**1. Define Database and Database Management System (DBMS)**

* **Database**: A database is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are often managed using a Database Management System (DBMS), which provides tools for storing, modifying, and extracting data. The structure of a database is usually designed using tables, schemas, views, and indexes.
* **Database Management System (DBMS)**: A DBMS is software that provides an interface for users to interact with databases. It helps in managing, storing, retrieving, and manipulating data efficiently. It ensures data integrity, security, and consistency while allowing multiple users to access the data concurrently. Examples of DBMS include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

**2. Describe Primary Key**

A **Primary Key** is a unique identifier for a record in a relational database table. It ensures that each record can be uniquely identified and helps in maintaining data integrity.

* A primary key column cannot contain **NULL** values.
* Every table can have only **one primary key**, which may consist of a single or multiple columns (composite key).
* It guarantees that the data in that column(s) will be unique across the table.

**3. E-R Diagram for Library Management System**

Here is a simplified outline of an Entity-Relationship (E-R) diagram for a Library Management System:

Entities:

* **Book**: Stores information about books.
  + Attributes: Book\_ID (PK), Title, Author, ISBN, Publication\_Year
* **Member**: Stores information about library members.
  + Attributes: Member\_ID (PK), Name, Address, Email, Phone
* **Loan**: Represents the loaning of books to members.
  + Attributes: Loan\_ID (PK), Member\_ID (FK), Book\_ID (FK), Issue\_Date, Return\_Date

Relationships:

* A **Member** can borrow multiple **Books** (1 to Many between Member and Loan).
* A **Book** can be borrowed multiple times by different **Members** (1 to Many between Book and Loan).
* The **Loan** entity connects **Member** and **Book**.

**E-R Diagram:**

scss

Copy code

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

| Member | | Loan | | Book |

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

| Member\_ID (PK) |----<| Loan\_ID (PK) | | Book\_ID (PK) |

| Name | | Member\_ID (FK) | | Title |

| Address | | Book\_ID (FK) | | Author |

| Email | | Issue\_Date | | ISBN |

| Phone | | Return\_Date | | Publication\_Year|

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

**4. Normalize the Following Table of EMP to 3NF**

The given table is:

scss

Copy code

EMP(empno, ename, mgr, job, deptno, loc, dname)

**Step 1: First Normal Form (1NF)**  
To achieve 1NF, we need to ensure all attributes contain atomic values and each record is unique.

* Remove repeating groups and ensure each column contains atomic values.
* **EMP** already satisfies 1NF, as each attribute contains atomic values (single values per field).

**Step 2: Second Normal Form (2NF)**  
To achieve 2NF, the table must first be in 1NF, and there should be no partial dependencies (i.e., non-prime attributes should depend on the entire primary key).

* **empno** is the primary key for the **EMP** table.
* However, the **dname** and **loc** depend only on **deptno**, not the entire primary key.
* Hence, we decompose the table to remove partial dependencies.

Tables in 2NF:

1. **EMP**:

scss

Copy code

empno (PK), ename, mgr, job, deptno (FK)

1. **DEPT**:

scss

Copy code

deptno (PK), dname, loc

**Step 3: Third Normal Form (3NF)**  
To achieve 3NF, the table must first be in 2NF, and there should be no transitive dependencies (i.e., non-prime attributes should not depend on other non-prime attributes).

* In **EMP**, **mgr** depends on **empno**, but **mgr** is a reference to **empno**, which is already in the **EMP** table.
* This is a transitive dependency, so we need to remove it.

Decompose into the following tables:

1. **EMP**:

scss

Copy code

empno (PK), ename, job, deptno (FK), mgr (FK)

1. **DEPT**:

scss

Copy code

deptno (PK), dname, loc

1. **MANAGER**:

scss

Copy code

mgr (PK), empno (FK)

* Here, **mgr** is now treated as a foreign key referring to **empno**, so no transitive dependencies remain.

