**Define Triggers. Explain the syntax of creating triggers in PL/SQL.**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

**Benefits of Triggers**

Triggers can be written for the following purposes:

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

**Creating Triggers**

The syntax for creating a trigger is:

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

 Declaration-statements

BEGIN

 Executable-statements

EXCEPTION

 Exception-handling-statements

END;

Where,

* CREATE [OR REPLACE] TRIGGER trigger\_name: Creates or replaces an existing trigger with the trigger\_name.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

**Example:**

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

 dbms\_output.put\_line('Old salary: ' || :OLD.salary);

 dbms\_output.put\_line('New salary: ' || :NEW.salary);

 dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

When the above code is executed at SQL prompt, it produces the following result:

Trigger created.

Here following two points are important and should be noted carefully:

* OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

**Triggering a Trigger**

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );

When a record is created in CUSTOMERS table, above create trigger **display\_salary\_changes** will be fired and it will display the following result:

Old salary:

New salary: 7500

Salary difference:

Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

When a record is updated in CUSTOMERS table, above create trigger**display\_salary\_changes** will be fired and it will display the following result:

Old salary: 7500

New salary: 8000

Salary difference: 500

**Explain the types of triggers.**

A **PL/SQL trigger** is a construct in PL/SQL that runs or "triggered" on event of changes being made to a table in the database. The triggering event is a INSERT, UPDATE or DELETE done on a table. The **trigger** can be made so it can be "fired" either BEFORE or AFTER the Data Manipulation Language is executed.

>A database trigger is a block of code that is automatically executed in response to certain events.

>Triggers are executed implicitly whenever the triggering event happens.

>The triggering event is an INSERT, DELETE, or UPDATE command.

>The timing can be either BEFORE or AFTER, INSTEAD OF trigger.

The **trigger** can be either row-level or statement-level, where the former fires once for each row affected by the triggering statement and the latter fires once for the whole statement.

You can write triggers that fire whenever one of the following operations occurs:

1. DML statements (INSERT, UPDATE, DELETE) on a particular table or view, issued by any user
2. DDL statements (CREATE or ALTER primarily) issued either by a particular schema/user or by any schema/user in the database
3. Database events, such as logon/logoff, errors, or startup/shutdown, also issued either by a particular schema/user or by any schema/user in the database

**A** **trigger** **has three basic parts**:

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

**Different types of triggers can be:**

**Row Triggers and Statement Triggers:** A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects. A row trigger fires once for each row affected by the triggering event.

**BEFORE and AFTER Triggers:** BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

**INSTEAD OF Triggers**: INSTEAD OF triggers describe how to perform insert, update, and delete operations against views that are too complex to support these operations natively. INSTEAD OF triggers allow applications to use a view as the sole interface for all SQL operations (insert, delete, update and select).

**Triggers on System Events and User Events:**You can use triggers to publish information about database events to subscribers. System events are for example Database startup and shutdown, Data Guard role transitions etc and User Events are User logon and logoff, DDL statements (CREATE, ALTER, and DROP) etc.

Let's create a table 'product\_check' which we can use to store messages when triggers are fired.

CREATE TABLE product

(Message varchar2(50),

Current\_Datenumber(32)

);

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

**1) BEFORE UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' before a sql update statement is executed, at the statement level.

CREATE or REPLACE TRIGGER Before\_Update\_Stat\_product

BEFORE

UPDATE ON product

Begin

INSERT INTO product\_check

Values('Before update, statement level',sysdate);

END;

/

**2) BEFORE UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' before each row is updated.

 CREATE or REPLACE TRIGGER Before\_Upddate\_Row\_product

 BEFORE

 UPDATE ON product

 FOR EACH ROW

 BEGIN

 INSERT INTO product\_check

Values('Before update row level',sysdate);

 END;

 /

**3) AFTER UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' after a sql update statement is executed, at the statement level.

