REVA UNIVERSITY

Bengaluru, India

School of Computer Science and Applications(I MCA-D)

RELATIONAL DATABASE MANAGEMENT SYSTEM

Course Code: M23DE0103

Prof Nagaraj C

TOPICS DISCUSSED IN UNIT - IV

- Functions :aggregate functions, Built-in functions –numeric, date, string functions, set operations,
- sub-queries, correlated sub-queries, Use of group by, having, order by
- Join and its types, Exist, Any, All ,
- Views and its types.
- Transaction control commands – Commit, Rollback, Savepoint.

- PL/SQL: A Programming Language:
- History, Fundamentals, Block Structure,Comments
- Data Types, Other Data Types,
- Declaration, Assignment operation,
- Bind variables, Substitution Variables,
- Printing
- Arithmetic Operators.

IN THIS UNIT, WE WILL DISCUSS

- PL/SQL Control Structures and Embedded SQL :
- Control Structures
- Nested Blocks
- SQL in PL/SQL
- Data Manipulation
- Transaction Control statements.

- PL/SQL Cursors and Exceptions: Cursors
- Implicit & Explicit Cursors and Attributes
- Cursor FOR loops
- Cursor with Parameters
- Cursor Variables
- Exceptions, Types of Exceptions.
- Named Blocks: Procedures,
- Functions Packages, Triggers, Data Dictionary Views.

1. AGGREGATE FUNCTIONS

Aggregate functions perform calculations on multiple rows and return a single value.

Function	Description	Example
SUM()	Returns the total sum of a numeric column.	SELECT SUM(salary) FROM employees;
AVG()	Returns the average value.	SELECT AVG(salary) FROM employees;
COUNT()	Returns the number of rows.	SELECT COUNT(*) FROM employees;
MAX()	Returns the highest value.	SELECT MAX(salary) FROM employees;
MIN()	Returns the lowest value.	SELECT MIN(salary) FROM employees;

EXAMPLE

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;
```

This query calculates the average salary for each department.

2. BUILT-IN FUNCTIONS

SQL provides built-in functions to perform operations on numbers, dates, and strings.

a) Numeric Functions

These functions perform mathematical operations.

Function	Description	Example
ABS(x)	Returns absolute value.	SELECT ABS(-10); → 10
ROUND(x, d)	Rounds a number to d decimal places.	SELECT ROUND(45.678, 2); → 45.68
CEIL(x)	Rounds a number up to the nearest integer.	SELECT CEIL(4.3); -- Output: 5
FLOOR(x)	Rounds a number down to the nearest integer.	SELECT FLOOR(4.7); -- Output: 4
MOD(a, b)	Returns remainder of division.	SELECT MOD(10, 3); → 1

B) DATE FUNCTIONS

These functions operate on date and time values.

Function	Description	Example
NOW()	Returns current date and time.	SELECT NOW();
CURDATE()	Returns current date.	SELECT CURDATE();
CURTIME()	Returns current time.	SELECT CURTIME();
YEAR(date)	Extracts year from date.	SELECT YEAR('2024-03-02'); → 2024
MONTH(date)	Extracts month from date.	SELECT MONTH('2024-03-02'); → 3
DAY(date)	Extracts day from date.	SELECT DAY('2024-03-02'); → 2
DATEDIFF(d1, d2)	Returns difference in days between two dates.	SELECT DATEDIFF('2024-03-10', '2024-03-02'); → 8

C) STRING FUNCTIONS

These functions perform operations on string values.

Function	Description	Example
LENGTH(str)	Returns length of a string.	SELECT LENGTH('Hello'); → 5
UPPER(str)	Converts string to uppercase.	SELECT UPPER('hello'); → 'HELLO'
LOWER(str)	Converts string to lowercase.	SELECT LOWER('HELLO'); → 'hello'
CONCAT(s1, s2, …)	Concatenates multiple strings.	SELECT CONCAT('Hello', ' World'); → 'Hello World'
SUBSTRING(str, start, length)	Extracts a substring.	SELECT SUBSTRING('Database', 1, 4); → 'Data'
REPLACE(str, old, new)	Replaces occurrences of old substring with new one.	SELECT REPLACE('Hello World', 'World', 'SQL'); → 'Hello SQL'

3. SET OPERATIONS

Set operations combine results of two or more queries.

Operator	Description
UNION	Combines results of two queries, removes duplicates.
UNION ALL	Combines results, keeps duplicates.
INTERSECT	Returns common records between queries.
EXCEPT (MINUS in some databases)	Returns records from first query not in second.

VIEWS IN DBMS

Views are virtual tables that provide a way to present data from one or more tables in a customized manner. Unlike physical tables, views do not store data on the disk; instead, they store a query that retrieves data from the underlying tables

Types of Views in DBMS

1. Simple View

2. **Complex View**

3. Materialized View

4.Updatable View

5.Read only view

CREATING VIEWS

Views can be created using the `CREATE VIEW` statement. A view can be created from a single table or multiple tables.

syntax for creating a view:

```
CREATE VIEW view_name AS
SELECT column1, column2,
FROM table_name
WHERE condition;
```

CREATING VIEWS

Example 1: Creating a View from a Single Table

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

Example 2: Creating a View from Multiple Tables

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

1 SIMPLE VIEW

- Based on a single table.
- Does not contain aggregate functions or joins.
- Can be used for basic operations like SELECT, INSERT, UPDATE, DELETE.

Example:

```
CREATE VIEW employee_view AS
SELECT id, name, salary FROM employees;
```

Usage:

```
SELECT * FROM employee_view;
```

Updating Data:

```
UPDATE employee_view SET salary = 60000 WHERE id = 101;
```

2 COMPLEX VIEW

- **Based on multiple tables using joins.**
- **Can contain aggregate functions.**
- **Does not always allow INSERT, UPDATE, or DELETE.**

Example

```
CREATE VIEW employee_department AS

SELECT e.name, e.salary, d.department_name

FROM employees e

JOIN departments d ON e.department_id = d.department_id;
```

Usage:

```
SELECT * FROM employee_department;
```

3. MATERIALIZED VIEW

A **materialized view** stores the query result **physically** in the database. It is used to improve performance for complex queries by precomputing or stored results.

Needs manual or automatic refresh to update the stored data.

- ✓ **Faster query execution**
- ✓ **Needs manual refresh to update data**

Example: Materialized View

```
CREATE MATERIALIZED VIEW high_salary_employees AS

SELECT id, name, salary FROM employees WHERE salary > 50000;
```

Real-Life Example: E-Commerce Sales Dashboard

- An e-commerce company needs to generate daily sales reports, which involve:
  - Summing total sales from millions of transactions.
  - Grouping data by product, category, and region.
  - Applying filters based on time (e.g., daily, weekly, monthly reports).

Without a Materialized View:
Every time a report is generated, SQL must process a huge amount of transactional data.

This query can be slow if it runs on millions of records every time.

With a Materialized View:
The results are stored and refreshed periodically, improving speed.

4. UPDATABLE VIEW

An **updatable view** allows INSERT, UPDATE, and DELETE operations if:

- It is based on a single table.
- It does not use aggregations or joins.

✓ **Allows DML operations under certain conditions.**

Example: Updatable View

```
CREATE VIEW editable_employee AS
SELECT id, name, salary FROM employees WHERE department_id = 1;
```

Updating Data in the View:

```
UPDATE editable_employee SET salary = 70000 WHERE id = 102;
```

5. READ-ONLY VIEW

1. A **read-only view** prevents modifications (INSERT, UPDATE, DELETE) to maintain **data integrity** and security.
2. ❌ **Does not allow DML operations.**

Example: Read-Only View:

```
CREATE VIEW readonly_employee AS
SELECT id, name, salary FROM employees
WITH READ ONLY;
```

Trying to Update Data:

```
UPDATE readonly_employee SET salary = 75000 WHERE id = 103;
-- Error: Cannot modify a read-only view
```

INTRODUCTION TO PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is an extension of SQL developed by Oracle Corporation. It allows users to write procedural code, including loops, conditional statements, and exception handling, making SQL more powerful.

Key Features of PL/SQL:

1. **Block Structure** - PL/SQL is divided into blocks (Anonymous, Procedure, Function, etc.).
2. **Variables & Constants** - Supports variables and constants for better control.
3. **Control Structures** - Supports loops (FOR, WHILE, LOOP), conditional statements (IF-ELSE).
4. **Exception Handling** - Manages runtime errors using EXCEPTION blocks.
5. **Cursors** - Handles query result sets efficiently.
6. **Triggers, Procedures & Functions** - Supports reusable stored programs.

PL/SQL BLOCK STRUCTURE

A PL/SQL block has three main sections:

1. **Declaration Section** (Optional) - Declares variables, constants, and cursors.
2. **Execution Section** (Mandatory) - Contains SQL and procedural statements.
3. **Exception Handling Section** (Optional) - Handles errors during execution.

Basic PL/SQL Program

```
DECLARE
    v_message VARCHAR2(50);
BEGIN
    v_message := 'Hello, PL/SQL!’;
    DBMS_OUTPUT.PUT_LINE(v_message);
END;
/
```

Explanation: Declares a variable v_message.

Assigns a string value 'Hello, PL/SQL!’.

Uses DBMS_OUTPUT.PUT_LINE to print the message.

CONNECTING TO THE SERVER TO RUN PL/SQL PROGRAMS