**Final Schema in 3NF:**

1. **EMP**:

scss

Copy code

empno (PK), ename, job, deptno (FK), mgr (FK)

1. **DEPT**:

scss

Copy code

deptno (PK), dname, loc

1. **MANAGER**:

scss

Copy code

mgr (PK), empno (FK)

2.

1) CREATE TABLE stud (

Rollno INT PRIMARY KEY,

Studname VARCHAR(50),

Percentage DECIMAL(5,2),

CONSTRAINT chk\_percentage CHECK (Percentage <= 100)

);

2) ALTER TABLE stud

ADD City VARCHAR(100);

3) ALTER TABLE stud

MODIFY Studname VARCHAR(60);

3.

1) DELETE FROM emp

WHERE ename = 'SMITH';

2) UPDATE emp

SET job = 'MANAGER'

WHERE ename = 'ADAMS';

3) SELECT empno, sal

FROM emp;

4.

1) CREATE USER Jay IDENTIFIED BY jay\_password;

2) GRANT CREATE TABLE, CREATE VIEW TO Jay;

3) GRANT SELECT, INSERT, UPDATE ON Emp TO Jay;

4) ALTER USER Jay IDENTIFIED BY newjay\_password;

5) REVOKE CREATE TABLE, CREATE VIEW FROM Jay;

6) REVOKE SELECT, INSERT, UPDATE ON Emp FROM Jay;

7) CREATE ROLE emp\_pvr;

8) GRANT CREATE TABLE, CREATE VIEW TO emp\_pvr;

5.

1) COMMIT;

2) DELETE FROM EMP WHERE EMP\_ID = 101;

ROLLBACK;

3) SAVEPOINT BeforeSalaryUpdate;

4)  **Atomicity**:

* This property ensures that all operations within a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged.
* In simple terms, "all or nothing."

 **Consistency**:

* A transaction must bring the database from one valid state to another. After a transaction, all data must be in a consistent state, adhering to all defined rules, constraints, and relationships.
* For example, if a rule specifies that an employee's salary cannot be negative, the database should not allow such a transaction.

 **Isolation**:

* Transactions should be isolated from each other, meaning the intermediate state of a transaction is invisible to other transactions. Even if multiple transactions occur simultaneously, each should appear as though it is the only transaction being processed.
* This prevents data inconsistencies caused by concurrent access.

 **Durability**:

* Once a transaction is committed, it becomes permanent. Even if there is a system failure after the commit, the changes made by the transaction are preserved and cannot be lost.
* This ensures the reliability of the database in the face of power failures or crashes.

6.

1) SELECT cust\_id, order\_id, items, amount, (amount + 200) AS total\_amount

FROM Orders;

2) SELECT cust\_id, order\_id, items, amount, (amount - 100) AS offer\_price

FROM Orders;

3) SELECT cust\_id, order\_id, items, amount, (amount \* 5) AS revised\_amount

FROM Orders;

4) SELECT cust\_id, order\_id, items, amount, (amount / 2) AS half\_amount

FROM Orders;

7.

1) SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM Emp

WHERE city = 'Mumbai' AND sal > 50000;

2) SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM Emp

WHERE job = 'Clerk' OR comm = 500;

3) SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM Emp

WHERE sal BETWEEN 20000 AND 50000;

4) SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM Emp

WHERE city IN ('Mumbai', 'Pune', 'Nashik', 'Nagpur');

8.

1)

| **Operator** | **Description** |
| --- | --- |
| **=** | Equal to: Compares if two values are equal. |
| **!=** or **<>** | Not equal to: Compares if two values are not equal. |
| **>** | Greater than: Compares if the left operand is greater than the right operand. |
| **<** | Less than: Compares if the left operand is less than the right operand. |
| **>=** | Greater than or equal to: Compares if the left operand is greater than or equal to the right operand. |
| **<=** | Less than or equal to: Compares if the left operand is less than or equal to the right operand. |
| **BETWEEN** | Used to check if a value is within a specified range. |
| **IN** | Used to check if a value matches any value in a specified list. |
| **LIKE** | Used to search for a specified pattern in a column (often used with wildcards). |
| **IS NULL** | Used to check if a value is NULL (i.e., missing or undefined). |

