL block.

You can create triggers for these events on DATABASE or SCHEMA. You also specify BEFORE or AFTER for the timing of the trigger.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

Creating Triggers on System Events

Syntax:

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


Create Trigger Syntax

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

You can create triggers for these events on DATABASE or SCHEMA, except SHUTDOWN and STARTUP, which apply only to DATABASE.

LOGON and LOGOFF Triggers: Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

LOGON and LOGOFF Triggers: Example

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify `ON SCHEMA`, the trigger fires for the specific user. If you specify `ON DATABASE`, the trigger fires for all users.

CALL Statements

```
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 OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution
/
```

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

ORACLE

CALL Statements

A CALL statement enables you to call a stored procedure, rather than code the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

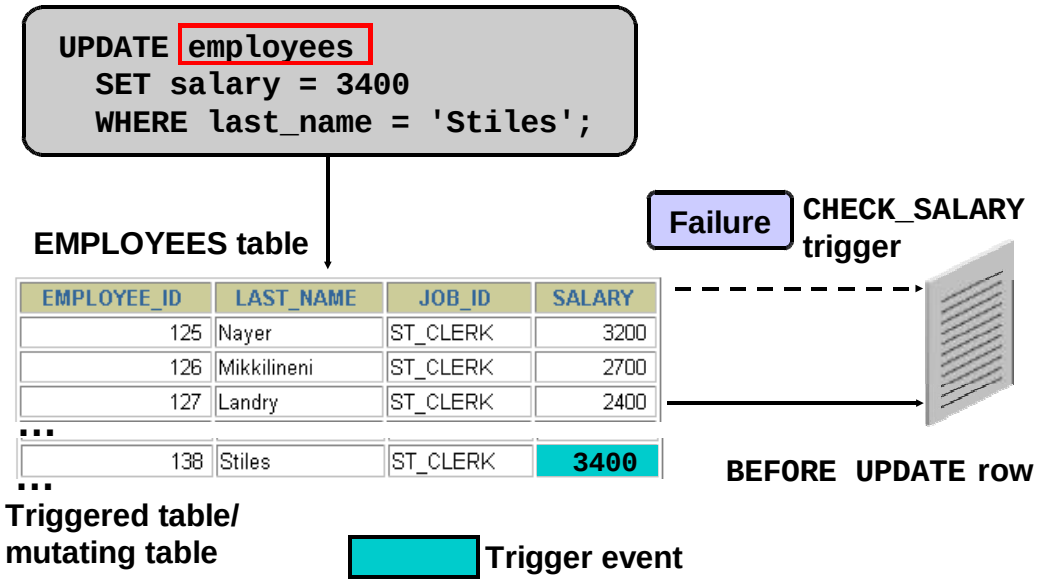
The call can reference the trigger attributes :NEW and :OLD as parameters, as in the following example:

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

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

In the preceding example, the trigger calls a check_salary procedure. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Reading Data from a Mutating Table



ORACLE

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of `ON DELETE CASCADE`.

Mutating Table

A mutating table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or a table that might need to be updated by the effects of a declarative `DELETE CASCADE` referential integrity action. For `STATEMENT` triggers, a table is not considered a mutating table.

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.

Mutating Table: Example

```
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
  minsalary employees.salary%TYPE;
  maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO  minsalary, maxsalary
   FROM  employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < minsalary OR
     :NEW.salary > maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
/
```

ORACLE

Mutating Table: Example

The CHECK_SALARY trigger in the example attempts to 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.

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. Therefore, it is said that the EMPLOYEES table is a mutating table.

Mutating Table: Example

```
UPDATE employees
 SET salary = 3400
 WHERE last_name = 'Stiles';
```

```
UPDATE employees
       *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "PLSQL.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'PLSQL.CHECK_SALARY'
```


Mutating Table: Example (continued)

In the example, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a run-time error. The `EMPLOYEES` table is mutating, or in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Possible Solutions

Possible solutions to this mutating table problem include the following:

- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.
- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a `BEFORE` statement trigger.

Depending on the nature of the problem, a solution can become more convoluted and difficult to solve. In this case, consider implementing the rules in the application or middle tier and avoid using database triggers to

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**

ORACLE

Benefits of Database Triggers

You can use database triggers:

- **As alternatives to features provided by the Oracle server**
- **If your requirements are more complex or more simple than those provided by the Oracle server**
- **If your requirements are not provided by the Oracle server at all**

Managing Triggers

The following system privileges are required to manage triggers:

- **The `CREATE/ALTER/DROP (ANY) TRIGGER` privilege that enables you to create a trigger in any schema**
- **The `ADMINISTER DATABASE TRIGGER` privilege that enables you to create a trigger on `DATABASE`**
- **The `EXECUTE` privilege (if your trigger refers to any objects that are not in your schema)**

Note: Statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

ORACLE

Managing Triggers

To create a trigger in your schema, you need the `CREATE TRIGGER` system privilege, and you must either own the table specified in the triggering statement, have the `ALTER` privilege for the table in the triggering statement, or have the `ALTER ANY TABLE` system privilege. You can alter or drop your triggers without any further privileges being required.

If the `ANY` keyword is used, then you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the `EXECUTE` privilege on the referenced procedures, functions, or packages, and not through roles. As with stored procedures, statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

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

Business Application Scenarios for Implementing Triggers

You can use triggers for:

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

Note: Appendix C covers each of these examples in more detail.

