plsql6 corrected

5. Cursors: (All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor).

Code:

```
-- 1️⃣ Drop tables if they exist
DROP TABLE IF EXISTS O_RollCall;
DROP TABLE IF EXISTS N_RollCall;


-- 2️⃣ Create tables
CREATE TABLE O_RollCall (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50)
);


CREATE TABLE N_RollCall (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50)
);


-- 3️⃣ Insert sample data
INSERT INTO O_RollCall (roll_no, name) VALUES
(101, 'Alice'),
(102, 'Bob');


INSERT INTO N_RollCall (roll_no, name) VALUES
(102, 'Bob'),
(103, 'Charlie'),
(104, 'David');
```

```
-- 4️⃣ Merge using a “parameterized-like” cursor
DO
$$
DECLARE
    v_min_roll_no INT := 0;  -- parameter for the cursor simulation
    r_rec RECORD;
    cur_n_rollcall REFCURSOR;  -- explicit cursor
BEGIN
    -- Open cursor with a parameter
    OPEN cur_n_rollcall FOR
        SELECT roll_no, name
        FROM N_RollCall
        WHERE roll_no >= v_min_roll_no;


    LOOP
        FETCH cur_n_rollcall INTO r_rec;
        EXIT WHEN NOT FOUND;


        -- Insert into O_RollCall if not exists
        IF NOT EXISTS (SELECT 1 FROM O_RollCall WHERE roll_no = r_rec.roll_no) THEN
            INSERT INTO O_RollCall (roll_no, name)
            VALUES (r_rec.roll_no, r_rec.name);
        END IF;
    END LOOP;


    CLOSE cur_n_rollcall;


    RAISE NOTICE 'Merge Completed Successfully!';
END;
$$;


-- 5️⃣ Verify merged table
SELECT * FROM O_RollCall;
```