**2)**

i) **SELECT stu\_name, course\_id**

**FROM Student**

**WHERE percentage >= 60 AND percentage <= 100;**

**ii)** **SELECT stu\_name, course\_id, Roll\_no, percentage**

**FROM Student**

**WHERE Roll\_no > 15;**

**iii) SELECT stu\_name, Roll\_no**

**FROM Student**

**WHERE course\_id != 121;**

**9.**

**1)** **SELECT ename FROM emp1**

**UNION ALL**

**SELECT ename FROM emp2;**

**2)** **SELECT ename FROM emp1**

**UNION**

**SELECT ename FROM emp2;**

**3)** **SELECT ename FROM emp1**

**INTERSECT**

**SELECT ename FROM emp2;**

**4)** **SELECT ename**

**FROM emp1**

**WHERE deptno IS NULL**

**UNION**

**SELECT ename**

**FROM emp2**

**WHERE deptno IS NULL;**

**10.**

**1)** **SELECT CONCAT('Jay', 'IITB') FROM Dual;**

**JayIITB**

**2)** **SELECT LTRIM('Shreya', 's') FROM Dual;**

**Shreya**

**3)** **SELECT UPPER('raj') FROM Dual;**

**RAJ**

**4)** **SELECT RPAD('HR', 10, '\*') FROM Dual;**

**HR\*\*\*\*\*\*\***

**11.**

**1)** **SELECT POWER(12, 3) AS cube\_12 FROM Dual; output—1728**

**SELECT POWER(14, 3) AS cube\_14 FROM Dual; output--2744**

**SELECT POWER(16, 3) AS cube\_16 FROM Dual; output--4096**

**2)** **SELECT empno, ename, sal, CEIL(sal) AS rounded\_sal**

**FROM emp;**

**3)** **SELECT ROUND(123.456, 2) FROM Dual; -- Output: 123.46**

**SELECT ROUND(123.454, 2) FROM Dual; -- Output: 123.45**

**and**

**SELECT TRUNC(123.456, 2) FROM Dual; -- Output: 123.45**

**SELECT TRUNC(123.454, 2) FROM Dual; -- Output: 123.45**

**4)** **SELECT FLOOR(123.67) FROM Dual; -- Output: 123**

**SELECT FLOOR(123.99) FROM Dual; -- Output: 123**

**And**

**SELECT CEIL(123.67) FROM Dual; -- Output: 124**

**SELECT CEIL(123.01) FROM Dual; -- Output: 124**

**12.**

**1)** **SELECT SYSDATE FROM Dual; SYSDATE 22-NOV-24 10:30:15**

**2)** **SELECT LAST\_DAY(SYSDATE) FROM Dual; LAST\_DAY(SYSDATE) 30-NOV-24**

**3) The MONTHS\_BETWEEN() function calculates the number of months between two dates. It returns a positive or negative number depending on the order of the dates.**