ORACLE

Business Application Scenarios for Implementing Triggers

Develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

Feature	Oracle server
Security	The Oracle server allows table access to users or roles. Triggers allow table access according to data values.
Auditing	The Oracle server tracks data operations on tables. Triggers track values for data operations on tables.
Data integrity	The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.
Referential integrity	The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.
Table replication	The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.
Derived data	The Oracle server computes derived data values manually. Triggers compute derived data values automatically.
Event logging	The Oracle server logs events explicitly. Triggers log events transparently.

Viewing Trigger Information

You can view the following trigger information:

- `USER_OBJECTS` data dictionary view: Object information
- `USER_TRIGGERS` data dictionary view: Text of the trigger
- `USER_ERRORS` data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger

Viewing Trigger Information

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The `USER_OBJECTS` view contains the name and status of the trigger and the date and time when the trigger was created.

The `USER_ERRORS` view contains the details about the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The `USER_TRIGGERS` view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The `SELECT Username FROM USER_USERS;` statement gives the name of the owner of the trigger, not the name of the user who is updating the table.

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

* Abridged column list

ORACLE

Using USER_TRIGGERS

If the source file is unavailable, then you can use *i*SQL*Plus to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object.

Listing the Code of Triggers

```
SELECT trigger_name, trigger_type, triggering_event,
       table_name, referencing_names,
       status, trigger_body
FROM   user_triggers
WHERE  trigger_name = 'RESTRICT_SALARY';
```

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	REFERENCING_NAMES	WHEN_CLAUS	STATUS	TRIGGER_BODY
RESTRICT_SALARY	BEFORE EACH ROW	INSERT OR UPDATE	EMPLOYEES	REFERENCING NEW AS NEW OLD AS OLD		ENABLED	BEGIN IF NOT (:NEW.JOB_ ID IN ('AD_PRES ', 'AD_VP')) AND :NE W.SAL

ORACLE

Example

Use the `USER_TRIGGERS` data dictionary view to display information about the `RESTRICT_SALARY` trigger.

Summary

In this lesson, you should have learned how to:

- **Use advanced database triggers**
- **List mutating and constraining rules for triggers**
- **Describe real-world applications of triggers**
- **Manage triggers**
- **View trigger information**

Practice 11: Overview

This practice covers the following topics:

- Creating advanced triggers to manage data integrity rules
- Creating triggers that cause a mutating table exception
- Creating triggers that use package state to solve the mutating table problem

Practice 11: Overview

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their job. You create a trigger for this rule.

During this process, your new triggers cause a cascading effect with triggers created in the practice section of the lesson titled "Creating Triggers." The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

Practice 11

1. **Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salary. Implement this requirement through a package procedure called by a trigger on the `JOBS` table. When you attempt to update the minimum salary in the `JOBS` table and try to update the employees' salary, the `CHECK_SALARY` trigger attempts to read the `JOBS` table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.**

   a. **Update your `EMP_PKG` package (from Practice 7) by adding a procedure called `SET_SALARY` that updates the employees' salaries. The procedure accepts two parameters: the job ID for those salaries that may have to be updated, and the new minimum salary for the job ID. The procedure sets all the employees' salaries to the minimum for their jobs if their current salaries are less than the new minimum value.**

   - **Create a row trigger named `UPD_MINSALARY_TRG` on the `JOBS` table that invokes the `EMP_PKG.SET_SALARY` procedure, when the minimum salary in the `JOBS` table is updated for a specified job ID.**
   - **Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is `'IT_PROG'`. Then update the minimum salary in the `JOBS` table to increase it by $1,000. What happens?**

2. **To resolve the mutating table issue, you create a `JOBS_PKG` to maintain in memory a copy of the rows in the `JOBS` table. Then the `CHECK_SALARY` procedure is modified to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, a `BEFORE INSERT OR UPDATE` statement trigger must be created on the `EMPLOYEES` table to initialize the `JOBS_PKG` package state before the `CHECK_SALARY` row trigger is fired.**

   a. **Create a new package called `JOBS_PKG` with the following specification:**

   ```
   PROCEDURE initialize;

   FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;

   FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;

   PROCEDURE set_minsalary(jobid VARCHAR2,min_salary NUMBER);

   PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary NUMBER);
   ```

   b. **Implement the body of the `JOBS_PKG`, where:**
   You declare a private PL/SQL index-by table called `jobs_tabtype` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
   You declare a private variable called `jobstab` based on the `jobs_tabtype`.
   The `INITIALIZE` procedure reads the rows in the `JOBS` table

The SET_MINSALARY procedure uses its jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
The SET_MAXSALARY procedure uses its jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.

c. Copy the CHECK_SALARY procedure from Practice 10, Exercise 1a, and modify the code by replacing the query on the JOBS table with statements to set the local minsal and maxsal variables with values from the JOBS_PKG data by calling the appropriate GET_*SALARY functions. This step should eliminate the mutating trigger exception.

d. Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.

e. Test the code changes by executing the query to display the employees who are programmers, then issue an update statement to increase the minimum salary of the IT_PROG job type by 1000 in the JOBS table, followed by a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

3. Because the CHECK_SALARY procedure is fired by the CHECK_SALARY_TRG before inserting or updating an employee, you must check whether this still works as expected.

   a. Test this by adding a new employee using EMP_PKG.ADD_EMPLOYEE with the following parameters: ('Steve', 'Morse', 'SMORSE', and sal => 6500). What happens?

   b. To correct the problem encountered when adding or updating an employee, create a BEFORE INSERT OR UPDATE statement trigger called EMPLOYEE_INITJOBS_TRG on the EMPLOYEES table that calls the JOBS_PKG.INITIALIZE procedure. Use the CALL syntax in the trigger body.

   c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the employees table by displaying the employee ID, first and last names, salary, job ID, and department ID.