Program 1

Aim: Write a SQL queries to demonstrate DDL & DML commands with Master Child Table's relationships.
Procedure:
When dealing with master-child table relationships in a database, you'll often use Data Definition Language (DDL) commands to define the schema and Data Manipulation Language (DML) commands to manipulate the data. Here's an example in SQL for creating a master-child relationship and manipulating the data.

DDL Commands

step1: Create the customer Table (Master Table)

Step2: Create the Orders Table (Child Table)

In this schema, CustomerID in the Orders table is a foreign key that references CustomerID in the Customers table, establishing a master-child relationship.

DML Commands

step1: Insert Data into Customers Table
step2: Insert Data into Orders Table

Step3: **Query Data from Customers and Orders**

To see all orders with customer details:

Step4: **Update Data in Orders Table**

To update the amount for a specific order:

Step5: **Delete Data from Orders Table**

To delete an order:

Program 2
Aim: Write a SQL queries to perform SELECT, PROJECT, SORTING and SET OPERATORS in a table's.

Procedure:

In relational databases, operations such as selection, projection, sorting, and set operations are fundamental for querying and manipulating data.

Selection (σ):

- **Definition**: The selection operation is used to retrieve rows from a table that satisfy a given condition.
- **Syntax**: σ_condition(T)
- **Example**: σ_age > 25(Employees) selects all employees older than 25.

Projection (π):

- **Definition**: The projection operation is used to retrieve specific columns from a table, effectively creating a subset of the attributes.
- **Syntax**: π_columns(T)
- **Example**: π_name, salary(Employees) retrieves only the name and salary columns from the Employees table.

Sorting (ORDER BY):

- **Definition:** The sorting operation is used to arrange the result set of a query in a specific order, either ascending or descending.
- **Syntax:** SELECT columns FROM table_name ORDER BY column_name ASC | DESC;
- **Example:** SELECT name, salary FROM Employees ORDER BY salary DESC;

Set Operators

Set operators are used to combine the results of multiple SELECT statements. The most common set operators are:

UNION

- **Definition:** The UNION operator combines the result sets of two or more SELECT statements, eliminating duplicate rows.
- **Syntax:** SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;
- **Example:** SELECT department FROM Employees UNION SELECT department FROM Customers;

UNION ALL

- **Definition:** The UNION ALL operator combines the result sets of two or more SELECT statements, including duplicate rows.
- **Syntax:** SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;
- **Example:** SELECT name FROM Employees UNION ALL SELECT name FROM Customers;

INTERSECT

- **Definition:** The INTERSECT operator returns the rows that are common to the result sets of two or more SELECT statements.
- **Syntax:** SELECT column_name(s) FROM table1 INTERSECT SELECT column_name(s) FROM table2;
- **Example:** SELECT name FROM Employees INTERSECT SELECT name FROM Customers;

EXCEPT (or MINUS)

- **Definition:** The EXCEPT (or MINUS) operator returns the rows that are in the result set of the first SELECT statement but not in the result set of the second SELECT statement.
- **Syntax:** SELECT column_name(s) FROM table1 EXCEPT SELECT column_name(s) FROM table2;
- **Example:** SELECT name FROM Employees EXCEPT SELECT name FROM Customers;

Program 3
Aim: Write a SQL program to demonstrate AGGREGATE functions.

Procedure:

Aggregate functions are used in database management and data analysis to perform a calculation on a set of values and return a single result.

SUM()

- **Purpose**: Calculates the total sum of a numeric column.
- **Example**: If you have a table of sales data, you can use SUM() to find the total sales.

COUNT()

- **Purpose**: Counts the number of rows in a dataset.
- **Example**: To count how many orders were placed.

AVG()

- **Purpose**: Computes the average value of a numeric column.
- **Example**: To find the average salary of employees.

MAX()

- **Purpose**: Returns the maximum value in a numeric column.
- **Example**: To find the highest price of products in inventory.

MIN()

- **Purpose**: Returns the minimum value in a numeric column.
- **Example**: To find the lowest price of products in inventory.

GROUP CONCAT() (MySQL) / STRING_AGG() (PostgreSQL, SQL Server)

- **Purpose**: Concatenates values from multiple rows into a single string.
- **Example**: To get a comma-separated list of all customer names.

These aggregate functions are very useful for summarizing and analyzing data, providing insights into patterns and trends.

Program 4
Aim: Write SQL queries for implementing SQL JOINS with given Tables.

- Inner join(=,>,<)
- Outer join(left outer, right outer, full outer)

Procedure:

Inner Join

An INNER JOIN returns only the rows that have matching values in both tables.

Example Tables:

- Employees (EmployeeID, Name, DepartmentID)
- Departments (DepartmentID, DepartmentName)

Left Outer Join

A LEFT OUTER JOIN (or simply LEFT JOIN) returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.

Right Outer Join

A RIGHT OUTER JOIN (or simply RIGHT JOIN) returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.

Full Outer Join

A FULL OUTER JOIN returns all rows when there is a match in one of the tables. It returns NULL for columns from the table where there is no match.