```
SQL> connect
Enter user-name: system
Enter password:
Connected.
SQL> set serveroutput on
```

PL/SQL PROGRAM WITH VARIABLES

```
DECLARE
    v_name VARCHAR2(20) := ‘John’;
    v_salary NUMBER := 50000;
BEGIN
   DBMS_OUTPUT.PUT_LINE(“Employee Name: ’|| v_name);
   DBMS_OUTPUT.PUT_LINE( ‘'Salary: ’ || v_salary);
END;
/
```

DECLARATION

Variables must be declared before use:

Example:

```
DECLARE
   v_salary NUMBER(10,2);
   v_employee_name VARCHAR2(50);
BEGIN
     v_salary := 50000.75;
     v_employee_name := 'Alice’;
    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee_name || ' Salary: ' || v_salary);
END;
/
```

OBTAIN INPUT FROM THE USER

```
SET SERVEROUTPUT ON;


DECLARE
    v_name VARCHAR2(50);
BEGIN
    v_name := '&enter_name';  -- Prompt for input
    DBMS_OUTPUT.PUT_LINE('Hello, ' || v_name || '!');
END;
/
```

Using ACCEPT to Get Input in SQL*Plus

```
SET SERVEROUTPUT ON;
ACCEPT v_age CHAR PROMPT 'Enter your age: ';


DECLARE
    v_age NUMBER;
BEGIN
    v_age := &v_age;  -- Assign input value
    DBMS_OUTPUT.PUT_LINE('Your age is ' || v_age);
END;
/
```

ASSIGNMENT OPERATION

Assignment is done using :=.

Example:

```
DECLARE
   v_counter NUMBER := 10;
BEGIN
   v_counter := v_counter + 5;
   DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
END;
/
```

PL/SQL PROGRAM TO FIND SUM OF 2 NUMBERS

```
SQL> set serveroutput on;
Declare
  x number(5);
  y number(5);
  z number(7);
Begin
 -- Here we Assigning 10 into x
  x:=10;
 -- Assigning 20 into x
 y:=20;
 -- Assigning sum of x and y into z
 z:=x+y;
 -- Print the Result
  dbms_output.put_line('Sum is '||z);
end;
/
```

PL/SQL %TYPE ATTRIBUTE

Oracle PL/SQL provides a special data type called the %TYPE data type.

The %TYPE data type allows you to declare a variable that is associated with a column in a database table.

The %TYPE attribute is a powerful feature designed to enhance the readability, maintainability, and flexibility of code by associating variables with database columns or other variables. .

The Oracle PL/SQL %TYPE attribute allow you to declare a constant, variable, or parameter to be of the same data type as previously declared variable, record, nested table, or database column.

To use the PL/SQL %TYPE data type, you first need to declare a variable. You can then use the variable in your PL/SQL code just like any other variable.

The %TYPE attribute can be used with variables, records, nested tables, and database columns.

Syntax:

```
identifier Table.column_name%TYPE;
```

EXAMPLE

Here is an example of how to declare a variable using the PL/SQL %TYPE attribute:

```
DECLARE
    v_name employee.lastname%TYPE;
    v_dep  number;
    v_min_dep v_dep%TYPE:=31;
BEGIN
    select lastname
    into v_name
    from EMPLOYEE
    where DEPARTMENTID=v_min_dep;
    DBMS_OUTPUT.PUT_LINE('v_name: '||v_name);
END;
/
```

CONTROL STRUCTURES IN PL/SQL – DEFINITION

Control structures in **PL/SQL** are constructs that allow the execution flow of a program to be directed based on conditions and iterations. These structures enable decision-making, looping, and branching, making the code more flexible and dynamic.

IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
   sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
   compute_bonus(empid);
   UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

You might want to place brief IF statements on a single line, as in `IF x > y THEN high := x; END IF;`

IF-THEN-ELSE STATEMENT

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
    sequence_of_statements1
ELSE
   sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed.

EXAMPLE:

In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
   UPDATE accounts SET balance = balance + credit WHERE acc_no=101
ELSE
   UPDATE accounts SET balance = balance - debit WHERE acc_no=101
END IF;
```

IF-THEN-ELSIF STATEMENT

Sometimes you want to select an action from several mutually exclusive alternatives.

