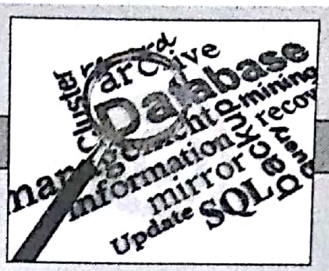
Chapter 34



DATABASE TRIGGERS

CHAPTER OBJECTIVES

In this chapter you will learn:

- Concept of Triggers
- Use of Database Triggers
- Database Trigger V/s Procedure
- Parts of a Trigger
- Types of Triggers
- Syntax for creating a Trigger
- Applications Using Database Triggers
- Error Handling in Triggers
- Enabling and Disabling Triggers
- Dropping Triggers

34.1 INTRODUCTION TO TRIGGERS

A database trigger is a stored procedure that is fired when an INSERT, UPDATE, or DELETE statement is issued against the associate table. The name trigger is appropriate, as these are triggered (fired) whenever the above-mentioned commands are executed. A trigger defines an action the database should take when some database related event occurs as shown in figure 34.1. A trigger can include SQL and PL/SQL statements to execute it as a unit and it can invoke other stored procedures. Triggers may use to provide referential integrity, to enforce complex business rules, or to audit changes to data. The code within a trigger, called the trigger body is made up of PL/SQL blocks.

A trigger is automatically executed without any action required by the user. A stored procedure on the other hand needs to be explicitly invoked. This is the main difference between a trigger and a stored procedure.

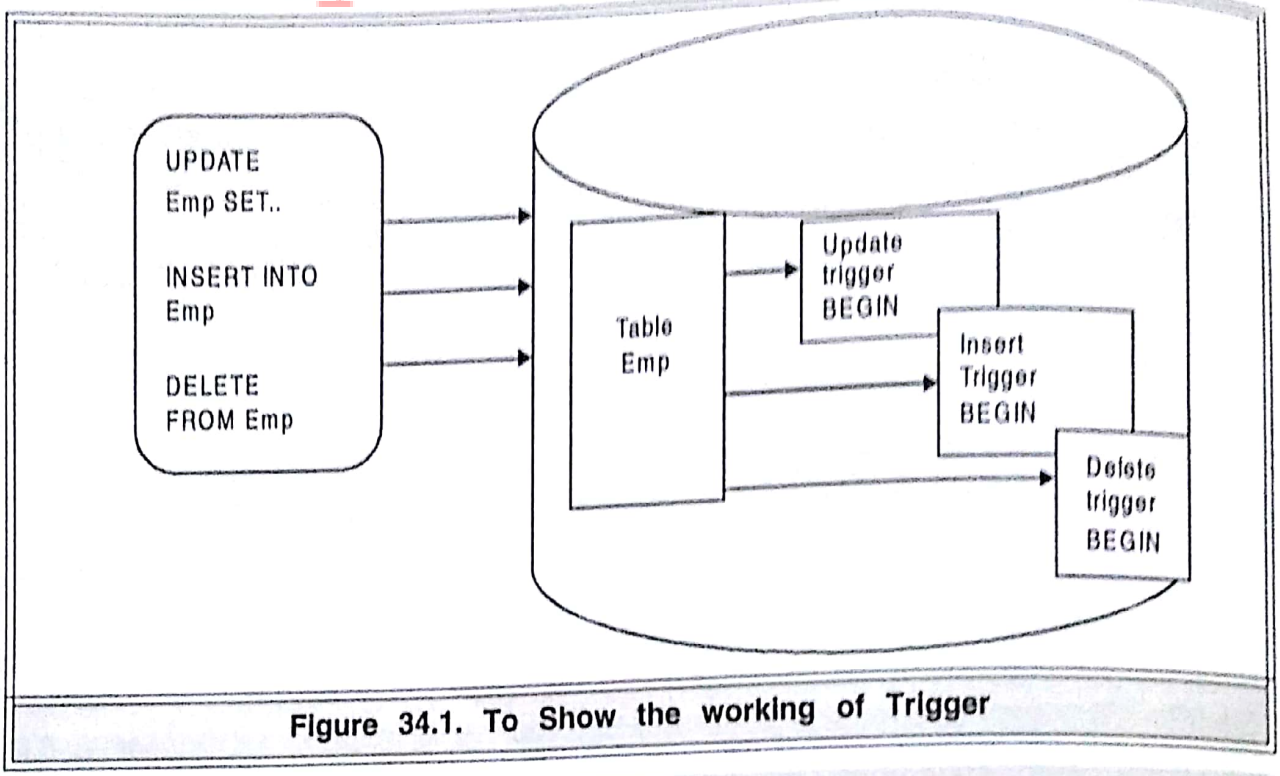


Figure 34.1. To Show the working of Trigger

34.2 USE OF DATABASE TRIGGERS

Database triggers can be used for the following purposes:

- To derive column values automatically.
- To enforce complex integrity constraints.
- To enforce complex business rules.
- To customize complex security authorizations.
- To maintain replicate tables.
- To audit data modifications.