correct plsql7

```
-- ============================================================
-- 🧩 DROP OLD TABLES IF THEY EXIST
-- ============================================================
DROP TABLE IF EXISTS library_audit;
DROP TABLE IF EXISTS library;

-- ============================================================
-- 🏛️ CREATE MAIN TABLE: library
-- ============================================================
CREATE TABLE library (
    bno INT PRIMARY KEY,
    bname VARCHAR(40),
    author VARCHAR(20),
    allowed_days INT
);

-- ============================================================
-- 🪶 CREATE AUDIT TABLE: library_audit
-- ============================================================
CREATE TABLE library_audit (
    audit_id SERIAL PRIMARY KEY,
    bno INT,
    old_allowed_days INT,
    new_allowed_days INT,
    action VARCHAR(10),
    action_time TIMESTAMP DEFAULT NOW()
);
```

```
-- ============================================================
-- 🧃 INSERT SAMPLE DATA
-- ============================================================
INSERT INTO library VALUES
(1, 'Database Systems', 'Tom', 10),
(2, 'System Programming', 'John', 20),
(3, 'Computer Networks', 'Sara', 18),
(4, 'Agile Project Management', 'Ken', 24),
(5, 'Python for Data Analysis', 'Wes', 12);

-- ============================================================
-- 🔹 1️⃣ ROW-LEVEL BEFORE TRIGGER
-- Runs before each row is updated or deleted
-- Used for validation or logging before actual change
-- ============================================================
CREATE OR REPLACE FUNCTION before_row_func()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'UPDATE' THEN
        RAISE NOTICE 'BEFORE UPDATE (Row-Level): Book % changing allowed_days from % tc
            OLD.bno, OLD.allowed_days, NEW.allowed_days;
    ELSIF TG_OP = 'DELETE' THEN
        RAISE NOTICE 'BEFORE DELETE (Row-Level): Book % with allowed_days % will be del
            OLD.bno, OLD.allowed_days;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_row_trigger
BEFORE UPDATE OR DELETE ON library
FOR EACH ROW
EXECUTE FUNCTION before_row_func();

-- ============================================================
-- 🔹 2️⃣ ROW-LEVEL AFTER TRIGGER
-- Runs after each row is updated or deleted
-- Used for auditing old and new values
-- ============================================================
CREATE OR REPLACE FUNCTION after_row_func()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'UPDATE' THEN
        INSERT INTO library_audit (bno, old_allowed_days, new_allowed_days, action)
        VALUES (OLD.bno, OLD.allowed_days, NEW.allowed_days, 'UPDATE');
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO library_audit (bno, old_allowed_days, new_allowed_days, action)
        VALUES (OLD.bno, OLD.allowed_days, NULL, 'DELETE');
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_row_trigger
AFTER UPDATE OR DELETE ON library
FOR EACH ROW
EXECUTE FUNCTION after_row_func();

-- ============================================================
-- 🔹 3️⃣ STATEMENT-LEVEL BEFORE TRIGGER
-- Runs once before the UPDATE or DELETE statement starts
-- ============================================================
CREATE OR REPLACE FUNCTION before_stmt_func()
RETURNS TRIGGER AS $$
```

```
BEGIN
    RAISE NOTICE 'BEFORE STATEMENT (Statement-Level) Trigger Fired for operation: %', T
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_stmt_trigger
BEFORE UPDATE OR DELETE ON library
FOR EACH STATEMENT
EXECUTE FUNCTION before_stmt_func();

-- ============================================================
-- 🔹 4️⃣ STATEMENT-LEVEL AFTER TRIGGER
-- Runs once after the UPDATE or DELETE statement completes
-- ============================================================
CREATE OR REPLACE FUNCTION after_stmt_func()
RETURNS TRIGGER AS $$
BEGIN
    RAISE NOTICE 'AFTER STATEMENT (Statement-Level) Trigger Fired for operation: %', TG
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_stmt_trigger
AFTER UPDATE OR DELETE ON library
FOR EACH STATEMENT
EXECUTE FUNCTION after_stmt_func();

-- ============================================================
-- 📜 TEST THE TRIGGERS
-- ============================================================
-- UPDATE a record (fires all 4 triggers)
UPDATE library SET allowed_days = 15 WHERE bno = 1;

-- DELETE a record (fires all 4 triggers)
DELETE FROM library WHERE bno = 2;

-- ============================================================
-- 🔍 VIEW RESULTS
-- ============================================================
SELECT * FROM library;         -- Current library data
SELECT * FROM library_audit;   -- Audit history
```

