# Unit 10

## PL/SQL Concept

# Index

- View
- Stored Procedures
- Database Triggers
- Cursors

# Features of PL/SQL

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

# Advantages of PL/SQL

- **Block structure**:
  - PL/SQL consist of block of code, which can be nested within each other.
  - Each block forms a unit of a task or a logical module.
  - PL/SQL blocks can be stored in the database and reused.
- **Procedural language capability**:
  - PL/SQL consist of procedural constructs such as conditional statements (if, if else, nested if, else if ladder) and loops (for, while, do while).
- **Better performance:**
  - PL/SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- **Error handling**:
  - PL/SQL handles errors or exceptions effectively during the execution of PL/SQL program.
  - Once an exception is caught, specific action can be taken depending upon the type of the exception or it can be displayed to the user with message.

# View

- In SQL, a VIEW is a **virtual relation based on the result-set of a SELECT statement**.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- In some cases, we can modify a view and present the data as if the data were coming from a single table.
- Syntax:-

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

# Example of view

- Consider the CUSTOMERS table having the following records:

| ID | NAME | AGE | CITY | SALARY |
|---|---|---|---|---|
| 1 | Ramesh | 32 | Ahmedabad | 20000 |
| 2 | Karan | 35 | Rajkot | 15000 |
| 3 | Mayur | 30 | Surat | 22000 |
| 4 | Dipak | 35 | Rajkot | 20000 |
| 5 | Nilesh | 38 | Surat | 29000 |
| 6 | Kalpesh | 37 | Ahmedabad | 26000 |

- We create a view that contain customer name and age from CUSTOMERS table:

# Example of view

- Syntax:-

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

| NAME | AGE |
|---|---|
| Ramesh | 32 |
| Karan | 35 |
| Mayur | 30 |
| Dipak | 35 |
| Nilesh | 38 |
| Kalpesh | 37 |

- Now, you can query CUSTOMERS_VIEW in similar way as you query an actual table. Following is the example:

```
SELECT * FROM CUSTOMERS_VIEW;
```

# Stored procedure

- A stored procedure (proc) is a **group of PL/SQL statements that performs specific task**.
- A procedure has two parts, **header and body**.
- The **header consists** of the **name of the procedure** and the **parameters** passed to the procedure.
- The **body consists** of **declaration section, execution section** and **exception section**.
- A procedure **may or may not return any value**. A procedure may return more than one value.

# Stored procedure (Syntax)

- CREATE [OR REPLACE] PROCEDURE procedure_name

  [list of parameters]

  AS

  sql_statement

# Stored procedure

- **Create**:-It will create a procedure.
- **Replace** :- It will re-create a procedure if it already exists.
- We can pass parameters to the procedures in three ways.
  1. **IN-parameters**: - These types of parameters are used to send values to stored procedures.
  2. **OUT-parameters**: - These types of parameters are used to get values from stored procedures. This is similar to a return type in functions but procedure can return values for more than one parameters.
  3. **IN OUT-parameters**: - This type of parameter allows us to pass values into a procedure and get output values from the procedure.

# Stored procedure

- AS indicates the beginning of the body of the procedure.
- sql_statement contains the SQL query. (select, insert, update or delete)
- The syntax within the brackets [ ] indicates that they are optional.
- By using CREATE OR REPLACE together the procedure is created if it does not exist and if it exists then it is replaced with the current code.

# How to execute a stored procedure?

- There are two ways to execute a procedure.

1. From the SQL prompt.

   **Syntax**: EXECUTE [or EXEC] procedure_name (parameter);

2. Within another procedure: simply use the procedure name.

   **Syntax**: procedure_name (parameter);

# Advantages of stored procedure

- **Security**:- We can **improve security** by giving rights to selected persons only.
- **Faster Execution**:- It is **precompiled** so **compilation** of procedure is **not required every time** you call it.
- **Sharing of code**:- Once procedure is created and stored, it can be **used by more than one user**.
- **Productivity**:- Code written in procedure is **shared by all programmers**. This eliminates redundant coding by multiple programmers so overall improvement in productivity.

# Example of stored procedure

- CREATE [OR ALTER] PROCEDURE get_studentname_by_id

  @id int

  AS

  SELECT studentname FROM student WHERE studentID = @id;

- **Execute**:- EXEC get_studentname_by_id 10

- **Explanation**:- Above procedure gives the name of student whose id is 10.

# Database triggers

- A trigger is a **PL/SQL block** structure which is **triggered (executed) automatically when DML statements** like Insert, Delete, and Update is **executed on a table**.
- In SQL Server we can create the following 3 types of triggers:
  1. Data Definition Language (DDL) triggers
  2. Data Manipulation Language (DML) triggers
  3. Logon triggers

# Data Definition Language (DDL) triggers

- In SQL Server we can **create triggers on DDL statements (like CREATE, ALTER and DROP)** and **certain system-defined Stored Procedures that does DDL-like operations**.

# Data Manipulation Language (DML) triggers

- In SQL Server we can **create triggers on DML statements (like INSERT, UPDATE and DELETE)** and **Stored Procedures that do DML-like operations**. DML Triggers are of two types.