34.3 DATABASE TRIGGER V/S PROCEDURE

Here is the comparison between database Triggers and Procedures:

Database Triggers	Procedures
Triggers do not accept parameters.	Procedures can accept parameters.
A trigger is automatically executed without any action required by the user.	A stored procedure on the other hand needs to be explicitly invoked.

34.4 PARTS OF A TRIGGER

A database trigger has three parts:

- Triggering event or Statement.
- Trigger constraint (Optional).
- Trigger action.

Triggering Event or Statement

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

Trigger Constraint or Restriction

A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire. The trigger actions are not executed if the trigger restriction evaluates to FALSE. A trigger restriction is an option available for triggers that are fired for each row. Its function is to conditionally control the execution of a trigger. A trigger restriction is specified using a WHEN clause. It is an optional part of trigger.

Trigger Action

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

34.5 TYPES OF TRIGGERS

A trigger's type is defined by the type of triggering transaction and by the level at which the trigger is executed. Oracle has the following types of triggers depending on the different applications:

- ✦ Row Level Triggers
- ✦ Statement Level Triggers
- ✦ Before Triggers
- ✦ After Triggers

Here is a brief description of above triggers:

Row Level Triggers

Row level triggers execute once for each row in a transaction. The commands of row level triggers are executed on all rows that are affected by the command that enables the trigger. 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 the triggering statement affects no rows, the trigger is not executed at all. Row level triggers are created using the FOR EACH ROW clause in the CREATE TRIGGER command.

Application: Consider a case where our requirement is to prevent updation of empno 100 record. Then whenever UPDATE statement update records, there must be PL/SQL block that will be fired automatically by UPDATE statement to check that it must not be 100, so we have to use Row level Triggers for that type of applications.

Statement Level Triggers

Statement level triggers are triggered only once for each transaction. For example when an UPDATE command updates 15 rows, the commands contained in the trigger are executed only once, and not with every processed row. Statement level trigger are the default types of trigger created via the CREATE TRIGGER command.

Application : Consider a case where our requirement is to prevent the DELETE operation during Sunday. For this whenever DELETE statement deletes records, there must

be PL/SQL block that will be fired only once by DELETE statement to check that day must not be Sunday by referencing system date, so we have to use Statement level Trigger which fires only once for above application.

Before and After Trigger

Since triggers are executed by events, they may be set to occur immediately before or after those events. When a trigger is defined, you can specify whether the trigger must occur before or after the triggering event i.e. INSERT, UPDATE, or DELETE commands.

BEFORE trigger execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situation:

- BEFORE triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using a BEFORE trigger, you can eliminate unnecessary processing of the triggering statement. For example: To prevent deletion on Sunday, for this we have to use Statement level before trigger on DELETE statement.
- BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER trigger executes the trigger action after the triggering statement is executed. AFTER triggers are used when you want the triggering statement to complete before executing the trigger action. For example: To perform cascade delete operation, it means that user delete the record from one table and the corresponding records in other tables are deleted automatically by a trigger which is fired after the execution of DELETE statement issued by the user.

When combining the different types of triggering actions, there are mainly 12 possible valid trigger types available to us. The possible configurations are:

- BEFORE INSERT row
- AFTER INSERT row
- BEFORE UPDATE row
- AFTER UPDATE Row
- BEFORE DELETE row
- AFTER DELETE row
- BEFORE INSERT statement
- AFTER INSERT statement
- BEFORE UPDATE statement
- AFTER UPDATE statement
- BEFORE DELETE statement
- AFTER DELETE statement

Here is the list of the triggering actions with their explanations and relative restrictions.