The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions,

```
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition.

An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom.

If any condition is true, its associated sequence of statements is executed and control passes to the next statement.

If all conditions are false or null, the sequence in the ELSE clause is executed.

EXAMPLE

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

If the value of sales is larger than 50000, the first and second conditions are true. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the INSERT statement.

USING IF-ELSE IN PL/SQL

```
DECLARE
    v_marks NUMBER := 85;
BEGIN
   IF v_marks >= 90 THEN
        DBMS_OUTPUT.PUT_LINE('Grade: A');
   ELSIF v_marks >= 75 THEN
        DBMS_OUTPUT.PUT_LINE('Grade: B’);
  ELSE
       DBMS_OUTPUT.PUT_LINE('Grade: C’);
  END IF;
END;
/
```

CASE STATEMENT

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. To compare the IF and CASE statements, consider the following code that outputs descriptions of school grades:

```
IF grade = 'A' THEN
  dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
  dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
  dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
  dbms_output. put_line('Fair');
ELSIF grade = 'F' THEN
  dbms_output.put_line('Poor');
ELSE
  dbms_output.put_line('No such grade');
END IF;
```

CASE EXAMPLE:

Notice the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. Let us rewrite the preceding code using the CASE statement, as follows:

```
CASE grade
  WHEN 'A' THEN dbms_output.put_line('Excellent');
  WHEN 'B' THEN dbms_output.put_line('Very Good');
  WHEN 'C' THEN dbms_output.put_line('Good');
  WHEN 'D' THEN dbms_output.put_line('Fair');
  WHEN 'F' THEN dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('No such grade');
END CASE;
```

CASE FEATURES

1. The CASE statement is a powerful and efficient control structure in PL/SQL, providing a clean and readable way to handle multiple conditions.
2. It enhances both procedural PL/SQL code and SQL queries, making it a versatile choice for decision-making.
3. The CASE statement begins with the keyword CASE. The keyword is followed by a selector or casename

USING LOOPS IN PL/SQL

1. FOR LOOP

```
BEGIN
 FOR i IN 1..5 LOOP
 DBMS_OUTPUT.PUT_LINE('Number: ' || i);
 END LOOP;
END;
/
```

2. WHILE LOOP

```
DECLARE
 v_count NUMBER := 1;
BEGIN
 WHILE v_count <= 5 LOOP
DBMS_OUTPUT.PUT_LINE('Count: ' || v_count);
 v_count := v_count + 1;
END LOOP;
END;
/
```

ARITHMETIC OPERATORS

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- MOD (Modulus)

ARITHMETIC OPERATORS-EXAMPLE

Example:

```
DECLARE
  a NUMBER := 10;
  b NUMBER := 3;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Addition: ' || (a + b));
  DBMS_OUTPUT.PUT_LINE('Subtraction: ' || (a - b));
  DBMS_OUTPUT.PUT_LINE('Multiplication: ' || (a * b));
  DBMS_OUTPUT.PUT_LINE('Division: ' || (a / b));
  DBMS_OUTPUT.PUT_LINE('Modulus: ' || MOD(a, b));
END;
/
```

A PROGRAM TO CHECK IF A GIVEN NUMBER IS **EVEN OR ODD** USING **IF-ELSE**.

```
DECLARE
    v_num NUMBER := 7;  -- Change this number to test different values
BEGIN
    -- Check if the number is even or odd
    IF MOD(v_num, 2) = 0 THEN
        DBMS_OUTPUT.PUT_LINE(v_num || ' is Even');
    ELSE
        DBMS_OUTPUT.PUT_LINE(v_num || ' is Odd');
    END IF;
END;
/
```

PL/SQL TRANSACTIONS

A transaction is a sequence of operations performed by executing the SQL statements in PL/SQL block of code, where the following rules are applicable:

1. If we have a set of statements in a transaction, then the complete set executes as a block, where if a single statement fails, the affect of all the previous successful statement executions is also reverted back.
2. In a transaction, either all the statements get executed successfully or none.
3. The scope of a transaction is defined by using COMMIT and ROLLBACK commands

USING PL/SQL TRANSACTIONS

It has a beginning and an end.

A transaction begins whenever the first SQL statement (particularily DML commands INSERT, UPDATE, DELETE, SELECT) is encountered and ends when a COMMIT or ROLLBACK command is executed.

USING COMMIT

Commit command is executed after every DML command as they are not auto saved or commited like DDL commands.

This way, the commit command permanently changes the data in the database.

Following is the **syntax:**

```
Commit;
```

NOTE: By default, **automatic commit for DML commands is off**. The automatic commit for DML commands can be set by using the following command:

```
set autocommit on;
-- and to turn it off
set autocommit off;
```

USING SAVEPOINT

For longer transactions, savepoint is quite useful as it divides longer transactions into smaller parts and marks certain points of a transaction as checkpoints.

It is useful when we want to rollback a particular part of a transaction instead of applying rollback to unwanted parts of a transaction or the complete transaction.

For example,

if a complete transaction has 8 DML statements, and we create a savepoint after 4 statements, then if, for some reason after the execution of 6th statement we want to rollback uptil the 4th statement, then we can easily do that and the transaction can again be executed starting from the 4th statement.

Following is the **syntax:**