1. After trigger (using FOR/AFTER CLAUSE)
2. Instead of trigger (using INSTEAD OF CLAUSE)

# Data Manipulation Language (DML) triggers

- DML Triggers are of two types.
  1. After trigger (using FOR/AFTER CLAUSE) : After triggers are **executed after completing the execution of DML statements**.
     - Example: If you insert a record/row into a table then the trigger related/associated with the insert event on this table will executed only after inserting the record into that table.
     - If the record/row insertion fails, SQL Server will not execute the after trigger.

# Database triggers

- When triggers can be used,
  - Based on change in one table, we want to update other table.
  - Automatically update derived columns whose values change based on other columns.
  - Logging.
  - Enforce business rules.

# Triggers (syntax)

- CREATE [OR ALTER] TRIGGER trigger_name

  ON table_name

  { FOR | AFTER | INSTEAD OF }

  { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

  AS

  BEGIN

  Executable statements

  END;

# Triggers

- CREATE [OR ALTER ] TRIGGER trigger_name:-
  - This clause creates a trigger with the given name or overwrites an existing trigger.
- ON table_name:-
  - This clause identifies the name of the table or view to which the trigger is related.
- { FOR | AFTER | INSTEAD OF }:-
  - This clause indicates at what time the trigger should be fired. Before executing DML statements or after executing DML statements.

# Triggers

- { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] } :-
  - This clause determines on which kind of statement the trigger should be fired.
  - Either on insert or update or delete or combination of any or all.
  - More than one statement can be used together separated by comma. The trigger gets fired at all the specified triggering event.

# Example of triggers

- Trigger to display a message when we perform insert operation on student table.

```
CREATE TRIGGER student_msg
on Student
AFTER INSERT
AS
BEGIN
        print ‘Record inserted successfully'
END
```

# Example of triggers

- Trigger to display a message when we perform insert, update or delete operation on student table.

```
CREATE TRIGGER student_msg
on Student
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
        print ‘One record is affected'
END
```

# Example of triggers

- **OUTPUT**:- Trigger is created.
- Now when you perform insert, update or delete operation on student table.
- SQL:> Insert into student values (102, 'Raj', 'CE'); OR

  Update student set Dept='EC' where Rno=101 OR

  Delete from student where Rno=101
- It displays following message after executing insert, update or delete statement.
- **Output**:- One record is affected
- We get message that "One record is affected" it indicates that trigger has executed after the insertion operation.

# Cursor

- Cursors are database **objects used to traverse the results of a select SQL query**.
- It is a **temporary work area created in the system memory** when a select SQL statement is executed.
- This temporary work area is **used to store the data retrieved** from the database, and manipulate this data.
- It **points to a certain location** within a record set and allow the operator to move forward (and sometimes backward, depending upon the cursor type).
- We can **process only one record at a time**.
- The set of rows the cursor holds which is called the active set (active data set).
- Cursors are often criticized for their high overhead.

# Types of cursor

- There are two types of cursors in PL/SQL:

1. **Implicit cursors:**
   - These are **created by default by SQL itself** when DML statements like, insert, update, and delete statements are executed.
   - They are also **created when a SELECT statement returns just one row**.
   - We **cannot use implicit cursors for user defined work**.
2. **Explicit cursors:**
   - Explicit cursors are **user defined cursors written by the developer**.
   - They can be **created when a SELECT statement returns more than one row**.
   - Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row.
   - When you fetch a row, the current row position moves to next row.

# Steps to manage explicit cursor

1. **Declare Cursor**: A **cursor is declared by defining the SQL statement that returns a result set**.
2. **Open**: A **Cursor is opened and populated by executing the SQL statement defined by the cursor**.
3. **Fetch**: When the cursor is opened, **rows can be fetched from the cursor one by one or in a block to perform data manipulation**.
4. **Close**: **After data manipulation, close the cursor explicitly**.
5. **Deallocate**: Finally, **delete the cursor definition** and **release all the system resources associated with the cursor**.

# Explicit cursor

- An explicit cursor is defined in the declaration section of the PL/SQL Block.
- It is created on a SELECT Statement which returns more than one row. A suitable name for the cursor.
- General syntax for creating a cursor:

  CURSOR cursor_name IS select_statement;

  - cursor_name – A suitable name for the cursor.
  - select_statement – A select query which returns multiple rows

# How to use explicit cursor?

- There are four steps in using an Explicit Cursor.
  1. **DECLARE the cursor** in the Declaration section.
  2. **OPEN the cursor** in the Execution Section.
  3. **FETCH the data from the cursor** into PL/SQL variables or records in the Execution Section.
  4. **CLOSE the cursor** in the Execution Section before you end the PL/SQL Block.

# Syntax of explicit cursor

```
DECLARE variables;
        records;
        create a cursor;
        BEGIN
OPEN cursor;
FETCH cursor;
        process the records;
 CLOSE cursor;
        END;
```

# Example of cursor

- Cursor to insert record from student table to student1 table if branch is CE.

```
DECLARE @rno int, @name varchar(50),@branch varchar(50);
DECLARE cursor_student CURSOR
        FOR SELECT rno, name, branch FROM student;
OPEN cursor_student;
        FETCH NEXT FROM cursor_student INTO @rno, @name, @branch;
        WHILE @@FETCH_STATUS = 0
BEGIN
        IF (@branch='CE')
                INSERT INTO student1 values (@rno, @name, @branch)
        FETCH NEXT FROM cursor_student INTO @rno, @name, @branch;
END;
CLOSE cursor_student;
DEALLOCATE cursor_student;
```