Name	Statement Level	Row Level
BEFORE option	Oracle fires the trigger only once, before executing the triggering statement	Oracle fires the trigger before modifying each row affected by the triggering statement
AFTER option	Oracle fires the trigger only once, after executing the triggering statement	Oracle fires the trigger after modifying each row affected by the triggering statement

Instead of Trigger

Instead of trigger was first featured in Oracle 8. This was something new in the world of triggers. These are triggers that are defined on a view rather than on a table. You can use

INSTEAD OF trigger to tell Oracle what to do *instead of* performing the actions that invoked the trigger.

34.6 SYNTAX FOR CREATING A TRIGGER

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
{| DELETE | [OR] INSERT | [OR] UPDATE [OF column_name, ...]
ON table_name
[REFERENCING {OLD AS old, NEW AS new}]
[FOR EACH ROW [WHEN condition ]
DECLARE
Variable declaration;
Constant declaration;
BEGIN
        PL/SQL subprogram body;
[EXCEPTION
        exception PL/SQL block;]
END;
```

Triggering event

Triggering restriction

Triggering action

Here:

REFERENCING : Specified correlation names. The user could use the Correlation names in the PL/SQL block and WHEN clause of a row trigger to refer specifically to old and new values of the current now. The default correlation names are OLD and NEW. If the row is associated with a table named OLD or NEW, this clause can be used to specify different names to avoid confusion between the table name and the correlation name.

WHEN Specifies the trigger restriction. This condition has to be satisfied to fire the trigger. This condition can be specified for the ROW TRIGGER.

NOTE: If an error occurs during the compilation of trigger, an invalid trigger is created. The Oracle engine displays a message after creation that the trigger was created with compilation errors. It does not display the errors. These errors can be viewed using the following commands:

```
SELECT *FROM user_errors;
        Or
Show errors;
```

34.7 APPLICATIONS USING DATABASE TRIGGERS

Here are some examples to design application with the use of triggers:

Example 34.1: ***To Create a trigger for the emp table, which makes the entry in ENAME column in uppercase.***

Solution:

```
CREATE OR REPLACE TRIGGER upper_trigger
BEFORE INSERT OR UPDATE OF ename ON emp
FOR EACH ROW
```

```
BEGIN
    :new.ename : = UPER (:new.ename);
END;
```

Description: The above example also illustrates the use of the new keyword, which refers to the new value of the column, and the old keyword, which refers to the old values of the column. When referencing the new and old keywords in the PL/SQL block, they are preceded by colon(:)

Example 34.2: ***To Create a trigger on the emp table, which shows the old values and new value of ename after any updation on ename of emp table.***

Solution:

```
CREATE OR REPLACE TRIGGER EMP_UPDATE
AFTER UPDATE OF ENAME ON EMP FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('OLD NAME: ' ||:OLD.ENAME);
  DBMS_OUTPUT.PUT_LINE('NEW NAME:' ||:NEW.ENAME);
END;
```

Example 34.3: ***To Create a trigger on the emp table, which store the empno and operation in table auditor for each operation i.e. Insert, Update and Delete.***

Solution:

```
CREATE OR REPLACE TRIGGER EMP_AUDIT
AFTER INSERT OR UPDATE OR DELETE ON EMP
FOR EACH ROW
BEGIN
IF INSERTING THEN
    INSERT INTO AUDITOR VALUES(:NEW.EMPNO,'INSERT');
ELSIF UPDATING THEN
   INSERT INTO AUDITOR VALUES(:NEW.EMPNO,'UPDATE');
   ELSIF DELETING THEN
   INSERT INTO AUDITOR VALUES(:OLD.EMPNO,'DELETE');
END IF;
END;
```

NOTE: The same functionality can be achieved with the BEFORE type trigger also.

34.8 ERROR HANDLING IN TRIGGERS

The Oracle engine provides a procedure named *raise_application_error* that allows programmers to issue user-defined error messages.

Syntax:

```
RAISE_APPLICATION_ERROR (error_number, message);
```

Here:

error_number	It is a negative integer in the range −20000 to −20999
message	It is a character string up to 2048 bytes in length

This procedure terminates procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message. Following example makes use of the RAISE_APPLICATION_ERROR.

Example 34.4: ***To Create a trigger so that no operation can be performed on emp table on Sunday.***

Solution:

```
CREATE OR REPLACE TRIGGER EMP_SUNDAY
BEFORE INSERT OR UPDATE OR DELETE ON EMP
BEGIN
    IF RTRIM(UPPER(TO_CHAR(SYSDATE,'DAY')))
    ='SUNDAY' THEN
    RAISE_APPLICATION_ERROR(-20022,'NO OPERATION CAN
        BE PERFORMED ON SUNDAY');
    END IF;
END;
```

Example 34.5: ***To create a trigger, which verify that no record has the commission value greater than, salary of an employee in table emp.***

Solution:

```
CREATE OR REPLACE TRIGGER REC_CHECK
BEFORE INSERT OR UPDATE ON EMP FOR EACH ROW
BEGIN
    IF :NEW.SAL<:NEW.COMM THEN
        RAISE_APPLICATION_ERROR
        (-20000,'RECORD IS ILLEGAL');
    END IF;
END;
```

Example 34.6: ***To create a trigger, which verify that updated salary of employee must be greater than his/her previous salary.***

Solution:

```
CREATE OR REPLACE TRIGGER UPDATE_CHECK
BEFORE UPDATE ON EMP
FOR EACH ROW
BEGIN
    IF :NEW.SAL<:OLD.SAL THEN
        RAISE_APPLICATION_ERROR(-20000,'NEW SALARY
    CANNOT BE LESS THAN OLD SALARY');
    END IF;
END;
```

Example 34.7 : ***Create a trigger to implement the primary key constraint on column eno of table emptemp (eno, ename, job, sal, deptno).***

Solution:

```
CREATE OR REPLACE TRIGGER PRIMARY_KEY
BEFORE INSERT ON EMPTEMP
FOR EACH ROW
DECLARE
```

```
E EMTEMP.ENO%TYPE;
BEGIN
    IF (:NEW.ENO IS NULL) THEN
    RAISE_APPLICATION_ERROR(-20002, 'PRIMARY KEY
    VIOLATION BECAUSE IT CANNOT BE NULL');
END IF;
SELECT ENO INTO E FROM EMPTEMP WHERE ENO=:NEW.ENO;
RAISE_APPLICATION_ERROR(-20003, 'PRIMARY KEY VIOLATION
BECAUSE IT SHOULD BE UNIQUE');
EXCEPTION
WHEN NO_DATA_FOUND THEN
NULL;
END;
```

Explanation: The trigger will be of row level before type having insert on table EMPTEMP as its firing statement, because the code to validate primary key needs to fired before every insertion of record into the table. Primary should be not null, so with the help of :NEW.ENO system can check that its value cannot be null in the insert statement. If it is null, the system will raise an application error and insert operation cannot be performed. The other constraint needs to be fulfilled by the primary key is unique. It means that if :NEW.ENO is already present into the database, then insert operation cannot be performed and system should raise an application error. This is performed by the SELECT INTO statement which verifies the occurrence of :NEW.ENO value into the database. If record found for this value then application error is raised, and if it is not found then SELECT INTO statement raises NO_DATA_FOUND exception. This condition is handled by the exception hander with NULL statement, i.e., no operation will be executed and insert statement will be executed successfully.

If user executes the following insert statement, the trigger will be fired to restrict the execution of insert statement.

```
INSERT INTO EMPTEMP VALUES(NULL,'Ajay','Clerk',10000,10);
```

The following error message will appear:

```
INSERT INTO EMPTEMP VALUES(NULL,'Ajay','Clerk',10000,10);
            *
ERROR at line 1:
ORA-20002: PRIMARY KEY VIOLATION BECAUSE IT CANNOT BE NULL
ORA-06512: at "PARTEEK.PRIMARY_KEY", line 5
ORA-04088: error during execution of trigger 'PARTEEK.PRIMARY_KEY'
```

If user tries to insert the duplicate eno into EMPTEMP with following insert table, the trigger will be fired to restrict the execution of insert statement.

```
INSERT INTO EMPTEMP VALUES(1,'Ajay','Clerk',10000,10);
```

The following error message will appear:

```
INSERT INTO EMPTEMP VALUES(1,'Ajay','Clerk',10000,10);
            *
ERROR at line 1:
 ORA-20003: PRIMARY KEY VIOLATION BECAUSE IT SHOULD BE UNIQUE
 ORA-06512: at "PARTEEK.PRIMARY_KEY", line 8
 ORA-04088: error during execution of trigger 'PARTEEK.PRIMARY_KEY'
```

Example 34.8: ***Create a trigger to implement the foreign key constraint on column deptno of table emptemp (empno, ename, job, sal, deptno) which refer to the deptno column of table depttemp (deptno, dname, loc).***

Solution:

```
CREATE OR REPLACE TRIGGER FOREIGN_KEY
BEFORE INSERT ON EMPTEMP
FOR EACH ROW
DECLARE
DNO DEPTTEMP.DEPTNO %TYPE;
BEGIN
SELECT DEPTNO INTO DNO FROM DEPTTEMP WHERE
DEPTNO=:NEW.DEPTNO;
NULL;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR(-20004, 'FOREIGN KEY VIOLATED BECAUSE
 VALUE IS NOT FOUND IN THE PARENT TABLE');
END;
```

Explanation: The trigger will be of row level before type having insert on table EMPTEMP as its firing statement, because the code to validate foreign key needs to fired before every insertion of record into the table. The foreign key constraint on deptno column of EMPTEMP table with DEPTTEMP table as its parent table, requires that the value supplied for the deptno column in insert statement on EMPTEMP table should be present in the DEPTTEMP table. If it is not present into parent table DEPTTEMP then system should raise an application error and insert statement should not be executed. This task is performed by the SELECT INTO statement which verifies the occurrence of :NEW.DEPTNO value into dept table. If record is found for this value then NULL statement is executed and no action is performed to restrict the insert statement. But, if the value is not found then SELECT INTO statement will raise NO_DATA_FOUND exception and RAISE_APPLICATION_ERROR function is called to prevent the execution of insert statement.

Now, if user executes the following insert statements, the records will be inserted into the respective tables.

```
INSERT INTO DEPTTEMP VALUES(10,'COMPUTER');
INSERT INTO EMPTEMP VALUES(2,'Ajay','Clerk',10000,10);
```

If user violates the foreign key constraint by inserting a value of deptno that is not present in the master table DEPTTEMP, the system will raise the error as shown below.

```
INSERT INTO EMPTEMP VALUES(3,'Ajay','Clerk',10000,20);
ERROR RAISED:
INSERT INTO EMPTEMP VALUES(3,'Ajay','Clerk',10000,20);
                            *
ERROR at line 1:
ORA-20004: FOREIGN KEY VIOLATED BECAUSE VALUE IS NOT FOUND IN
THE PARENT TABLE
ORA-06512: at "PARTEEK.FOREIGN_KEY", line 8
ORA-04088: error during execution of trigger 'PARTEEK.FOREIGN_KEY'
```

34.9 DICTIONARY INFORMATION ABOUT TRIGGERS

Oracle has provided us many dictionary views to show information about the triggers. These are DBA_TRIGGERS, USER_TRIGGERS and ALL_TRIGGERS. These views basically contain the same columns. Following is the list of columns of USER_TRIGGERS view with their descriptions.

TRIGGER_NAME

Gives the information of name of the trigger.

TRIGGER_TYPE

Gives the information about whether the trigger is a STATEMENT or ROW trigger.

TRIGGERING_EVENT

Gives the information about the event that causes the trigger to fire, which may be either INSERT,DELETE or UPDATE.

TABLE_NAME

The table to which the trigger is associated.

REFERENCING_NAMES

Gives the information of OLD or NEW referencing names you have chosen for the trigger. It contains NULL for STATEMENT trigger.

WHEN_CLAUSE

It contains the trigger restriction i.e. when clause which determines if the trigger will fire for that row. It also contain NULL for STATEMENT trigger.

STATUS

Tells about whether the trigger is ENABLED or DISABLED .

TRIGGER_BODY

Unlike stored procedures only the source code of database trigger is stored in the database. When we query this column of USER_TRIGGERS view it will return the PL/SQL code in the body of your trigger. Its data type is LONG. All these column are present in

all the three types of views mentioned above. With the exception Of OWNER and TABLE_OWNER columns. These two appear only in the DBA_TRIGGERS and ALL_TRIGGERS views.

Let's say if somebody wants to know the Trigger Name, Trigger Type and Trigger Event of all the triggers associated with particular table , the following SQL query will provide this information.

```
SQL > select trigger_name,trigger_type, triggering_event
      from USER_TRIGGERS
            Where TABLE_NAME ='ACCOUNTS';
```

The above query will give the list of triggers associated with ACCOUNTS table with the following attributes (before running the above stated query please make sure that you have created this trigger AUDIT_ACCOUNTS on ACCOUNTS table).

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT
AUDIT_ACCOUNTS	BEFORE EACH ROW	UPDATE OR DELETE

Suppose if we are interested in the detail of particular trigger. The following query can get it.

