UNIT-I

1. Database and Database Management System (DBMS) – Definition

What is a Database?

A **database** is a **systematic collection of data** that is stored electronically and can be accessed, managed, and updated efficiently.

- Data in a database is organized in rows and columns (tables).
- It is used to store real-world information such as customer data, sales records, employee information, etc.

Example: A school database that stores data about students, teachers, and courses.

What is a Database Management System (DBMS)?

A **DBMS** is a **software tool** that helps users create, retrieve, update, and manage data in databases.

Functions of DBMS:

1. **Data Storage and Retrieval**: Stores data systematically and allows for fast retrieval.
2. **Data Manipulation**: Inserting, deleting, updating, and querying data.
3. **Data Security**: Provides access control mechanisms.
4. **Backup and Recovery**: Ensures data is safe and recoverable after failure.
5. **Data Integrity Management**: Maintains accuracy and consistency.
6. **Concurrency Control**: Allows multiple users to access data simultaneously without conflict.

2. File Management System and its Disadvantages

What is a File Management System?

Before DBMS, data was stored in **file systems**—individual files managed by operating systems.

- Each application had its own set of files.
- Data was stored in flat files (e.g., .txt, .csv, etc.).
- No centralized control over the data.

Disadvantages of File Management Systems:

Problem	Explanation
1. Data Redundancy	Same data stored in multiple files, wasting space.
2. Data Inconsistency	Due to redundancy, data can be out-of-sync.
3. Lack of Data Integrity	No mechanism to enforce rules or constraints.
4. Difficulty in Accessing Data	Cannot retrieve complex queries easily.
5. No Concurrency Control	Two users accessing the same file may cause conflicts.
6. Poor Security	Basic file-level security only; cannot restrict field-level access.
7. No Backup and Recovery	Manual backups only, prone to data loss.
8. Data Isolation	Data is scattered across files, making relationships hard to establish.

Conclusion: File management is simple but inefficient for handling large volumes of structured data.

3. Benefits of Database Management System (DBMS)

DBMS overcomes the limitations of file-based systems. Here's how:

Major Benefits of DBMS:

Benefit	Description
1. Data Redundancy Control	Centralized data management reduces duplication.

2. Improved Data Consistency	Single source of truth across the system.
3. Data Integrity	Constraints (like primary key, foreign key) ensure accurate data.
4. Security	Controlled access via user roles and permissions.
5. Data Sharing	Multiple users can work on the data simultaneously.
6. Backup and Recovery	Automated systems to recover data after failures.
7. Easy Data Access	Uses SQL (Structured Query Language) for complex queries.
8. Data Independence	Data structure changes don't affect application programs.
9. Transaction Management	Ensures complete and consistent execution of transactions.
10. Scalability	Can handle increasing amounts of data and users.

4. RDBMS – Definition and Features

What is an RDBMS (Relational Database Management System)?

An **RDBMS** is an advanced type of DBMS that stores data in the form of **tables (relations)**. Each table consists of **rows (records)** and **columns (fields)**.

Key Features of RDBMS:

1. **Tables as Data Storage**: Data is organized in relations (tables).
2. **Primary and Foreign Keys**: Define relationships between tables.
3. **Normalization**: Eliminates redundancy and improves data integrity.
4. **SQL Support**: Uses SQL to interact with data.
5. **Data Integrity**: Enforces rules using constraints.
6. **Transactions Support**: Ensures atomic, consistent, isolated, and durable operations (ACID properties).
7. **Multi-user Support**: Allows concurrent data access.
8. **Security**: User authentication, privileges, and access control.