mongo1.txt

mongo1

```
// ---------------------------------------------------
// PIZZA DELIVERY SYSTEM - MONGODB CRUD OPERATIONS
// ---------------------------------------------------

// 1️⃣  CREATE / USE DATABASE
use PizzaDeliverySystem;
// Creates a new database called 'PizzaDeliverySystem' if it doesn't exist
// or switches to it if it already exists

// ---------------------------------------------------
// 2️⃣  CREATE COLLECTIONS & INSERT DOCUMENTS
// ---------------------------------------------------

// Insert multiple pizza documents into the 'pizzas' collection
db.pizzas.insertMany([
  { _id: 1, name: "Margherita", size: "Medium", price: 300, ingredients: ["Cheese", "To
  { _id: 2, name: "Pepperoni", size: "Large", price: 450, ingredients: ["Cheese", "Pepp
  { _id: 3, name: "Farmhouse", size: "Medium", price: 400, ingredients: ["Cheese", "Veg
]);
// insertMany → Adds multiple documents at once
// _id → unique identifier for each pizza
// ingredients → array field listing pizza ingredients
// tags → array field for categories/labels

// Insert one pizza document
db.pizzas.insertOne({
  _id: 4,
  name: "Devinshire",
  size: "Medium",
  price: 230,
  ingredients: ["Basil", "Tomato"],
  tags: ["Vegetarian", "American"]
});
```

```
// insertOne → Adds a single document to the collection

// Insert multiple customer documents into 'customers' collection
db.customers.insertMany([
  { _id: 1, name: "Amit", phone: "9001112233", address: "Sector 10, City Center", favor
  { _id: 2, name: "Sara", phone: "9003445566", address: "Lakeview Apartments", favorite
]);
// favorite_pizzas → array field storing customer's favorite pizzas

// Insert multiple orders into 'orders' collection
db.orders.insertMany([
  { _id: 1001, customer_id: 1, pizza_ids: [1, 3], total: 700, order_date: ISODate("2025
  { _id: 1002, customer_id: 2, pizza_ids: [2], total: 450, order_date: ISODate("2025-07
]);
// pizza_ids → array field storing ordered pizza IDs
// order_date → ISODate format for date and time

// ------------------------------------------------
// 3️⃣  READ OPERATIONS (QUERY, LOGICAL OPERATORS, SORT, LIMIT, PROJECTION)
// ------------------------------------------------

// View all pizzas with pretty formatting
db.pizzas.find().pretty();
// pretty() → formats JSON output for readability

// View all customers
db.customers.find().pretty();

// View all orders
db.orders.find().pretty();

// Find pizzas with price greater than 350
db.pizzas.find({ price: { $gt: 350 } });
// $gt → comparison operator: greater than

// Find orders with total greater than or equal to 700
db.orders.find({ total: { $gte: 700 } });
// $gte → greater than or equal

// Find Medium pizzas with price > 300 using $and logical operator
db.pizzas.find({
  $and: [
    { size: "Medium" },
    { price: { $gt: 300 } }
  ]
});
// $and → all conditions must be true

// Find customers named Amit OR Sara using $or
db.customers.find({
  $or: [
    { name: "Amit" },
    { name: "Sara" }
  ]
});
// $or → at least one condition must be true

// Find pizzas with price not greater than 400 using $not
db.pizzas.find({ price: { $not: { $gt: 400 } } });
// $not → negates a condition (price ≤ 400)

// Find customers where neither condition is true using $nor
db.customers.find({
  $nor: [
```

```
    { name: "Amit" },
    { phone: "9003445566" }
  ]
});
// $nor → none of the conditions should be true

// Find pizzas with tags either Spicy or Loaded using $in
db.pizzas.find({ tags: { $in: ["Spicy", "Loaded"] } });

// Find pizzas excluding Farmhouse and Pepperoni using $nin
db.pizzas.find({ name: { $nin: ["Farmhouse", "Pepperoni"] } });

// ------------------------------------------------
// SORT, LIMIT, PROJECTION examples
// ------------------------------------------------

// Show pizza names and prices only, sorted by price descending, limit 3
db.pizzas.find({}, { name: 1, price: 1, _id: 0 }).sort({ price: -1 }).limit(3);