```
SQL> select trigger_type,triggering_event,table_name,status from
        USER_TRIGGERS where TRIGGER_NAME ='AUDIT_ACCOUNTS';
```

The above query will give the following detail of the AUDIT_ACCOUNTS trigger:

TRIGGER_TYPE TRIGGERING_EVENT TABLE_NAME STATUS

EFORE EACH ROW UPDATE OR DELETE ACCOUNTS ENABLED

Since the body of the trigger is also stored in USER_TRIGGERS view, so we can also see the body of trigger by the following query.

```
SQL > select TRIGGER_BODY from USER_TRIGGERS
        where TRIGGER_NAME ='AUDIT_ACCOUNTS';
```

34.10 ENABLING AND DISABLING TRIGGERS

When a trigger is created it is automatically enabled and is triggered whenever the triggering command and the execution command is true. An enabled trigger executes the trigger body if the triggering statement is issued. To disable the execution use the ALTER TRIGGER command with the DISABLE clause. A disabled trigger does not execute the trigger body even if the triggering statement is issued.

We can disable / enable the trigger by the following syntax:

```
ALTER TRIGGER <trigger name> DISABLE / ENABLE
```

34.11 DROPPING TRIGGERS

Triggers may be dropped via the drop trigger command. In order to drop a trigger, you must either own the trigger or have the DROP ANY TRIGGER system privilege.

Syntax:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER emp_update;
```

Flash Back

A database trigger is a stored procedure that is fired when an INSERT, UPDATE, or DELETE statements is issued against the associate table. The name trigger is appropriate, as these are triggered (fired) whenever the above-mentioned commands are executed. A trigger defines an action the database should take when some database related event occurs.

A trigger is automatically executed without any action required by the user. A stored procedure on the other hand needs to be explicitly invoked. This is the main difference between a trigger and a stored procedure.

Database triggers can be used for the following purposes: To derive column values automatically, to enforce complex integrity constraints, to enforce complex business rules, to customize complex security authorizations, to maintain replicate tables and to audit data modifications.

A database trigger has three parts:

- ✦ Triggering event or Statement.
- ✦ Trigger action
- ✦ Trigger constraint (Optional)

Oracle has the following types of triggers depending on the different applications:

- ✦ Row Level Triggers
- ✦ Before Triggers
- ✦ Statement Level Triggers
- ✦ After Triggers

The Oracle engine provides a procedure named *raise_application_error* that allows programmers to issue user-defined error messages.

Syntax :

RAISE_APPLICATION_ERROR (*error_number, message);*

We can disable / enable the trigger by the following syntax:

ALTER TRIGGER <trigger name> DISABLE / ENAMBLE

Triggers may be dropped via the drop trigger command. In order to drop a trigger, you must either own the trigger or have the DROP ANY TRIGGER system privilege.

Syntax :

DROP TRIGGER trigger_name;

REVIEW QUESTIONS

1. What does Trigger mean? What are its uses and explain its working?
2. Compare Database trigger and procedure?
3. What are the parts of Triggers? Explain the importance of each part?

4. What are the types of Triggers? Explain each type.
5. What are applications where we can use row level triggers?
6. What are applications where we can use statement level triggers?
7. What are applications where we can use before type triggers?
8. What are applications where we can use after type triggers?
9. What is Instead of trigger?
10. What is the syntax to create a Trigger?
11. How can we issue user defined error message during firing of a trigger?
12. How can we enable/disable a trigger?
13. How can we alter the existing trigger?
14. How can we drop a trigger?

HANDS ON SESSION

1. Trigger is a PL/SQL Block that can return value?
   (a) True (b) False
2. Trigger is a database object that can be called Explicitly like Procedure/Functions?
   (b) True (b) False
3. Trigger body can contain Transaction control language statements like (COMMIT, ROLLBACK, and SAVEPOINT).
   (a) True (b) False
4. Which of the following Privileges are required to create a trigger on a table?
   (1) Alter Any Table (2) Create Trigger (3) Create Session
   (4) Create Procedure (5) Alter Database
   (a) 1,2 (b) 1,2,4,5 (c) 2 (d) 2,5
5. Triggers can also act as Data Integrity Constraints?
   (a) True (b) False
6. Once a trigger is defined for a table it can not be disabled?
   (a) True (b) False
7. Which of the following are the triggering events that cause trigger to be fired on a table
   (1) Insert (2) Delete (3) Update (4) Create
   (a) 2,3,4 (b) 1,2,4 (c) 1,2,3 (d) 1,2,3,4
8. Trigger information Is stored in _____ Data Dictionary?
   1. DBA_Triggers 2. All_triggers 3. ALL_Users 4. USER_triggers
   (a) 1,2,3 (b) 1,2,3,4 (c) 2,3,4 (d) 1,2,4
9. Inside Trigger Body Variables cannot be declared using the LONG or LONG RAW data types.
   (a) True (b) False
10. Row and Statement Trigger can be Before and After Type of Triggers?
    (a) True (b) False