Examples of RDBMS:

- MySQL
- Oracle
- PostgreSQL

- Microsoft SQL Server
- IBM DB2

5. DBMS vs RDBMS – Detailed Comparison

Feature	DBMS	RDBMS
Data Format	Stores data as files or hierarchical models	Stores data in tabular form (tables)
Data Relationship	No relationships between data	Supports relationships using keys
Data Redundancy	High	Reduced through normalization
Data Integrity	Not enforced	Enforced via constraints (e.g., PK, FK)
Normalization	Not supported	Fully supported
Query Language	May or may not use SQL	Uses SQL for querying and managing data
ACID Transactions	Limited or no support	Full support (Atomicity, Consistency, etc.)
Multi-user Access	Limited	Efficient multi-user access
Security Features	Basic	Advanced user and role management
Example Systems	XML DB, IMS, File Systems	Oracle, MySQL, SQL Server, PostgreSQL

Unit-II

1. E. F. Codd's Rules for Relational Databases

Who is E. F. Codd?

Dr. Edgar F. Codd, a British computer scientist, introduced the **relational model** in 1970 and proposed **12 rules** (originally 13, Rule 0 to 12) to define a **true relational database system**.

These rules serve as guidelines to evaluate whether a database management system is fully relational.

Codd's 13 Rules:

Rule No.	Rule Name	Description
0	Foundation Rule	System must manage databases entirely through its relational capabilities.
1	Information Rule	All data is stored as values in tables (relations).
2	Guaranteed Access Rule	Each piece of data must be accessible using a combination of table name, column name, and primary key.
3	Systematic Treatment of Nulls	DBMS must support NULLs (unknown/missing values) and treat them systematically.
4	Dynamic Online Catalog	Catalog (metadata) should be stored as tables and accessible using SQL.
5	Comprehensive Data Sub-language Rule	Must support a language (like SQL) for data definition, manipulation, and access.
6	View Updating Rule	All views that can be updated theoretically must be updatable.
7	High-Level Insert, Update, Delete	Must support set-level operations, not just row-level.
8	Physical Data Independence	Changing physical storage does not affect the application.
9	Logical Data Independence	Changing logical schema (e.g., adding a column) should not affect applications.

10	Integrity Independence	Integrity constraints must be stored in the catalog, not in application code.
11	Distribution Independence	Must work without changes regardless of data being distributed.
12	Non-subversion Rule	Low-level operations should not bypass security or integrity constraints.

2. Normalization: 1NF, 2NF, 3NF, BCNF

What is Normalization?

Normalization is a process of organizing data in a database to reduce **redundancy** and improve **data integrity**.

It involves breaking large tables into smaller, related tables and defining relationships between them.

- **1NF – First Normal Form**

📌 Rule:

- Eliminate repeating groups.
- Each cell must contain **atomic (indivisible)** values.
- Each record must be **unique**.

🕮 Example (Before 1NF):

Student_ID	Name	Courses
101	Alice	Math, Science
102	Bob	English

➞ **Problem**: Courses contain multiple values (not atomic).

After 1NF:

Student_ID	Name	Course
101	Alice	Math
101	Alice	Science
102	Bob	English

- **2NF – Second Normal Form**

📌 Rule:

- Must be in **1NF**.
- Remove **partial dependencies** (i.e., no non-prime attribute should depend on part of a composite key).

🕮 Example (Before 2NF):

Student_ID	Course_ID	Student_Name	Course_Name
101	C01	Alice	Math

➞ **Problem**: Student_Name depends only on Student_ID, not the whole composite key.

✅ After 2NF:

Student Table

Student_ID	Student_Name
101	Alice

Course Table

Course_ID	Course_Name
C01	Math

Enrollment Table

Student_ID	Course_ID
101	C01

- **3NF – Third Normal Form**

📌 Rule:

- Must be in **2NF**.
- Remove **transitive dependencies** (non-prime attribute depending on another non-prime attribute).

🕮 Example (Before 3NF):

Emp_ID	Emp_Name	Dept_ID	Dept_Name
1	John	D1	HR

➞ **Problem**: Dept_Name depends on Dept_ID, which is not a primary key.

✅ After 3NF:

Employee Table

Emp_ID	Emp_Name	Dept_ID
1	John	D1

Department Table

Dept_ID	Dept_Name
D1	HR

- **BCNF – Boyce-Codd Normal Form**

📌 Rule:

- A stronger version of 3NF.
- For every **functional dependency A → B**, A should be a **super key**.