 CREATE or REPLACE TRIGGER After\_Update\_Stat\_product

 AFTER

 UPDATE ON product

 BEGIN

 INSERT INTO product\_check

Values('After update, statement level', sysdate);

 End;

 /

**4) AFTER UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' after each row is updated.

 CREATE or REPLACE TRIGGER After\_Update\_Row\_product

 AFTER

insert On product

 FOR EACH ROW

 BEGIN

 INSERT INTO product\_check

Values('After update, Row level',sysdate);

 END;

 /

Now let’s execute a update statement on table product.

 UPDATE PRODUCT SET unit\_price = 800

 WHERE product\_id in (100,101);

Lets check the data in 'product\_check' table to see the order in which the trigger is fired.

 SELECT \* FROM product\_check;

**Explain Triggers on view.**

* Views are commonly used to separate the logical database schema from the physical schema. Unfortunately the desired transparency often falls short in the case of UPDATE, DELETE or INSERT operations, since all but the simplest views are not updatable.To be precise we need to distinguish between three levels of updatability:
1. **Deletable:** In order for DB2 to delete a row from a view it must be able to map the row specified in the view to one, and only one, row in a base table.
2. **Updatable:** In order to update a column in a view, DB2 must not only be able to map the row specified in the view to one row in a base table, it also must be able to map the column to be updated to a column in a base table. Hence all views that are updatable must by definition also be deletable.
3. **Insertable:** In order for a row to be inserted into a view, DB2 must be able to map the new row to a table and to map all the columns specified to columns in that table. Thus all views that are insertable are by definition updatable and hence deletable.
* While database programming sometimes there may besituation that the triggers has two options and it must fired withalternate options. The INSTEAD OF trigger satisfy the condition.Triggers on views are known as INSTEAD OF triggers. They areknown by their name because they skip the current triggering eventaction and perform alternate one. Other reason could be that onlytiming mode available in such triggers is INSTEAD OF. It isrequired for the complex view because it is based on more thanone table. Any DML on complex view would be successful only if allkey columns, not null columns are selected in the view definition.Alternatively, INSTEAD OF trigger can be created to synchronizethe effect of DML across all the tables.Instead of trigger is a row level trigger and can be used onlywith a view, and not with tables.

**LISTING OF TRIGGERS INFORMATION:**We can see all our user defined triggers by doing a selectstatement on USER\_TRIGGERS. This will gives us clear ideaabout the trigger and its structure.For example:
**SELECT TRIGGER\_NAME FROM USER\_TRIGGERS;**

**ALTERING A TRIGGER:**
ALTER TRIGGER trigger\_name [ENABLE|DISABLE];

**Syntax for removing Triggers:**For removing the trigger we have to use following syntax.
DROP TRIGGER trigger\_name;

**KNOWING INFORMATION ABOUT TRIGGERS:**We can use the data dictionary view 'USER\_TRIGGERS' to obtaininformation about any trigger. The below statement shows thestructure of the view 'USER\_TRIGGERS’.
**DESC USER\_TRIGGERS;**

**Write short note on Dynamic SQL.**

* In programming world the word Dynamic refers for runtime execution. The Dynamic SQL is used to write programs that mention SQL statements whose full text is not known until runtime. The complete query or the procedure is evaluated only at the run time, which gives results depending on the code.

The full code of static SQL statements is known at compilation, which
gives the following benefits:
* The total compilation checks that the necessary privileges are already given to access the database objects.
* The total compilation verifies that the SQL statements reference valid database objects.

## Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

## When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at precompile time:

* Text of the SQL statement (commands, clauses, and so on)
* The number of host variables
* The datatypes of host variables
* References to database objects such as columns, indexes, sequences, tables, usernames, and views

## How Dynamic SQL Statements are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Oracle then parses the SQL statement to ensure it meets syntax rules.

Next, Oracle binds the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.

Then Oracle executes the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

**How to execute PL/SQL Block Dynamically?**

* We can use dynamic SQL to create applications that execute
dynamic queries, whose full code is not known until runtime. Many
types of programs need to use dynamic queries, including:
* The Code or Programs that allow users to input or choose query search or sorting criteria at runtime.
* The Code or Programs that allow users to input or choose optimizer hints at run time.
* The Code or Programs that query a database where the datadefinitions of tables are constantly changing.
* The Code or Programs that query a database where newtables are often created.