// {} → no filter, returns all documents
// Projection → { name: 1, price: 1, _id: 0 } → include name & price, exclude _id
// sort({ price: -1 }) → descending
// limit(3) → return only 3 documents

// Show all orders sorted by total ascending, limit 5
db.orders.find().sort({ total: 1 }).limit(5);

// Show customer names only, sorted alphabetically
db.customers.find({}, { name: 1, _id: 0 }).sort({ name: 1 });

// Combined example: Medium pizzas, only name & price, sorted by price descending, limi
db.pizzas.find({ size: "Medium" }, { name: 1, price: 1, _id: 0 }).sort({ price: -1 }).l

// ------------------------------------------------
// 4️⃣  UPDATE OPERATIONS
// ------------------------------------------------

// Update single field: change Amit's phone
db.customers.updateOne({ name: "Amit" }, { $set: { phone: "9000000000" } });
// $set → updates or adds a field

// Increment numeric field: increase order total by 50
db.orders.updateOne({ _id: 1002 }, { $inc: { total: 50 } });
// $inc → increment numeric field by specified value

// Add element to array: add 'BestSeller' tag to Margherita
db.pizzas.updateOne({ name: "Margherita" }, { $push: { tags: "BestSeller" } });
// $push → adds element to array

// Add multiple elements to array using $each
db.pizzas.updateOne({ name: "Farmhouse" }, { $push: { tags: { $each: ["Offers", "Recomm
// $each → adds multiple elements at once

// Add unique element to array using $addToSet
db.pizzas.updateOne({ name: "Pepperoni" }, { $addToSet: { tags: "Classic" } });
// $addToSet → adds element only if it doesn't exist

// Rename a field: phone → contact
db.customers.updateOne({ name: "Amit" }, { $rename: { phone: "contact" } });
// $rename → renames a field

// Multiply numeric field: increase price by 10%
db.pizzas.updateOne({ name: "Pepperoni" }, { $mul: { price: 1.1 } });
// $mul → multiply numeric field
```

```
// Remove specific field: remove Sara's address
db.customers.updateOne({ name: "Sara" }, { $unset: { address: "" } });
// $unset → deletes a field from a document

// ------------------------------------------------
// 5️⃣  SAVE METHOD
// ------------------------------------------------

// Update existing document using save
var pizza = db.pizzas.findOne({ name: "Margherita" });
pizza.price = 320;
db.pizzas.save(pizza);
// save() → replaces the document if _id exists; otherwise inserts new

// Insert new document using save
db.pizzas.save({
  _id: 5,
  name: "Veggie Delight",
  size: "Small",
  price: 250,
  ingredients: ["Cheese", "Bell Pepper", "Onion"],
  tags: ["Vegetarian", "Healthy"]
});

// ------------------------------------------------
// 6️⃣  DELETE OPERATIONS
// ------------------------------------------------

// Delete one document: remove Devinshire pizza
db.pizzas.deleteOne({ name: "Devinshire" });

// Delete many documents: remove pizzas with price < 300
db.pizzas.deleteMany({ price: { $lt: 300 } });

// Remove element from array: remove Margherita from Amit's favorite pizzas
db.customers.updateOne({ name: "Amit" }, { $pull: { favorite_pizzas: "Margherita" } });
// $pull → removes matching elements from an array

// Remove last element from pizza_ids array in order 1001
db.orders.updateOne({ _id: 1001 }, { $pop: { pizza_ids: 1 } });
// $pop → removes first (-1) or last (1) element from array

// ------------------------------------------------
// 7️⃣  BONUS - AGGREGATION EXAMPLE
// ------------------------------------------------

// Aggregate orders by customer, count total orders and sum total_amount, sort descendi
db.orders.aggregate([
  { $group: { _id: "$customer_id", total_orders: { $sum: 1 }, total_amount: { $sum: "$t
  { $sort: { total_amount: -1 } }
]);
// $group → groups documents by _id (customer_id), calculates aggregates
// $sum → sums values or counts documents
// $sort → sorts by total_amount descending
```