```
Savepoint  <savepointname>;
```

USING ROLLBACK

Rollback means **undo**. Whenever **rollback** command is executed, it ends the transaction and undoes all the changes made during the transaction. Rollback can be applied to those transactions which are not committed.

The rollback command will have no affect if it is executed after the commit command because in that case the commit command will make the changes done in the transaction permanent.

Following is the **syntax:**

Rollback [to savepoint <savepointname >];

where,

savepoint is an optional parameter and is used to rollback a transaction partly upto a certain specified point.

savepointname is the name given to the savepoint created during the transaction and is user-defined.

COMMIT EXAMPLE

```
set serveroutput on;
DECLARE
        rollno student.sno%type;
        snm student.sname%type;
        s_age student.age%type;
        s_cr student.course%type;
BEGIN
        rollno := &sno;
        snm := '&sname';
        s_age := &age;
        s_cr := '&course';
        INSERT into student values(rollno,snm,age,course);
        dbms_output.put_line('One record inserted');
        COMMIT;
1.      END;
```

In the above code of PL/SQL block, there is a table called **STUDENT** in the database with columns **sno** as number, **sname** as varchar2, **age** as number and **course** as varchar2.

In the code, we have executed an INSERT statement and then used the COMMIT statement to commit or permanently save the changes into the database.

Instead of the COMMIT statement, if we use the ROLLBACK statement there, then even though the INSERT statement executed successfully, still, after the execution of the PL/SQL block if you will check the Student table in the database, you will not find the new student entry because we executed the ROLLBACK statement and it rolled back the changes.

PL/SQL CODE EXAMPLE WITH SAVEPOINT AND ROLLBACK

Let's add two insert statement in the above code and put a savepoint in between them and then use the ROLLBACK command to revert back changes of one insert statement.

```
set serveroutput on;
DECLARE
        rollno student.sno%type;
        snm student.sname%type;
        s_age student.age%type;
        s_cr student.course%type;
BEGIN
        rollno := &sno;
        snm := '&sname';
        s_age := &age;
        s_cr := '&course';
        INSERT into student values(rollno,snm,age,course);
        dbms_output.put_line('One record inserted');
        COMMIT;
        -- adding savepoint
        SAVEPOINT savehere;
        -- second time asking user for input
        rollno := &sno;
        snm := '&sname';
        s_age := &age;
        s_cr := '&course';
        INSERT into student values(rollno,snm,age,course);
        dbms_output.put_line('One record inserted');
        ROLLBACK [TO SAVEPOINT savehere];
END;
```

After execution of the above code, we will have one entry created in the Student table, while the second entry will be rolled back.

[Oracle PL/SQL Tutorial – Learn Oracle SQL]()

Cursors in PL/SQL

Cursors in **PL/SQL** allow you to retrieve and process multiple rows from a database query.

There are two types:

1.Implicit Cursor –

- Automatically Created When a Query or Manipulation is for a Single Row
- Automatically created for SELECT INTO, INSERT, UPDATE, or DELETE statements.

2.Explicit Cursor –

- Defined by the user for handling multiple rows.Must
- Be Declared by the User.
- Creates a Unit of Storage Called a Result Set

IMPLICIT CURSOR EXAMPLE

```
DECLARE
  emp_name employee.ename%TYPE;
  emp_salary employee.salary%TYPE;
BEGIN
  SELECT ename, salary INTO emp_name, emp_salary
  FROM employee
  WHERE eid = 101;  -- Change to a valid ID


  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_salary);
END;
```

EXPLICIT CURSORS

Declaring an Explicit Cursor

```
CURSOR CursorName IS SelectStatement;
```

Opening an Explicit Cursor

```
OPEN CursorName;
```

Accessing Rows from an Explicit Cursor

```
FETCH CursorName INTO RowVariables;
```

CURSORS

Declaring Variables of the Proper Type with %TYPE

```
        VarName TableName.FieldName%TYPE;
```

Declaring Variables to Hold An Entire Row

```
        VarName CursorName%ROWTYPE;
```

Releasing the Storage Area Used by an Explicit Cursor

```
        CLOSE CursorName;
```

CURSOR CONTROL WITH LOOPS

- Need a Way to Fetch Repetitively
- Need a Way to Determine How Many Rows to Process With a Cursor
  - Cursor Attributes
    - **CursorName%ROWCOUNT – Number of Rows in a Result Set**
    - **CursorName%FOUND – True if a Fetch Returns a Row**
    - **CursorName%NOTFOUND – True if Fetch Goes Past Last Row**

CURSOR FOR LOOP

- Processing an Entire Result Set Common
- Special Form of FOR … IN to Manage Cursors
- No Need for Separate OPEN, FETCH and CLOSE statements
- Requires %ROWTYPE Variable

EXPLICIT CURSOR EXAMPLE

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, salary FROM employee;
    emp_name employee.ename%TYPE;
  emp_salary employee.salary%TYPE;
BEGIN
  OPEN emp_cursor;
    LOOP
    FETCH emp_cursor INTO emp_name, emp_salary;
    EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_name || ', Salary: ' || emp_salary);
  END LOOP;
CLOSE emp_cursor;
END;
```