Cross Join

A CROSS JOIN returns the Cartesian product of the two tables, i.e., it returns all possible combinations of rows.

Self Join

A SELF JOIN is a regular join but the table is joined with itself.

Program 5
Aim: Create types Locations, Person, Qualification and Create An Object Type Department which Contains Information About An Employee with Above Types

Procedure:

Step 1: Define the Location Type

1. **Create a Location type**:
    - This type will capture information about geographical details.
    - Include properties such as city, state, country, and an optional postalCode

Step 2: Define the Person Type

2. **Create a Person type**:
    - This type will capture personal information.
    - Include properties such as firstName, lastName, dateOfBirth, gender, email, and an optional phoneNumber.

Step 3: Define the Qualification Type

3. **Create a Qualification type**:
    - This type will capture educational qualifications.
    - Include properties such as degree, institution, yearOfGraduation, and an optional major.

Step 4: Define the Department Type

4. **Create a Department type**:
    - This type will encapsulate all the relevant information about an employee.
    - Include properties such as employeeId, person, qualification, position, location, hireDate, and departmentName.
    - Use the previously defined types (Location, Person, and Qualification) as properties.

Step 5: Create an Example Object of Type Department

5. **Create an object of type Department**:
    - Use the Department type to instantiate an example object with sample data.

Summary

- **Define Types**: Start by defining Location, Person, and Qualification types.
- **Create Department Type**: Use these types to create the Department type.
- **Instantiate Object**: Create an example object of type Department with sample data to ensure the structure is correct.

Program 6
Aim: Demonstrate The use of V ARRAYS in SQL.
Procedure:

Virtual arrays (or V arrays) aren't a standard feature of SQL. However, different SQL databases have different ways to work with arrays or similar constructs. I'll demonstrate how you might handle arrays or similar data structures in a few popular SQL systems.

PostgreSQL

PostgreSQL supports arrays natively. You can define an array column and use array functions to work with arrays.

MySQL

MySQL doesn’t have native array support, but you can use JSON data types to mimic array behavior.

Oracle

Oracle SQL provides a way to work with arrays through its VARRAY data type, but it is less commonly used in everyday SQL queries compared to other methods.

Different SQL databases have different methods for handling arrays:

- **PostgreSQL**: Native array support.
- **MySQL**: JSON arrays.
- **Oracle**: VARRAY types.

Program7

Aim: Write a Pl/SQL Program To Count the number of records using CURSORS.

Procedure:

To count the number of records using cursors in a database, you need to follow a series of steps. Below is a general approach for doing this in SQL, using SQL Server as an example, but the principles can be adapted to other database systems like Oracle, MySQL, or PostgreSQL.

SQL Server

1. **Declare and Open the Cursor:** First, you need to declare and open the cursor to iterate through the records.
2. **Fetch Records:** Use a loop to fetch records one by one and count them.

Close and Deallocate the Cursor: Once you're done, close and deallocate the cursor to free up resources.

Oracle

In Oracle, the procedure is slightly different. Here's how you can do it:

1. **Declare and Open the Cursor:**

MySQL

MySQL does not support cursors for counting rows directly in a procedure in a straightforward way like SQL Server or Oracle. Typically, you'd use a different approach. However, if you still want to use a cursor for some reason:

1. **Declare and Open the Cursor:**

2. Call the Procedure:

Important Points

- Using cursors to count records is generally less efficient than using aggregate functions like COUNT(), but can be necessary if you need to process records one by one.
- Always ensure that cursors are properly closed and deallocated to avoid resource leaks.
- In many cases, a simple SELECT COUNT(*) FROM YourTable is sufficient and more efficient if you only need to count records without additional processing.

Program8

Aim: Write a PL/SQL code to calculate the total salary of first N records of employee table. The value of N is passed to CURSOR AS PARAMETER

Procedure:

To calculate the total salary of the first n records from an employee table using a cursor in PL/SQL, you'll need to follow these steps:

1. **Declare the cursor** to select the desired number of rows from the employee table.
2. **Open the cursor** and fetch the rows.
3. **Accumulate the salaries** into a total.
4. **Close the cursor**.

Explanation:

1. **Variable Declarations:**
    - total_salary to store the accumulated salary.
    - emp_salary to temporarily hold the salary value fetched from the cursor.
    - v_count to keep track of the number of records processed.
    - n to specify the number of records to process.
2. **Cursor Definition:**
    - The emp_cursor selects salaries from the employee table. Modify the ORDER BY clause if you need a specific order.
3. **Cursor Operations:**
    - Open the cursor with OPEN emp_cursor.
    - Fetch records in a loop. Exit the loop when no more records are available or when the counter v_count reaches n.
    - Accumulate the salaries and increment the counter.
4. **Closure and Output:**
    - Close the cursor with CLOSE emp_cursor.
    - Output the total salary using DBMS_OUTPUT.PUT_LINE.

This PL/SQL block assumes that you have an employee table with at least the columns salary and employee_id. Adjust the table and column names as necessary for your specific schema.