**Syntax:**

**sql**

**Copy code**

**MONTHS\_BETWEEN(date1, date2)**

**date1: The first date.**

**date2: The second date.**

**Example:**

**Let's say you want to calculate the number of months between a birthdate (01-JAN-2000) and today's date (SYSDATE):**

**SELECT MONTHS\_BETWEEN(SYSDATE, TO\_DATE('01-JAN-2000', 'DD-MON-YYYY')) AS months\_diff FROM Dual;**

**4)** **SELECT FLOOR(MONTHS\_BETWEEN(SYSDATE, TO\_DATE('01-JAN-1990', 'DD-MON-YYYY')) / 12) AS age\_years FROM Dual;**

**AGE\_YEARS 34**

**13.**

**1)** **SELECT**

**MIN(sal) AS Minimum,**

**MAX(sal) AS Maximum,**

**SUM(sal) AS Sum,**

**AVG(sal) AS Average**

**FROM Emp;**

**Minimum | Maximum | Sum | Average**

**-----------------------------------------**

**2000 | 10000 | 123000 | 5500**

**2)** **SELECT COUNT(DISTINCT mgr) AS "Number of Managers"**

**FROM Emp**

**WHERE mgr IS NOT NULL;**

**Number of Managers**

**--------------------**

**3)** **SELECT (MAX(sal) - MIN(sal)) AS DIFFERENCE**

**FROM Emp;**

**DIFFERENCE**

**-----------**

**8000**

**4)** **SELECT COUNT(\*) AS "Number of Employees with Commission"**

**FROM Emp**

**WHERE deptno = 10 AND comm IS NOT NULL;**

**Number of Employees with Commission**

**--------------------------------------**

**14.**

**1)** **SELECT deptno, MIN(sal) AS "Minimum Salary"**

**FROM Emp**

**GROUP BY deptno;**

**DEPTNO | Minimum Salary**

**-----------------------**

**10 | 2000**

**20 | 2500**

**30 | 3000**

**2)** **SELECT deptno, SUM(sal) AS "Total Salary"**

**FROM Emp**

**GROUP BY deptno;**

**DEPTNO | Total Salary**

**---------------------**

**10 | 35000**

**20 | 45000**

**30 | 30000**

**3)** **SELECT deptno, COUNT(\*) AS "Total Employees"**

**FROM Emp**

**GROUP BY deptno**

**HAVING COUNT(\*) > 5;**

**DEPTNO | Total Employees**

**----------------------**

**10 | 6**

**20 | 8**

**4)** **SELECT empno, ename, job, sal, deptno**

**FROM Emp**

**ORDER BY ename ASC;**

**EMPNO | ENAME | JOB | SAL | DEPTNO**

**----------------------------------------------**

**7839 | Allen | Clerk | 3000 | 10**

**7566 | James | Manager | 5000 | 20**

**7698 | King | President | 10000 | 30**

**7782 | Scott | Analyst | 6000 | 20**

**15.**

**1) Left Outer Join (LEFT JOIN): Returns all rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.**

** Right Outer Join (RIGHT JOIN): Returns all rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for columns from the left table.**

** Full Outer Join (FULL JOIN): Returns all rows when there is a match in one of the tables. If there is no match, NULL values are returned for the missing side.**