🕮 Example:

Professor	Subject	Department
A	DBMS	CS
A	OS	CS

➞ If a professor can teach multiple subjects but is always in the same department, there's a **functional dependency**: Professor → Department.

This violates BCNF if Professor is not a superkey in the relation.

3. Relational Database Terminology

Term	Definition	Example
Relation	A **table** with rows and columns. Represents an entity set.	Student, Employee tables
Tuple	A **row** in a relation. Represents a single record.	One student's data: (101, Alice, CS)
Attribute	A **column** in a relation. Represents a field.	Student_ID, Name, Course
Cardinality	Number of **tuples (rows)** in a relation.	If a table has 100 rows, cardinality = 100
Degree	Number of **attributes (columns)** in a relation.	If a table has 4 columns, degree = 4
Domain	The **set of valid values** for an attribute.	For Age attribute: 0–120

🔁 Example Table: STUDENT

Student_ID	Name	Age	Department
101	Alice	20	CS
102	Bob	21	IT

- **Relation**: STUDENT
- **Tuple**: (101, Alice, 20, CS)
- **Attributes**: Student_ID, Name, Age, Department
- **Cardinality**: 2 (2 rows)
- **Degree**: 4 (4 columns)
- **Domain of Age**: Integer values between 17 and 30 (depending on business rules)

UNIT-III

1. Keys in Relational Databases

What is a Key?

A **key** is an attribute or a set of attributes that is used to **identify rows (tuples)** uniquely in a table and to define relationships between tables.

1. Super Key

- A **super key** is any combination of attributes that can uniquely identify a row in a table.
- It **may contain extra attributes** that are not necessary for uniqueness.

🕮 Example:

For a STUDENT table:

Roll_No	Name	Email
101	Alice	alice@example.com
102	Bob	bob@example.com

Possible super keys:

- {Roll_No}
- {Roll_No, Name}
- {Email}
- {Roll_No, Email}

➞ All these combinations can uniquely identify a student.

2. Candidate Key

- A **candidate key** is a **minimal super key** — a super key with **no unnecessary attributes**.
- A table can have **multiple candidate keys**, but only **one is chosen** as the **primary key**.

🕮 Example:

From the above STUDENT table:

- {Roll_No} and {Email} are **candidate keys**.
- {Roll_No, Email} is not a candidate key because it's not minimal.

3. Primary Key

- A **primary key** is a **chosen candidate key** that **uniquely identifies** each record in a table.
- Cannot be NULL or duplicate.
- Each table can have **only one primary key**.

🕮 Example:

CREATE TABLE STUDENT (Roll_No INT PRIMARY KEY, Name VARCHAR(50), Email VARCHAR(100));

4. Foreign Key

- A **foreign key** is an attribute in one table that **refers to the primary key** of another table.
- It creates a **relationship** between two tables.

🕮 Example:

CREATE TABLE DEPARTMENT (Dept_ID INT PRIMARY KEY, Dept_Name VARCHAR(50));

CREATE TABLE EMPLOYEE (Emp_ID INT PRIMARY KEY, Emp_Name VARCHAR(50), Dept_ID INT, FOREIGN KEY (Dept_ID) REFERENCES DEPARTMENT(Dept_ID));

➞ EMPLOYEE.Dept_ID is a **foreign key** referencing DEPARTMENT.Dept_ID.

2. Structured Query Language (SQL)

Features of SQL:

1. **Standardized Language**: ANSI and ISO standards.
2. **Non-Procedural**: Focuses on *what* to do, not *how* to do.
3. **Multiple Capabilities**:
   - DDL (Data Definition Language)
   - DML (Data Manipulation Language)
   - DCL (Data Control Language)
   - TCL (Transaction Control Language)
4. **Powerful Querying** using SELECT, JOIN, GROUP BY, etc.
5. **Platform Independent**.
6. **Security Control** via GRANT, REVOKE.