mongo 2.txt

mongo 2

```
//---------------------------------------------------------
// PIZZA DELIVERY SYSTEM - AGGREGATION & INDEXING
//---------------------------------------------------------

// 1️⃣ USE DATABASE
use PizzaDel2;

//---------------------------------------------------------
// 2️⃣ INSERT SAMPLE DATA
//---------------------------------------------------------
db.pizzas.insertMany([
  { _id: 1, name: "Margherita", size: "Medium", price: 300, tags: ["Vegetarian", "Class
  { _id: 2, name: "Pepperoni", size: "Large", price: 450, tags: ["Non-Vegetarian", "Spi
  { _id: 3, name: "Farmhouse", size: "Medium", price: 400, tags: ["Vegetarian", "Loadec
  { _id: 4, name: "Devinshire", size: "Medium", price: 230, tags: ["Vegetarian", "Ameri
  { _id: 5, name: "BBQ Chicken", size: "Large", price: 480, tags: ["Non-Vegetarian", "S
]);

db.customers.insertMany([
  { _id: 1, name: "Amit", favorite_pizzas: ["Margherita"] },
```

```
  { _id: 2, name: "Sara", favorite_pizzas: ["Pepperoni"] },
  { _id: 3, name: "Pierre", favorite_pizzas: ["Farmhouse", "BBQ Chicken"] }
]);

db.orders.insertMany([
  { _id: 1001, customer_id: 1, pizza_ids: [1,3], total: 700, status: "Delivered" },
  { _id: 1002, customer_id: 2, pizza_ids: [2], total: 450, status: "Preparing" },
  { _id: 1003, customer_id: 3, pizza_ids: [3,5], total: 880, status: "Delivered" },
  { _id: 1004, customer_id: 1, pizza_ids: [4], total: 230, status: "Cancelled" }
]);

//------------------------------------------------------------
// 3️⃣ AGGREGATION EXAMPLES
//------------------------------------------------------------

// Total sales per customer
db.orders.aggregate([
  { $group: { _id: "$customer_id", totalSpent: { $sum: "$total" } } }
]);

// Count pizzas by size
db.pizzas.aggregate([
  { $group: { _id: "$size", count: { $sum: 1 } } }
]);

// Total revenue per order status
db.orders.aggregate([
  { $group: { _id: "$status", totalRevenue: { $sum: "$total" } } }
]);

// Top 2 most expensive pizzas
db.pizzas.aggregate([
  { $sort: { price: -1 } },
  { $limit: 2 }
]);

// Count vegetarian pizzas
db.pizzas.aggregate([
  { $match: { tags: "Vegetarian" } },
  { $count: "vegetarianPizzas" }
]);

//------------------------------------------------------------
// 4️⃣ INDEXING EXAMPLES
//------------------------------------------------------------

// Text index on name and tags
db.pizzas.createIndex({ name: "text", tags: "text" });
db.pizzas.find({ $text: { $search: "Vegetarian" } });

// Unique index on customer name
db.customers.createIndex({ name: 1 }, { unique: true });
db.customers.insertOne({_id :4 , name: "Sara" });


// Single field index on pizza price
db.pizzas.createIndex({ price: 1 });
db.pizzas.find({ price: 400 });
db.pizzas.find().sort({ price: 1 });


// Compound index on pizza size and price
db.pizzas.createIndex({ size: 1, price: -1 });
db.pizzas.find({ size: "Medium" }).sort({ price: -1 });
```

```
// Check indexes
db.pizzas.getIndexes();
db.customers.getIndexes();
db.orders.getIndexes();
```

mongo 3.txt

```
mongo 3
```

```
//---------------------------------------------------------
// MAPREDUCE EXAMPLE - BILL COLLECTION
//---------------------------------------------------------
```

```
// 1️⃣ Use Database
use bill;

// 2️⃣ Insert Sample Data (if not already present)
db.pay.drop(); // Clear existing data

db.pay.insertMany([
  { Cust_ID: "A123", Product: "Milk", Amount: 40, Status: "P" },
  { Cust_ID: "A123", Product: "Parle_G", Amount: 50, Status: "NP" },
  { Cust_ID: "A123", Product: "Lays Chips", Amount: 40, Status: "P" },
  { Cust_ID: "B123", Product: "Mentos", Amount: 10, Status: "P" },
  { Cust_ID: "B123", Product: "Maggie", Amount: 60, Status: "NP" }
]);

print("✅ Sample Data:");
db.pay.find().pretty();

//------------------------------------------------------------
// 3️⃣ MAPREDUCE - Calculate Total Pending Amount per Customer
//------------------------------------------------------------

// Map Function: emits customer ID and amount only for unpaid items
var mapFunc = function() {
    if (this.Status === "NP") {        // Only consider Not Paid items
        emit(this.Cust_ID, this.Amount);
    }
};

// Reduce Function: sums all amounts for each customer
var reduceFunc = function(keyCustID, valueAmounts) {
    return Array.sum(valueAmounts);
};

// Optional Finalize Function: add descriptive info
var finalizeFunc = function(keyCustID, reducedVal) {
    return { totalPending: reducedVal, note: "Pending payment" };
};

// Execute MapReduce
db.pay.mapReduce(
    mapFunc,
    reduceFunc,
    {
        out: "PendingAmounts",  // Output collection
        finalize: finalizeFunc  // Optional finalize for better output
    }
);

print("🎯 Total Pending Amount per Customer:");
db.PendingAmounts.find().pretty();

//------------------------------------------------------------
// 4️⃣ MAPREDUCE - Calculate Total Amount (All Payments) per Customer
//------------------------------------------------------------

// Map Function: emits customer ID and amount for all items
var mapFuncAll = function() {
    emit(this.Cust_ID, this.Amount);
};

// Reduce Function: sums all amounts for each customer
var reduceFuncAll = function(keyCustID, valueAmounts) {
    return Array.sum(valueAmounts);
```

```
};

// Execute MapReduce for all payments
db.pay.mapReduce(
    mapFuncAll,
    reduceFuncAll,
    { out: "TotalAmounts" }  // Output collection
);

print("🎯 Total Amount per Customer (All Payments):");
db.TotalAmounts.find().pretty();
```