CURSOR WITH PARAMETERS

```
DECLARE

  CURSOR emp_cursor(p_salary NUMBER) IS

    SELECT ename, salary FROM employee WHERE salary > p_salary;

   emp_name employee.ename%TYPE;

  emp_salary employee.salary%TYPE;

BEGIN

  OPEN emp_cursor(5000);  -- Change salary limit as needed

   LOOP

    FETCH emp_cursor INTO emp_name, emp_salary;

    EXIT WHEN emp_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_name || ', Salary: ' || emp_salary);

  END LOOP;

    CLOSE emp_cursor;

END;

/
```

USING FOR LOOP WITH CURSOR

```
BEGIN
  FOR emp_record IN (SELECT ename, salary FROM employee WHERE
salary > 5000)
LOOP
    DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_record.ename || ', Salary: '
|| emp_record.salary);
  END LOOP;
END;
/
```

CURSOR WITH UPDATE STATEMENT

```
DECLARE
  CURSOR emp_cursor IS
    SELECT eid, salary FROM employee WHERE salary <= 50000;
  v_emp_id employee.eid%TYPE;
  v_salary employee.salary%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp_id, v_salary;
    EXIT WHEN emp_cursor%NOTFOUND;
 UPDATE employee SET salary = v_salary + 500 WHERE eid = v_emp_id;
  END LOOP;
  CLOSE emp_cursor;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Salaries updated successfully!');
END;
```

STORED PROCEDURE

A **Stored Procedure** in **PL/SQL (Procedural Language/Structured Query Language)** is a **named, precompiled block of SQL and PL/SQL statements** that is stored in the **Oracle Database**. It allows for code reuse, improved performance, and better security.

Stored procedures can take input parameters, perform operations (such as INSERT, UPDATE, DELETE, SELECT), and return output values. They help in modular programming and improve database efficiency.

CHARACTERISTICS OF A STORED PROCEDURE

1. **Stored in the database**: Once created, it resides in the database schema and can be executed multiple times without redefining the logic.

2. **Can accept parameters**: Procedures can take IN, OUT, and IN OUT parameters for input and output operations.

3. **Security & access control**: Permissions can be granted or revoked, ensuring only authorized users execute them.

4. **Supports transaction management**: It can include COMMIT, ROLLBACK, and SAVEPOINT statements.

5. **Supports exception handling**: It can handle exceptions and errors using the EXCEPTION block.

STORED PROCEDURES SYNTAX

```
CREATE PROCEDURE ProcedureName(parameter1 datatype, parameter2 datatype, ...) AS
   Additional declarations of local variables
BEGIN
   executable section
EXCEPTION
   Optional exception section
END;
```

PROCEDURE

```
CREATE OR REPLACE PROCEDURE welcome_message IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL Stored Procedures!');
END;
/
```

To execute the procedure:

```
BEGIN
  welcome_message;
END;
/
```

STORED PROCEDURES

- The first line is called the Procedure Specification.
- The remainder is the Procedure Body.
- A procedure is compiled and loaded in the database as an object.
- Procedures can have parameters passed to them.
- Run a procedure with the PL/SQL EXECUTE command.
- Parameters are enclosed in parentheses.

PROCEDURE WITH IN PARAMETER

```
CREATE OR REPLACE PROCEDURE get_employee_details(emp_id NUMBER) IS
  v_name employee.ename%TYPE;
  v_salary employee.salary%TYPE;
BEGIN
  SELECT ename, salary INTO v_name, v_salary
  FROM employee WHERE eid = emp_id;
DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name || ', Salary: ' || v_salary);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Error: Employee not found.');
END;
/
```

CALLING THE PROCEDURE

```
BEGIN
  get_employee_details(101);  -- Replace with a valid employee_id
END;
/
```

STORED PROCEDURE WITH IN AND OUT PARAMETERS

```
CREATE OR REPLACE PROCEDURE fetch_employee_name(
  emp_id IN NUMBER,
  emp_name OUT VARCHAR2
) IS
BEGIN
  SELECT ename INTO emp_name FROM employee WHERE eid = emp_id;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    emp_name := 'Not Found';
END;
/
```