SQL*PLUS

- **SQL*Plus** is an **Oracle proprietary command-line tool** used to execute SQL and PL/SQL commands.
- Used for:
  - Writing scripts
  - Formatting reports
  - Connecting to Oracle database
  - Running queries and DDL/DML commands

SQL vs SQL*Plus

Feature	SQL	SQL*Plus
Type	Language	Oracle tool (interface)
Purpose	To query and manage data	To execute SQL commands
Format Output	Does not format output	Can format output using SET, COLUMN
Dependency	Works in many DB systems	Works only in Oracle
Commands	SELECT, INSERT, etc.	SET, SPOOL, DESCRIBE, etc.

Rules for SQL:

1. SQL is **not case-sensitive**, but **keywords** are written in **UPPERCASE**.
2. **Statements end with a semicolon (;)**.
3. **Strings are enclosed in single quotes** ('value').
4. **Comments**:
    - Single line: -- comment
    - Multi-line: /* comment */

SQL Delimiters:

Delimiters separate different elements in SQL syntax.

Delimiter	Use
,	Separate column names/values
;	End of SQL statement
' '	Enclose string values
()	Grouping columns or conditions
= != >	Comparison operators

Components of SQL:

1. **DDL (Data Definition Language)** – Create or modify table structure CREATE, ALTER, DROP
2. **DML (Data Manipulation Language)** – Insert or update data INSERT, UPDATE, DELETE
3. **DCL (Data Control Language)** – Control access to data GRANT, REVOKE
4. **TCL (Transaction Control Language)** – Manage transactions COMMIT, ROLLBACK, SAVEPOINT
5. **DQL (Data Query Language)** – Retrieve data SELECT

3. Constraints in SQL

What are Constraints?

Constraints are rules applied to table columns to ensure **data integrity and correctness**.

They prevent invalid data from being entered into the database.

Types of Data Constraints (Column Level)

Constraint	Description	Example
UNIQUE	Ensures all values in a column are different	Email column in a user table
NOT NULL	Column must have a value, cannot be NULL	Name must be entered
CHECK	Validates a condition before inserting/updating	Age > 18
NULL	Allows column to have null values (default behavior)	Optional middle name

🕮 Examples of Each Constraint:

- **NOT NULL**

```
CREATE TABLE EMPLOYEE (
Emp_ID INT NOT NULL,
Emp_Name VARCHAR(50) NOT NULL
```

);

- **UNIQUE**

CREATE TABLE STUDENT (
Roll_No INT PRIMARY KEY,
Email VARCHAR(100) UNIQUE
);

- **CHECK**

CREATE TABLE ACCOUNT (
Acc_ID INT PRIMARY KEY,
Balance DECIMAL(10,2),
CHECK (Balance >= 0)
);

- **NULL (Default behavior)**

CREATE TABLE CONTACT (
Contact_ID INT PRIMARY KEY,
Middle_Name VARCHAR(50) NULL
);

UNIT - IV

1. Relational Algebra – Introduction

Definition:

Relational Algebra is a **procedural query language** that uses **set theory operations** to manipulate and retrieve data from **relational databases**.

- It forms the **theoretical foundation** for SQL.
- It takes **one or more relations (tables)** as input and **produces a new relation** as output.

---

2. Basic Operations in Relational Algebra

1. Select (σ)

- Filters rows based on a **condition**.
- **Syntax**: σ<condition>(Relation)

🕮 Example:

σ Age > 18 (STUDENT)

➞ Selects all students older than 18.

2. Project (π)

- Selects **specific columns** (attributes) from a relation.
- **Syntax**: π<attribute_list>(Relation)

🕮 Example:

π Name, Age (STUDENT)

➞ Retrieves only Name and Age columns.

3. Union (∪)

- Combines **rows** from two relations.
- Duplicate rows are **eliminated**.
- Relations must be **union-compatible** (same number and type of attributes).

🕮 Example:

STUDENT_2023 ∪ STUDENT_2024

4. Set Difference (−)

- Returns tuples that are in **one relation but not in the other**.
- Also requires **union compatibility**.