# Native Dynamic SQL

Dynamic SQL allows an application to run SQL statements whose contents are not known until runtime. The statement is built up as a string by the application and is then passed to the server, in a similar way to the ADO interface in VB.

Generally dynamic SQL is slower than static SQL so it should not be used unless absolutely necessary. Also, since syntax checking and object validation cannot be done until runtime, code containing large amounts of dynamic SQL may be littered with mistakes but still compile.

The main advantage of dynamic SQL is that it allows you to perform DDL commands that are not supported directly within PL/SQL, such as creating tables. It also allows you to access objects that will not exist until runtime.

## DDL Operations

Commands such as DDL operations that are not directly supported by PL/SQL can be performed using dynamic SQL.

BEGIN

 EXECUTE IMMEDIATE 'TRUNCATE TABLE my\_table;';

END;

/

**Native Dynamic SQL vs DBMS\_SQL**

|  |  |
| --- | --- |
|  |  |
| **Native Dynamic SQL** | **DBMS\_SQL** |
| Easy to use and concise. | Often long-winded and awkward. |
| PL/SQL interpreter has built in support for Native Dynamic SQL so it is more efficient than DBMS\_SQL. | DBMS\_SQL uses a Procedural API so it is generally slower than Native Dynamic SQL. |
| Supports user defined types. | Does not support user defined types. |
| Supports FETCH INTO record types | Does not support FETCH INTO record types |
| Not supported in client site code. | Supported in client side code. |
| Does not support DESCRIBE\_COLUMNS | Supports DESCRIBE\_COLUMNS |
| Does not support bulk Dynamic SQL, but it can be faked by placing all statements in a PL/SQL block. | Supports bulk Dynamic SQL. |
| Only supports Single row Updates/Deletes with RETURNING clause. | Supports Single and Multiple row Updates/Deletes with RETURNING clause. |
| Does not support SQL statements bigger than 32K | Does support SQL statements bigger than 32K |
| Parse required for every execution | Parse once, execute many possible |

**State the Advantages of DBMS\_SQL Package**

* This is the standard SQL package provided by the oracle toDatabase programmer to bring flexibility in the coding. TheDBMS\_SQL package gives access to dynamic SQL and dynamicPL/SQL from within PL/SQL programs. "Dynamic" means that theSQL statements we execute with this package are not prewritteninto our programs. They are, constructed at runtime as characterstrings and then passed to the SQL engine for execution.
* The DBMS\_SQL package provides an entity called a SQLcursor number. Because the SQL cursor number is a PL/SQLinteger, we can pass it across call boundaries and store it. We canalso use the SQL cursor number to obtain information about theSQL statement that we are executing.
* **Very easy to Use:**Because native dynamic SQL is integrated with SQL, we canuse it in the same way that we use static SQL within PL/SQL code. Native dynamic SQL code is typically more compact and readable than equivalent code that uses the DBMS\_SQL package. With the DBMS\_SQL package we must call many procedures and functions in a strict sequence, making even simple operations require a lot of code. We can avoid this complexity by using native dynamic SQL instead.
* The Native dynamic SQL in PL/SQL performs comparably to the performance of static SQL, because the PL/SQL interpreter has built-in support for it. Programs that use native dynamic SQL are much faster than programs that use the DBMS\_SQL package. Typically, native dynamic SQL statements perform 1.5 to 3 times better than equivalent DBMS\_SQL calls.
* We can Reuse SQL Statements using DBMS\_SQL
* Client-Side Program supports DBMS\_SQL Package
* Supports Multiple Row Updates and Deletes with a RETURNING Clause
* It Supports DESCRIBE\_COLUMNS procedure in the DBMS\_SQL
package can be used to describe the columns for a cursor opened and parsed through DBMS\_SQL.
* Supports SQL Statements Larger than 32KB
* The dynamic SQL is also supported in various database languages with their language specifications.