**2)** **SELECT e.emp\_id, e.emp\_name, e.dept\_id, d.location**

**FROM Employees e**

**INNER JOIN Departments d ON e.dept\_id = d.dept\_id**

**WHERE e.emp\_name = 'Nikhil';**

**3)** **SELECT e.emp\_id, e.emp\_name, e.dept\_id**

**FROM Employees e**

**INNER JOIN Departments d ON e.dept\_id = d.dept\_id**

**WHERE d.dept\_name = 'Sales';**

**4)** **SELECT e.emp\_id, e.emp\_name, e.dept\_id**

**FROM Employees e**

**INNER JOIN Departments d ON e.dept\_id = d.dept\_id**

**WHERE d.dept\_name != 'Sales';**

**5)** **SELECT e.emp\_name, e.salary**

**FROM Employees e**

**INNER JOIN Employees m ON e.manager\_id = m.emp\_id**

**WHERE m.emp\_name = 'Sumit Patil';**

**16.**

**1)** **CREATE VIEW emp\_view AS**

**SELECT emp\_no, e\_name, salary**

**FROM emp;**

**2)** **UPDATE emp\_view**

**SET e\_name = 'Jay'**

**WHERE emp\_no = 101;**

**3)** **DELETE FROM emp\_view**

**WHERE emp\_no = 105;**

**4)** **DROP VIEW emp\_view;**

**5)** **CREATE OR REPLACE VIEW dept\_view AS**

**SELECT dept\_no, dept\_name, location**

**FROM departments**

**WHERE dept\_no = 'new\_dept\_no'; -- Modify as needed**

**17.**

**1)IF condition THEN**

**-- Statements to execute if condition is true**

**END IF;**

**IF condition THEN**

**-- Statements to execute if condition is true**

**ELSE**

**-- Statements to execute if condition is false**

**END IF;**

**IF condition1 THEN**

**-- Statements if condition1 is true**

**ELSIF condition2 THEN**

**-- Statements if condition2 is true**

**ELSE**

**-- Statements if none of the above conditions are true**

**END IF;**

**2)** **DECLARE**

**num NUMBER;**

**BEGIN**

**num := &num;**

**IF num > 0 THEN**

**DBMS\_OUTPUT.PUT\_LINE('Number is positive');**

**ELSE**

**DBMS\_OUTPUT.PUT\_LINE('Number is not positive');**

**END IF;**

**END;**

**18.**

**1)** **FOR counter IN lower\_bound..upper\_bound LOOP**

**-- Statements to execute in each iteration**

**END LOOP;**

**WHILE condition LOOP**

**-- Statements to execute while condition is true**

**END LOOP;**

**LOOP**

**-- Statements to execute**

**EXIT WHEN condition;**

**END LOOP;**

**2)** **DECLARE**

**num NUMBER; -- To hold the number being checked**

**i NUMBER; -- Loop counter**

**is\_prime BOOLEAN := TRUE; -- Flag to determine if the number is prime**

**BEGIN**

**-- Loop through numbers from 1 to 50**

**FOR num IN 2..50 LOOP**

**is\_prime := TRUE; -- Assume the number is prime**

**-- Check divisibility from 2 to num-1**

**FOR i IN 2..num-1 LOOP**

**IF mod(num, i) = 0 THEN**

**is\_prime := FALSE; -- If divisible, the number is not prime**

**EXIT; -- Exit inner loop if divisible**

**END IF;**

**END LOOP;**

**-- If the number is prime, print it**

**IF is\_prime THEN**

**DBMS\_OUTPUT.PUT\_LINE(num || ' is a prime number');**

**END IF;**

**END LOOP;**

**END;**

**19.**

**1) Sequential Control in PL/SQL**

**Sequential control refers to the default execution flow in PL/SQL, where statements are executed one after the other in the order in which they are written. In addition to the regular flow, PL/SQL allows branching (e.g., using IF statements) and looping (e.g., using FOR or WHILE loops) to alter the order of execution.**

**GOTO Statement in PL/SQL**

**The GOTO statement allows the program to jump to a specified label in the code. It's used to transfer control to another part of the program. While GOTO is rarely used in modern programming due to its potential for making code harder to read, it is sometimes used for control flow in PL/SQL.**

**Syntax:**

**sql**

**Copy code**

**GOTO label\_name;**

**2)** **DECLARE**

**i NUMBER := 1; -- Variable to store the current number**

**BEGIN**

**-- Loop through numbers from 1 to 10**

**LOOP**

**IF i = 5 THEN**

**-- Skip number 5 using GOTO**

**i := i + 1; -- Increment the number**

**GOTO skip\_number; -- Jump to the label 'skip\_number'**

**END IF;**

**-- Print the number**

**DBMS\_OUTPUT.PUT\_LINE(i);**

**-- Increment the number**

**i := i + 1;**

**-- Exit the loop when i exceeds 10**

**IF i > 10 THEN**

**EXIT;**

**END IF;**

**END LOOP;**

**-- Label to skip printing the number 5**

**skip\_number:**

**NULL; -- No operation, just a placeholder for the GOTO statement**

**END;**

**20.**

**1)1. Implicit Cursor:**

**An Implicit Cursor is automatically created by Oracle when a SQL query is executed in PL/SQL. It is used for SELECT INTO, INSERT, UPDATE, and DELETE operations, and you do not need to explicitly declare or manage them. Oracle handles the opening, fetching, and closing of the cursor automatically for these types of SQL statements.**

* **Advantages:**
  + **Simpler to use as it requires no explicit declaration or management.**
  + **Oracle automatically handles the cursor for you, making it easier for simple queries.**
* **Disadvantages:**
  + **Provides limited control over fetching data.**
  + **Cannot fetch multiple rows in complex scenarios, such as when more than one row is returned.**