🕮 Example:

STUDENT − GRADUATES

➞ Returns students who haven’t graduated.

5. Intersection (∩)

- Returns **common tuples** between two relations.

🕮 Example:

STUDENT ∩ SCHOLARSHIP_STUDENTS

6. Cartesian Product (×)

- Returns the **cross product** of two relations: all possible combinations of rows.
- Result has all attributes from both tables.

📖 Example:

STUDENT × COURSE

➞ Every student paired with every course.

3. Join Operations

1. Join (⋈)

- Combines tuples from two relations **based on a common attribute**.

📖 Example (Theta Join):

STUDENT ⋈ STUDENT.Dept_ID = DEPARTMENT.Dept_ID

2. Natural Join (⋈)

- A **special type of join** that automatically joins on **all common attributes**.
- Removes duplicate columns.

📖 Example:

STUDENT ⋈ DEPARTMENT

➞ Matches on Dept_ID if both tables contain it.

4. Types of Queries

I. Simple Queries

- Basic **SELECT-FROM-WHERE** statements.
- Often used with one table.

📖 SQL Example:

```
SELECT Name FROM STUDENT WHERE Age > 18;
```

II. Nested Queries (Subqueries)

- A query **inside another query**.
- Can be used in WHERE, FROM, or SELECT clause.

🕮 Example:

```
SELECT Name FROM STUDENT
WHERE Dept_ID IN (
  SELECT Dept_ID FROM DEPARTMENT WHERE Dept_Name = 'CS'
);
```

3. Join Queries

- Queries using **JOIN clauses** to combine data from multiple tables.

🕮 Example:

```
SELECT S.Name, D.Dept_Name
FROM STUDENT S
JOIN DEPARTMENT D ON S.Dept_ID = D.Dept_ID;
```

4. Semi-Join

- Returns rows from the **first relation** that have **matching rows in the second**, but doesn't include the second relation’s attributes.

🕮 Conceptual Example:

STUDENT ⋉ DEPARTMENT

➞ Only returns columns from STUDENT that have matching Dept_ID in DEPARTMENT.

💡 Note: SQL doesn’t have an explicit SEMI-JOIN; it's emulated using EXISTS or IN.

🕮 SQL Equivalent:

```
SELECT Name FROM STUDENT
WHERE EXISTS (
  SELECT 1 FROM DEPARTMENT
  WHERE STUDENT.Dept_ID = DEPARTMENT.Dept_ID
);
```

5. Self-Join

- A table is joined **with itself**.
- Requires use of **aliases**.

🕮 Example:

```
SELECT A.Name AS Employee, B.Name AS Manager
FROM EMPLOYEE A, EMPLOYEE B
WHERE A.Manager_ID = B.Emp_ID;
```

✓ Summary Table of Relational Algebra Operations

Operation	Symbol	Function	SQL Equivalent
Select	σ	Filters rows based on condition	WHERE clause
Project	π	Selects specific columns	SELECT clause
Union	∪	Combines unique rows	UNION
Difference	−	Rows in one table not in another	MINUS / EXCEPT
Intersection	∩	Rows common to both tables	INTERSECT
Cartesian Product	×	All combinations of rows	Cross Join
Join	⋈	Combines rows with matching values	JOIN
Natural Join	⋈	Join on common attributes automatically	NATURAL JOIN
Semi-Join	⋉	Filters one relation based on another	IN, EXISTS
Self-Join	—	Join a table to itself	Alias-based join

Final Notes:

- **Relational Algebra** is more theoretical; **SQL** is its practical implementation.
- SQL supports all relational algebra operations through clauses like SELECT, JOIN, WHERE, etc.
- Understanding **joins** and **query nesting** is essential for mastering complex SQL queries.

UNIT-V

1. Basic Introduction to PL/SQL

What is PL/SQL?

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension to SQL. It combines the **data manipulation power of SQL** with the **processing power of procedural languages**.