Program 9

Aim: Write a PL/SQL program to print the Electricity Bill using cursors in the format given below. Electricity bill for the month of…….

……-

Meter no	category A/C/I/D
Consumer Name	Address Current
Reading	Previous reading
Billed units	unit cost
Arrears Total	Amount Payable
Last Date without fine	Last Date with fine

Procedure:

Assuming you have a table named electricity_bills with relevant columns like bill_id, customer_name, amount_due, and due_date, here's a step-by-step example of how to use a cursor to retrieve and print the electricity bill details.

1. Create the Table (if not already present)

First, make sure you have a table to work with. Here's an example of how to create such a table:

2. Insert Sample Data (for demonstration)

3. Use PL/SQL Block with Cursor to Print Bill

Here's a PL/SQL block that uses a cursor to fetch and print the bill details:

Explanation:

1. **Cursor Declaration**: CURSOR bill_cursor IS SELECT ... FROM electricity_bills; declares a cursor to select all columns from the electricity_bills table.
2. **Record Type**: bill_record bill_cursor%ROWTYPE; declares a record variable that can hold the row fetched by the cursor.
3. **Opening the Cursor**: OPEN bill_cursor; opens the cursor.
4. **Fetching Data**: FETCH bill_cursor INTO bill_record; fetches the data into the record variable.
5. **Printing Data**: DBMS_OUTPUT.PUT_LINE is used to print the details.
6. **Loop Control**: EXIT WHEN bill_cursor%NOTFOUND; exits the loop when no more rows are fetched.
7. **Closing the Cursor**: CLOSE bill_cursor; closes the cursor once all rows have been processed.

Program 10

Aim: Design PL/SQL program to calculate ranks for EMCET/ICET (use correct rules) using CURSORS and generate SQL *PLUS Report for it with ranks and hall ticket numbers.

Procedure:

To calculate ranks for EAMCET (Engineering Agricultural and Medical Common Entrance Test) or ICET (Integrated Common Entrance Test) results using PL/SQL and cursors, you can follow these steps:

1. **Define the Table Structure**: Assume you have a table named test_results with columns for student ID, score, and other relevant information.
2. **Create a Procedure to Calculate Ranks**: Use PL/SQL to create a procedure that will calculate ranks based on scores.

Here's a step-by-step guide with an example:

1. Table Structure

Let's assume you have a table test_results with the following structure:

2. Create a Procedure to Calculate Ranks

In this example, we'll use PL/SQL cursors to calculate and update ranks based on scores.

Explanation

1. **Cursor Declaration**: The cursor c_scores selects all records from test_results ordered by score in descending order. This ordering ensures that we calculate ranks from highest to lowest scores.
2. **Variables**:
    - v_rank: Tracks the current rank.
    - v_prev_score: Keeps track of the score of the previous student to handle ties.
    - v_prev_rank: Keeps track of the rank to be assigned for tied scores.
3. **Loop Through Cursor**:
    - For each student record, check if the current score is different from the previous score. If so, update v_prev_rank to the current rank.
    - Update the rank in the test_results table for each student.
4. **Commit**: After updating all records, commit the changes to the database.

Program 11

Aim: Write a code in PL/SQL to implement a Trigger that records user activity (inserts, updates, deletes) in an audit log for a given set of tables.

Procedure:
To implement a trigger using PL/SQL in Oracle, you need to follow these steps:

1. **Understand the Trigger Types**:
    - **BEFORE Trigger**: Executes before an insert, update, or delete operation on a table.
    - **AFTER Trigger**: Executes after an insert, update, or delete operation on a table.
    - **INSTEAD OF Trigger**: Executes in place of an insert, update, or delete operation (commonly used with views).
2. **Define the Trigger**: You'll specify the trigger name, the triggering event (INSERT, UPDATE, DELETE), and the timing (BEFORE, AFTER, INSTEAD OF). You also write the PL/SQL block that will be executed.

Here's a simple example of how to create a trigger in PL/SQL. This trigger logs changes made to a table into an audit table.

Example Scenario

Let's assume we have two tables:

1. employees - The main table where employee data is stored.
2. audit_log - An audit table to keep track of changes made to the employees table.

employees Table
audit_log Table

Trigger Implementation

1. **Create the Trigger**

   Here's how to create a trigger that logs changes (insertions, updates, deletions) to the audit_log table whenever changes occur in the employees table.

2. Explanation of the Trigger Code

- AFTER INSERT OR UPDATE OR DELETE ON employees: Specifies that the trigger should fire after an INSERT, UPDATE, or DELETE operation on the employees table.
- FOR EACH ROW: Indicates that the trigger should execute for each row affected by the DML operation.
- DECLARE: This section is used for declaring local variables.
- BEGIN ... END;: Contains the PL/SQL block that will be executed when the trigger fires.
- :NEW and :OLD: These are pseudo-records that represent the new and old values of the row being inserted, updated, or deleted.

3. Test the Trigger

- To test the trigger, perform some operations on the employees table and then query the audit_log table to ensure that changes are being recorded correctly.