CALLING THE PROCEDURE

```
DECLARE
  v_name VARCHAR2(50);
BEGIN
  fetch_employee_name(101, v_name);  -- Replace with a valid ID
  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);
END;
/
```

STORED PROCEDURE WITH IN OUT PARAMETER

```
CREATE OR REPLACE PROCEDURE update_salary(
  emp_id IN NUMBER,
  emp_salary IN OUT NUMBER
) IS
BEGIN
  UPDATE employee SET salary = emp_salary WHERE eid = emp_id;
  SELECT salary INTO emp_salary FROM employee WHERE eid = emp_id;
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Error: Employee ID not found.');
END;
```

CALLING THE PROCEDURE

```
DECLARE
  v_salary NUMBER := 6000;  -- New salary
BEGIN
  update_salary(101, v_salary);  -- Replace with a valid ID
  DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || v_salary);
END;
/
```

WHAT IS A FUNCTION IN PL/SQL?

A **PL/SQL function** is a **named PL/SQL block** that performs a specific task and **returns a single value**.

Functions are stored in the database and can be called in SQL statements or PL/SQL blocks.

Features of a PL/SQL Function

✓ **Returns a Single Value** – Every function must return exactly one value using the **RETURN** statement.

✓ **Can Be Used in SQL Queries** – Unlike procedures, functions can be used in SQL **SELECT** statements.

✓ **Supports Parameters** – Functions can accept IN, OUT, and IN OUT parameters.

✓ **Cannot Modify the Database** – A function should not contain INSERT, UPDATE, or DELETE statements unless it is called inside a PL/SQL block.

✓ **Supports Exception Handling** – Errors can be managed using the EXCEPTION block.

Syntax of a PL/SQL Function

```
CREATE OR REPLACE FUNCTION function_name (param1 IN datatype, param2 IN datatype)
 RETURN return_datatype AS
-- Variable declarations (optional)
BEGIN
   -- Function logic
   RETURN value; -- Returns a single value
EXCEPTION
   -- Exception handling (optional)
   WHEN others THEN
      RETURN NULL; -- Return a default value in case of error
END function_name;
/
```

FUNCTION TO CALCULATE SQUARE OF A NUMBER

```
CREATE OR REPLACE FUNCTION get_square (p_number IN NUMBER)
RETURN NUMBER AS
 v_result NUMBER;
BEGIN
   v_result := p_number * p_number;
   RETURN v_result;
END get_square;
/
```

How to Call the Function using SQL Statement:

```
SELECT get_square(5) FROM dual;
```

PL/SQL BLOCK TO CALL THE FUNCTION

```
DECLARE
   v_square NUMBER;
BEGIN
   -- Call the function and store the result in v_square
   v_square := get_square(5);

   -- Print the square value
   DBMS_OUTPUT.PUT_LINE('Square of 5 is: ' || v_square);
END;
/
```

FUNCTION TO GET EMPLOYEE SALARY

```
CREATE OR REPLACE FUNCTION get_salary (p_emp_id IN NUMBER)
 RETURN NUMBER AS
v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary FROM employee WHERE EID = p_emp_id;
   RETURN v_salary;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN 0; -- If employee is not found, return 0
END get_salary;
/
```

Calling Function: SELECT get_salary(101) FROM dual;

PL/SQL BLOCK TO CALL THE FUNCTION

```
DECLARE
   v_emp_salary NUMBER;
BEGIN
   -- Call the function and store the salary in v_emp_salary
   v_emp_salary := get_salary(102);
   -- Print the salary
   DBMS_OUTPUT.PUT_LINE('Employee Salary: ' || v_emp_salary);
END;
/
```

DEFINITION OF SEQUENCE IN SQL (ORACLE 11G)

A **SEQUENCE** is a database object used to generate unique numeric values, often for primary keys. It helps avoid concurrency issues when multiple users insert data simultaneously.

Syntax:

```
CREATE SEQUENCE sequence_name
START WITH start_value
INCREMENT BY increment_value
[ MAXVALUE max_value | NOMAXVALUE ]
[ MINVALUE min_value | NOMINVALUE ]
[ CYCLE | NOCYCLE ]
```

EXAMPLE:

```
SQL> create sequence my_sequence1 start with 1 increment by 1 nocycle;

Sequence created.

SQL> select my_sequence1.nextval from dual;

  NEXTVAL
----------
         1

SQL> select my_sequence1.nextval from dual;

   NEXTVAL
----------
         2
```

```
SQL> insert into emp1(id,name,age)values (my_sequence1.nextval,'Hashvik',11);

1 row created.

SQL> insert into emp1(id,name,age)values (my_sequence1.nextval,'Nagaraj',11);

1 row created.

SQL> select*from emp1;

        ID NAME                    AGE
---------- -------------------- ----------
        21  Nagaraj                11
        22  Hashvik                11
         3  Hashvik                11
         4  Nagaraj                11
```

RETRIEVE THE CURRENT VALUE OF THE SEQUENCE

```
SELECT emp_seq.CURRVAL FROM dual;
```

CURRVAL: Returns the most recently generated sequence number for the session.