sql 3.txt

```
sql 3
```

```
CREATE DATABASE IF NOT EXISTS office_db;
USE office_db;
CREATE TABLE IF NOT EXISTS dept (did INT AUTO_INCREMENT PRIMARY KEY, dname VARCHAR(30),
CREATE TABLE IF NOT EXISTS emp (eid INT AUTO_INCREMENT PRIMARY KEY, ename VARCHAR(30),
INSERT INTO dept (dname, city) VALUES ('HR','Pune'),('IT','Mumbai'),('Admin','Delhi'),(
INSERT INTO emp (ename, salary, gender, did) VALUES ('Harshada',55000,'F',2),('Sarvesh'
SELECT * FROM dept;
SELECT * FROM emp;
SELECT e.ename, d.dname, d.city FROM emp e INNER JOIN dept d ON e.did=d.did;
SELECT e.ename, d.dname FROM emp e LEFT JOIN dept d ON e.did=d.did;
SELECT e.ename, d.dname FROM emp e RIGHT JOIN dept d ON e.did=d.did;
SELECT e.ename, d.dname FROM emp e CROSS JOIN dept d;
SELECT ename, salary FROM emp WHERE salary>(SELECT AVG(salary) FROM emp);
SELECT ename, salary FROM emp WHERE salary=(SELECT MAX(salary) FROM emp);
SELECT dname FROM dept WHERE did=(SELECT did FROM emp GROUP BY did ORDER BY COUNT(*) DE
SELECT e.ename, e.salary FROM emp e WHERE e.salary>(SELECT AVG(salary) FROM emp WHERE d
CREATE VIEW emp_view AS SELECT e.ename, e.salary, d.dname, d.city FROM emp e LEFT JOIN
SELECT * FROM emp_view;
UPDATE emp SET salary=salary+5000 WHERE ename='Riya';
SELECT * FROM emp_view;
DROP VIEW emp_view;
```