- PL/SQL is used to **write programs, procedures, and functions** inside the Oracle database.
- It is **block-structured** and supports **control structures**, **loops**, **conditions**, **exception handling**, and **modular programming**.

2. Advantages of PL/SQL

Advantage	Description
Block Structure	Organizes code into logical blocks (declarations, execution, exception handling).
Tight Integration with SQL	Allows SQL operations and transaction control within code.
Better Performance	Reduces communication between application and database server (e.g., loops execute server-side).
Code Reusability	Supports procedures, functions, packages.
Improved Security	Controls access through stored procedures and packages.
Error Handling	Robust **exception handling** capabilities.
Portability	Runs on any Oracle-supported platform.

3. The Generic PL/SQL Block Structure

```
DECLARE
  -- Declarations (variables, constants, cursors, etc.)
BEGIN
  -- Executable statements (SQL and procedural code)
```

```
EXCEPTION
  -- Error handling
END;
```

□ Sections of a PL/SQL Block:

1. **DECLARE**: Optional – Used to declare variables, constants, cursors.
2. **BEGIN**: Mandatory – Contains executable code.
3. **EXCEPTION**: Optional – Catches and handles runtime errors.
4. **END**: Mandatory – Ends the block.

4. Literals, Variables, Constants, Comparisons, Comments

Literals

Literals are **fixed values** that appear directly in code.

Type	Example
Number	100, 3.14
Character	'Hello'
Boolean	TRUE, FALSE
Date	TO_DATE('2024-05-19', 'YYYY-MM-DD')

Variables

Used to **store data** during block execution.

```
DECLARE
  v_name VARCHAR2(50);
  v_salary NUMBER := 50000;
```

- Variable names begin with a letter and can include letters, digits, and underscores.
- Assignment: := (colon equals)

Constants

A **constant** is a variable whose value **cannot be changed** after assignment.

```
DECLARE
  pi CONSTANT NUMBER := 3.14159;
```

Comparisons

Used in conditions with comparison operators:

Operator	Meaning
=	Equal
!= or <>	Not equal
>, <	Greater/Less than
>=, <=	Greater/Equal etc.

```
IF salary > 50000 THEN
  DBMS_OUTPUT.PUT_LINE('High Salary');
END IF;
```

Comments

- **Single-line**:
  -- This is a single line comment
- **Multi-line**:
  /* This is a multiline comment */

5. Control Structures in PL/SQL

PL/SQL supports three types of control structures:

Conditional Control (IF Statements)

🕮 Syntax:

```
IF condition THEN
  -- statements
ELSIF another_condition THEN
  -- statements
ELSE
  -- statements
END IF;
```

🕮 Example:

```
IF grade >= 90 THEN
  DBMS_OUTPUT.PUT_LINE('A Grade');
ELSIF grade >= 75 THEN
  DBMS_OUTPUT.PUT_LINE('B Grade');
ELSE
  DBMS_OUTPUT.PUT_LINE('C Grade');
END IF;
```

Iterative Control (LOOPs)

Used to **repeat** execution of code blocks.

◆ BASIC LOOP

```
LOOP
  -- statements
  EXIT WHEN condition;
END LOOP;
```

◆ WHILE LOOP

```
WHILE condition LOOP
  -- statements
END LOOP;
```

◆ FOR LOOP

```
FOR i IN 1..5 LOOP
  DBMS_OUTPUT.PUT_LINE(i);
END LOOP;
```

➞ Iterates from 1 to 5.

3. Sequential Control (GOTO and NULL)

◆ GOTO Statement

Jumps to a labeled part of the program.

```
BEGIN
  GOTO skip;
  DBMS_OUTPUT.PUT_LINE('This will be skipped');
<<skip>>
  DBMS_OUTPUT.PUT_LINE('Jumped to here');
END;
```

◆ NULL Statement

Used when no action is required (a placeholder).

```
IF salary < 0 THEN
  NULL; -- Do nothing
END IF;
```

UNIT-VI

1. Cursors in PL/SQL

What is a Cursor?