**2. Explicit Cursor:**

**An Explicit Cursor is explicitly declared by the programmer in PL/SQL when you need more control over data retrieval. It is used when you want to fetch multiple rows, process them one by one, and manage the cursor's lifecycle (open, fetch, close).**

* **Advantages:**
  + **Provides more control over the SQL query execution.**
  + **Allows you to process multiple rows.**
  + **You can fetch rows iteratively and handle complex queries with ease.**
* **Disadvantages:**
  + **Requires more code for declaration, fetching, and closing.**
  + **Can be more complex to manage.**

**2)** **DECLARE**

**-- Declare a cursor to fetch items with price greater than 10000**

**CURSOR item\_cursor IS**

**SELECT item\_id, item\_name, price**

**FROM store**

**WHERE price > 10000;**

**-- Declare a variable to store the count of items**

**item\_count NUMBER := 0;**

**-- Declare a record variable to hold the data from the cursor**

**item\_record item\_cursor%ROWTYPE;**

**BEGIN**

**-- Open the cursor**

**OPEN item\_cursor;**

**-- Loop through the cursor and count the number of items**

**LOOP**

**-- Fetch the data into item\_record**

**FETCH item\_cursor INTO item\_record;**

**-- Exit the loop when no more rows are fetched**

**EXIT WHEN item\_cursor%NOTFOUND;**

**-- Increment the item count**

**item\_count := item\_count + 1;**

**END LOOP;**

**-- Close the cursor**

**CLOSE item\_cursor;**

**-- Display the count of items with price greater than 10000**

**DBMS\_OUTPUT.PUT\_LINE('Number of items with price greater than 10000: ' || item\_count);**

**END;**

**21.**

**1)** **In PL/SQL, predefined exceptions are exceptions that are automatically provided by Oracle for common errors. These exceptions can be caught and handled in a PL/SQL block using the EXCEPTION section. When an error occurs that matches one of these predefined exceptions, PL/SQL automatically raises the corresponding exception.**

**Some common predefined exceptions include:**

1. **ZERO\_DIVIDE: Raised when a division by zero is attempted.**
2. **NO\_DATA\_FOUND: Raised when a SELECT INTO query returns no rows.**
3. **TOO\_MANY\_ROWS: Raised when a SELECT INTO query returns more than one row.**
4. **VALUE\_ERROR: Raised when a variable's value cannot be converted to a different data type (e.g., trying to store a string in a number).**
5. **INVALID\_CURSOR: Raised when a cursor operation is attempted on a cursor that is not open.**
6. **DUP\_VAL\_ON\_INDEX: Raised when trying to insert a duplicate value into a column with a unique constraint.**

**2)** **DECLARE**

**num1 NUMBER; -- Variable to store the first number**

**num2 NUMBER; -- Variable to store the second number**

**result NUMBER; -- Variable to store the result of the division**

**BEGIN**

**-- Accept input from the user (for simplicity, we assign values directly)**

**num1 := &num1; -- User input for the first number**

**num2 := &num2; -- User input for the second number**

**-- Try to divide num1 by num2**

**BEGIN**

**result := num1 / num2; -- Perform the division**

**DBMS\_OUTPUT.PUT\_LINE('Result of ' || num1 || ' divided by ' || num2 || ' is ' || result);**

**EXCEPTION**

**WHEN ZERO\_DIVIDE THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: Division by zero is not allowed.');**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);**

**END;**

**END;**

**22.**

**1) Defining a User-Defined Exception: You can define a user-defined exception using the EXCEPTION keyword. You declare the exception in the DECLARE section of your PL/SQL block.**