sql2.txt

sql2

```
SHOW DATABASES;
CREATE DATABASE employee;
USE employee;
CREATE TABLE emp_details (emp_no INT AUTO_INCREMENT PRIMARY KEY, emp_name VARCHAR(30),
DESC emp_details;
INSERT INTO emp_details (emp_name, emp_gender, emp_sal, emp_dept) VALUES ('Ram','M',300
SELECT * FROM emp_details;
CREATE TABLE emp_info AS SELECT emp_no, emp_name, emp_gender FROM emp_details;
SELECT * FROM emp_info;
TRUNCATE TABLE emp_info;
DROP TABLE emp_info;
CREATE VIEW emp_view1 AS SELECT * FROM emp_details;
CREATE VIEW emp_view2 AS SELECT * FROM emp_details WHERE emp_dept='Designing';
SELECT * FROM emp_view1;
SELECT * FROM emp_view2;
UPDATE emp_details SET emp_dept='Coding' WHERE emp_name='Mohan';
DROP VIEW emp_view1;
DROP VIEW emp_view2;
CREATE INDEX emp_ind ON emp_details(emp_no, emp_name);
SHOW INDEX FROM emp_details;
SELECT * FROM emp_details;
SELECT emp_name, emp_dept FROM emp_details WHERE emp_dept='Coding';
SELECT emp_name, emp_sal FROM emp_details WHERE emp_sal>300000;
UPDATE emp_details SET emp_sal=emp_sal*1.10 WHERE emp_dept='Designing';
DELETE FROM emp_details WHERE emp_name='Om';
SELECT * FROM emp_details ORDER BY emp_sal DESC;
SELECT emp_dept, COUNT(*) AS total_emp FROM emp_details GROUP BY emp_dept;
SELECT emp_dept, AVG(emp_sal) AS avg_salary FROM emp_details GROUP BY emp_dept;
SELECT emp_name, emp_sal FROM emp_details ORDER BY emp_sal DESC LIMIT 1;
ALTER TABLE emp_details ADD emp_city VARCHAR(20);
UPDATE emp_details SET emp_city='Pune' WHERE emp_dept='Coding';
UPDATE emp_details SET emp_city='Mumbai' WHERE emp_dept='Designing';
UPDATE emp_details SET emp_city='Delhi' WHERE emp_dept='Management';
SELECT emp_name, emp_dept, emp_city FROM emp_details;
```

plsql 4.txt

```
plsql 4



DROP TABLE IF EXISTS fine;
DROP TABLE IF EXISTS borrower;

CREATE TABLE borrower(
    roll_no INT,
    name TEXT,
    date_of_issue DATE,
    name_of_book TEXT,
    status CHAR(1)
);

CREATE TABLE fine(
    roll_no INT,
    return_date DATE,
    amount NUMERIC(10,2)
);

INSERT INTO borrower VALUES
(101, 'Alice', '2025-10-01', 'Mathematics', 'I'),
(102, 'Bob', '2025-09-15', 'Physics', 'I'),
(103, 'Charlie', '2025-09-20', 'Chemistry', 'I'),
(104, 'David', '2025-10-05', 'Biology', 'I'),
(105, 'Eve', '2025-09-25', 'English', 'I');

DO $$
```

```
DECLARE
    v_roll_no INT := 103;
    v_book TEXT := 'Chemistry';
    v_issue_date DATE;
    v_days INT;
    v_fine NUMERIC := 0;
BEGIN
    SELECT date_of_issue INTO v_issue_date
    FROM borrower
    WHERE roll_no = v_roll_no AND name_of_book = v_book;

    IF NOT FOUND THEN
        RAISE NOTICE 'No record found for roll_no % and book %', v_roll_no, v_book;
        RETURN;
    END IF;

    v_days := CURRENT_DATE - v_issue_date;
    RAISE NOTICE 'Days since issue: %', v_days;

    IF v_days > 30 THEN
        v_fine := (30 * 5) + (v_days - 30) * 50;
    ELSIF v_days >= 15 THEN
        v_fine := v_days * 5;
    ELSE
        v_fine := 0;
    END IF;

    RAISE NOTICE 'Calculated fine: Rs.%', v_fine;

    UPDATE borrower
    SET status = 'R'
    WHERE roll_no = v_roll_no AND name_of_book = v_book;

    IF v_fine > 0 THEN
        INSERT INTO fine VALUES (v_roll_no, CURRENT_DATE, v_fine);
    END IF;

    RAISE NOTICE 'Return and fine process completed successfully!';
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Unexpected error occurred!';
END;
$$ LANGUAGE plpgsql;

SELECT * FROM borrower;
SELECT * FROM fine;
```