11. The developer issues the following statement:

```
Create or replace trigger soccer before delete on soccer_fans
Begin
delete from soccer_fans_snacks where fan_id = :old.fan_id;
end;
```

Why will trigger creation fail?

(a) The row trigger does not properly reference the old value in FAN_ID.
(b) The statement trigger should have been defined as a row trigger.
(c) The statement trigger fires after the delete statement is processed.
(d) The row trigger does not properly define the associated table.

12. To achieve better performance on row triggers, you should
(a) Set the trigger to process after the triggering statement.
(b) Set the trigger to process before the trigging statement.
(c) Set the trigger to process in parallel with the triggering statement.
(d) Set the trigger to update the triggering statement.

13. The developer issues the following statement:

```
Create or replace trigger soccer_fans_snacks_02 before delete on soccer_fans for each row
Begin
DELETE FROM soccer_fans_snacks WHERE fan_id = :prechange.fan_id;
End;
```

Why does the trigger fail on creation?

(a) The statement trigger improperly references the changed row data.
(b) The row trigger does not define prechange as the referencing keyword for old column values.
(c) Row triggers cannot process before the triggering statement.
(d) Statement triggers cannot process before the triggering statement.

14. You define a trigger that contains the clause AFTER UPDATE OR DELETE ON SOCCER_FAN_SNACKS. Which of the following keywords may be useful in your trigger source code to distinguish what should run, and when?

(a) inserting (b) updating (c) deleting (d) truncating

15. The SOCCER_FANS table has a trigger associated with it that inserts data into SOCCER_FANS_SNACKS whenever rows are inserted into SOCCER_FANS. A foreign key constraint exists between FAN_ID on SOCCER_FANS and SOCCER_FANS_SNACKS. What happens when the trigger fires?

(a) The trigger processes normally. (b) The trigger invalidates.
(c) The trigger execution fails because of a mutating or constraining table.
(d) The trigger execution succeeds because the trigger is a statement trigger.

16. The developer issues the following statement:

```
CREATE OR REPLACE TRIGGER soccer_fans
BEFORE DELETE ON SOCCER_FANS FOR EACH ROW
```

BEGIN DELETE FROM soccer_fans WHERE fan_id = :old.fan_id;
END;

Which of the following permissions are required to run this trigger?

(a) execute on SOCCER_FANS
(b) execute on SOCCER_FANS_SNACKS_02
(c) delete on SOCCER_FANS_SNACKS_02
(d) delete on SOCCER_FANS

17. The developer issues the following statement:

CREATE OR REPLACE TRIGGER soccer_fans BEFORE DELETE ON SOCCER_FANS FOR EACH ROW
BEGIN DELETE FROM soccer_fans_snacks WHERE fan_id = :old.fan_id;
END;

Which of the following statements best describes the trigger created?

(a) An update trigger that fires before Oracle processes the triggering statement
(b) An insert trigger that fires after Oracle processes the triggering statement
(c) An insert trigger that fires after Oracle processes the triggering statement
(d) A delete trigger that fires before Oracle processes the triggering statement

18. Table SOCCER_FAN_SEAT contains two columns: FAN and SEAT_NUM. A trigger is created in this table, whose triggering statement definition is AFTER UPDATE OF SEAT_NUM ON SOCCER_FAN_SEAT. You issue an UPDATE statement that changes column FAN only. Which of the following best describes what happens next?

(a) The trigger fires successfully
(b) The trigger fires unsuccessfully
(c) Nothing, the SEAT_NUM column was not updated
(d) The trigger invalidates.

Solution Keys

1. (b)	2. (b)	3. (b)	4. (a)	5. (a)	6. (b)	7. (c)	8. (d)	9. (a)
10. (a)	11. (b)	12. (a)	13. (b)	14. (b&c)	15. (c)	16. (d)	17. (d)	18. (c)

●●●