Dropping a Sequence

If you need to remove a sequence, use:

```
DROP SEQUENCE emp_seq;
```

WHAT IS A TRIGGER IN PL/SQL?

A **trigger** in PL/SQL is a **stored procedure that automatically executes** when a specified event occurs in a database table or view. It is used for enforcing business rules, auditing changes, and maintaining data integrity.

Types of Triggers in PL/SQL

Triggers can be classified based on **when** and **how** they execute:

1. Based on Timing

- **BEFORE Trigger** – Executes before an INSERT, UPDATE, or DELETE operation.
- **AFTER Trigger** – Executes after an INSERT, UPDATE, or DELETE operation.

2. Based on Event

- **Row-Level Trigger** – Executes for each row affected.

BASIC SYNTAX FOR CREATING A TRIGGER

```
CREATE OR REPLACE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | UPDATE | DELETE}
ON table_name
[ FOR EACH ROW ]
BEGIN
    -- PL/SQL block (code to execute)
END;
/
```

EXAMPLE

```
CREATE OR REPLACE TRIGGER trg_check_salary
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary < 5000 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Salary must be at least 5000.');
    END IF;
END;
/
/
```

DROPPING A TRIGGER

To delete an existing trigger:

```
DROP TRIGGER trg_check_salary;/
/
```

EXCEPTION HANDLING IN PL/SQL

Definition:

Exception handling in PL/SQL allows developers to manage runtime errors and take appropriate actions without terminating the program unexpectedly. It ensures that the program continues running smoothly even when an error occurs.

Syntax:

```
BEGIN
  -- PL/SQL statements that may raise exceptions
EXCEPTION
  WHEN exception_name1 THEN
    -- Handling code for exception_name1
  WHEN exception_name2 THEN
    -- Handling code for exception_name2
  WHEN OTHERS THEN
    -- Handling code for all other exceptions
END;
```

TYPES OF EXCEPTIONS IN PL/SQL

1. **Predefined Exceptions** – These are automatically raised by Oracle when a certain condition is met.
    - Example: NO_DATA_FOUND, ZERO_DIVIDE, TOO_MANY_ROWS
2. **User-Defined Exceptions** – Custom exceptions defined by the user using EXCEPTION and raised using RAISE.
3. **Named System Exceptions** – Oracle assigns names to some system exceptions using PRAGMA EXCEPTION_INIT.

EXAMPLE IN ORACLE 11G:

Handling Predefined Exception (ZERO_DIVIDE)

```
SET SERVEROUTPUT ON;
DECLARE
  v_num NUMBER := 10;
  v_den NUMBER := 0;
  v_result NUMBER;
BEGIN
  v_result := v_num / v_den;  -- This will raise ZERO_DIVIDE
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
END;
/
```

Expected Output:

```
Error: Division by zero is not allowed.
```

Explanation:

1. The program initializes two numbers: v_num = 10 and v_den = 0.
2. An attempt is made to divide v_num by v_den, which raises a **ZERO_DIVIDE** exception.
3. The **EXCEPTION** block catches the error and prints: "Error: Division by zero is not allowed."
4. The program gracefully handles the error instead of abruptly terminating.

EXAMPLE OF USER-DEFINED EXCEPTION

```
SET SERVEROUTPUT ON;
DECLARE
  v_salary NUMBER := 10000;
  salary_too_low EXCEPTION;
BEGIN
  IF v_salary < 5000 THEN
    RAISE salary_too_low;
  END IF;
EXCEPTION
  WHEN salary_too_low THEN
    DBMS_OUTPUT.PUT_LINE('Error: Salary is too low.');
END;
/
```

Example Using WHEN OTHERS (Catching All Exceptions)

```
SET SERVEROUTPUT ON;
DECLARE
  v_num NUMBER := 10;
  v_den NUMBER := 0;
  v_result NUMBER;
BEGIN
  v_result := v_num / v_den;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

KEY POINTS:

USE **WHEN OTHERS** TO CATCH ANY UNEXPECTED ERRORS.

USE **SQLERRM** TO DISPLAY ERROR MESSAGES.

USE **USER-DEFINED EXCEPTIONS** WHEN SPECIFIC BUSINESS LOGIC NEEDS TO BE HANDLED.

ALWAYS HANDLE EXCEPTIONS GRACEFULLY TO AVOID ABRUPT TERMINATION OF THE PROGRAM.

THANK YOU