A **cursor** is a pointer or handle to the result set of a **SQL query**. It allows **row-by-row** processing of query results in PL/SQL.

Types of Cursors

Type	Managed By	Usage
Implicit Cursor	Oracle	Automatically created for DML and SELECT INTO statements
Explicit Cursor	Developer	Manually declared and controlled for more complex queries

---

I. Implicit Cursor

- Used automatically by Oracle when executing:
  - INSERT, UPDATE, DELETE, SELECT INTO
- **Attributes** of Implicit Cursor:
  - %FOUND: Returns TRUE if a row is affected
  - %NOTFOUND: Returns TRUE if no row is affected
  - %ROWCOUNT: Number of rows affected
  - %ISOPEN: Always FALSE for implicit cursor

🕮 Example:

```
DECLARE
  v_name STUDENT.NAME%TYPE;
BEGIN
  SELECT NAME INTO v_name FROM STUDENT WHERE ROLL_NO = 101;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
  END IF;
END;
```

II. Explicit Cursor

- Defined and controlled manually.
- Useful for **multi-row queries**.

🕮 Syntax:

```
DECLARE
  CURSOR c_emp IS SELECT emp_name, salary FROM employee;
  v_name employee.emp_name%TYPE;
  v_salary employee.salary%TYPE;
BEGIN
  OPEN c_emp;
  LOOP
    FETCH c_emp INTO v_name, v_salary;
    EXIT WHEN c_emp%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name || ' earns ' || v_salary);
  END LOOP;
  CLOSE c_emp;
END;
```

2. PL/SQL Database Objects: Procedures and Functions

What is a Procedure?

A **procedure** is a **named PL/SQL block** that performs a task and may or may not return a value.

🕮 Syntax:

```
CREATE OR REPLACE PROCEDURE greet_user(name IN VARCHAR2) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, ' || name);
END;
```

➞ **Call with:** `EXEC greet_user('John');`

What is a Function?

A **function** is similar to a procedure, but it **must return a value**.

🕮 Syntax:

```
CREATE OR REPLACE FUNCTION get_bonus(salary IN NUMBER)
RETURN NUMBER IS
BEGIN
  RETURN salary * 0.10;
END;
```

➞ **Call with:** `SELECT get_bonus(50000) FROM DUAL;`

Advantages of Procedures and Functions

Feature	Benefit
Code Reuse	Define once, use multiple times
Modularity	Divide complex programs into subprograms
Maintenance	Easier to maintain and update
Security	Access data only via stored logic
Performance	Stored and executed at the database server

3. Triggers in PL/SQL

What is a Trigger?

A **trigger** is a **stored PL/SQL block** that is **automatically executed** (fired) when a **specific database event** (INSERT, UPDATE, DELETE) occurs on a table or view.

Syntax of Trigger:

```
CREATE OR REPLACE TRIGGER trg_before_insert
BEFORE INSERT ON student
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before Insert Trigger Fired');
END;
```

Triggers vs Procedures

Feature	Trigger	Procedure
Invocation	Automatically fired by database event	Manually invoked using EXEC or call
Timing	BEFORE or AFTER DML events	Called when needed
Returns Value	Cannot return values	Can return values
Use Case	Enforcing business rules, auditing	Reusable logic, calculations

Types of Triggers

Type	Description
BEFORE Trigger	Executes **before** the triggering DML event
AFTER Trigger	Executes **after** the DML event
INSTEAD OF Trigger	Used with **views** to replace DML actions
Row-Level Trigger	Fires **once for each row** affected
Statement-Level Trigger	Fires **once for the whole DML statement**, not per row
Compound Trigger	Combines multiple timing points in a single trigger (Oracle 11g+)

🕮 Example: AFTER INSERT Trigger

```
CREATE OR REPLACE TRIGGER trg_log_insert
AFTER INSERT ON student
FOR EACH ROW
BEGIN
INSERT INTO student_log (roll_no, name, action)
VALUES (:NEW.roll_no, :NEW.name, 'INSERTED');
END;
```