**plsql**

**Copy code**

**DECLARE**

**exception\_name EXCEPTION;**

**BEGIN**

**-- code that triggers the exception**

**EXCEPTION**

**WHEN exception\_name THEN**

**-- Exception handling code**

**END;**

**2)** **DECLARE**

**-- Declare the user-defined exception**

**INVALID\_ID EXCEPTION;**

**-- Declare a variable to hold the customer ID entered by the user**

**customer\_id NUMBER;**

**-- Declare a variable to store the count of the customer ID found in the table**

**customer\_count NUMBER;**

**BEGIN**

**-- Accept input from the user**

**customer\_id := &customer\_id; -- Use a substitution variable for user input**

**-- Query to check if the customer ID exists in the customer table**

**SELECT COUNT(\*)**

**INTO customer\_count**

**FROM customer**

**WHERE id = customer\_id;**

**-- If customer\_count is 0, it means the ID doesn't exist**

**IF customer\_count = 0 THEN**

**-- Raise the user-defined exception**

**RAISE INVALID\_ID;**

**ELSE**

**-- If the ID is valid, display the customer details (just an example)**

**DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || customer\_id || ' is valid.');**

**END IF;**

**EXCEPTION**

**-- Handle the user-defined exception**

**WHEN INVALID\_ID THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: Invalid Customer ID entered.');**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);**

**END;**

**23.**

**1)** **CREATE [OR REPLACE] PROCEDURE procedure\_name**

**[parameter\_name datatype [IN | OUT | IN OUT] [, ...]]**

**IS**

**-- Declarations (optional)**

**BEGIN**

**-- Executable statements (logic of the procedure)**

**-- e.g., SQL statements or control structures**

**EXCEPTION**

**-- Exception handling (optional)**

**END procedure\_name;**

**2)** **CREATE OR REPLACE PROCEDURE emp\_count (dept\_no IN NUMBER) IS**

**-- Declare a variable to store the employee count**

**emp\_count NUMBER;**

**BEGIN**

**-- Query to count employees in the given department**

**SELECT COUNT(\*)**

**INTO emp\_count**

**FROM employees**

**WHERE department\_id = dept\_no;**

**-- Output the count of employees in the given department**

**DBMS\_OUTPUT.PUT\_LINE('Number of employees in department ' || dept\_no || ': ' || emp\_count);**

**EXCEPTION**

**-- Handle case where department\_id does not exist or any other error**

**WHEN NO\_DATA\_FOUND THEN**

**DBMS\_OUTPUT.PUT\_LINE('No employees found in department ' || dept\_no);**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);**

**END emp\_count;**

**24.**

**1)** **CREATE [OR REPLACE] FUNCTION function\_name**

**(parameter\_name datatype [, ...])**

**RETURN return\_datatype**

**IS**

**-- Declarations (optional)**

**BEGIN**

**-- Executable statements (the logic of the function)**

**RETURN return\_value; -- Return value to the caller**

**EXCEPTION**

**-- Exception handling (optional)**

**END function\_name;**

**2)** **CREATE OR REPLACE FUNCTION get\_max (val1 IN NUMBER, val2 IN NUMBER)**

**RETURN NUMBER**

**IS**

**BEGIN**

**-- Return the maximum of the two values**

**IF val1 > val2 THEN**

**RETURN val1;**

**ELSE**

**RETURN val2;**

**END IF;**

**END get\_max;**

**n25**

**25**

**1)**

| **Feature** | **Row-Level Trigger** | **Statement-Level Trigger** |
| --- | --- | --- |
| **Fired** | **Once for each row affected by the operation.** | **Once for the entire SQL statement.** |
| **Access to affected rows** | **Has access to the values of individual rows (using :NEW and :OLD pseudorecords).** | **Does not have access to the values of individual rows.** |
| **Use Case** | **Used when you need to perform actions on each row (e.g., validation, auditing per row).** | **Used for actions that apply to the statement as a whole (e.g., logging, counting affected rows).** |
| **Syntax** | **FOR EACH ROW keyword is used.** | **No FOR EACH ROW keyword is used.** |

**2)** **CREATE OR REPLACE TRIGGER salary\_check\_trigger**

**BEFORE UPDATE ON EMP**

**FOR EACH ROW**

**BEGIN**

**-- Check if the new salary is below 5000**

**IF :NEW.sal < 5000 THEN**

**-- Raise an error if the salary is below 5000**

**RAISE\_APPLICATION\_ERROR(-20001, 'Salary cannot be below 5000.');**

**END IF;**

**END salary\_check\_trigger;**