plsql 5.txt

plsql 5

```
DROP TABLE IF EXISTS result;
DROP TABLE IF EXISTS stud_marks;

CREATE TABLE stud_marks (
    name VARCHAR(20),
    total_marks INT
);

CREATE TABLE result (
    roll_no SERIAL PRIMARY KEY,
    name VARCHAR(20),
    class VARCHAR(30)
);

INSERT INTO stud_marks VALUES
('Suresh', 995),
('Harish', 865),
('Samarth', 920),
('Mohan', 1000),
('Soham', 745);

INSERT INTO result (name) VALUES
('Suresh'),
('Harish'),
('Samarth'),
('Mohan'),
('Soham');
```

```
CREATE OR REPLACE PROCEDURE proc_grade(p_roll INT)
LANGUAGE plpgsql
AS $$
DECLARE
    v_name VARCHAR(20);
    v_marks INT;
    v_class VARCHAR(25);
BEGIN
    SELECT name INTO v_name FROM result WHERE roll_no = p_roll;
    SELECT total_marks INTO v_marks FROM stud_marks WHERE name = v_name;

    IF v_marks BETWEEN 990 AND 1500 THEN
        v_class := 'Distinction';
    ELSIF v_marks BETWEEN 900 AND 989 THEN
        v_class := 'First Class';
    ELSIF v_marks BETWEEN 825 AND 899 THEN
        v_class := 'Higher Second Class';
    ELSE
        v_class := 'Fail';
    END IF;

    UPDATE result
    SET class = v_class
    WHERE roll_no = p_roll;

    RAISE NOTICE 'Roll No: %, Name: %, Marks: %, Class: %', p_roll, v_name, v_marks, v_

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE NOTICE 'No record found for Roll No: %', p_roll;
END;
$$;

CREATE OR REPLACE FUNCTION func_grade(p_roll INT)
RETURNS VARCHAR
LANGUAGE plpgsql
AS $$
DECLARE
    v_class VARCHAR(25);
BEGIN
    CALL proc_grade(p_roll);
    SELECT class INTO v_class FROM result WHERE roll_no = p_roll;
    RETURN v_class;
END;
$$;

SELECT func_grade(1);
SELECT func_grade(2);
SELECT func_grade(3);
SELECT func_grade(4);
SELECT func_grade(5);

SELECT * FROM result;
```

plsql6_final.txt

```
-- 1️⃣ Drop tables if they exist
DROP TABLE IF EXISTS O_RollCall CASCADE;
DROP TABLE IF EXISTS N_RollCall CASCADE;

-- 2️⃣ Create tables
CREATE TABLE O_RollCall (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50)
);

CREATE TABLE N_RollCall (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50)
);

-- 3️⃣ Insert sample data
INSERT INTO O_RollCall (roll_no, name) VALUES
(101, 'Alice'),
(102, 'Bob');

INSERT INTO N_RollCall (roll_no, name) VALUES
(102, 'Bob'),
(103, 'Charlie'),
(104, 'David');
```

```
-- 4️⃣ Create procedure using parameterized cursor
CREATE OR REPLACE PROCEDURE merge_rollcalls()
LANGUAGE plpgsql
AS $$
DECLARE
    r_row RECORD;           -- record variable to loop through N_RollCall
    r_rec RECORD;           -- record variable to fetch cursor data
    c_merge CURSOR(p_roll_no INT) FOR
        SELECT roll_no, name
        FROM N_RollCall
        WHERE roll_no = p_roll_no;
BEGIN
    -- Loop through all rows of N_RollCall
    FOR r_row IN SELECT roll_no FROM N_RollCall LOOP
        OPEN c_merge(r_row.roll_no);        -- pass parameter
        FETCH c_merge INTO r_rec;

        -- Insert only if not already in O_RollCall
        IF NOT EXISTS (SELECT 1 FROM O_RollCall WHERE roll_no = r_rec.roll_no) THEN
            INSERT INTO O_RollCall (roll_no, name)
            VALUES (r_rec.roll_no, r_rec.name);
            RAISE NOTICE 'Inserted: Roll_no=% | Name=%', r_rec.roll_no, r_rec.name;
        ELSE
            RAISE NOTICE 'Skipped duplicate: Roll_no=% | Name=%', r_rec.roll_no, r_rec.
        END IF;

        CLOSE c_merge;
    END LOOP;

    RAISE NOTICE '🎯 Merge Completed Successfully!';
END;
$$;

-- 5️⃣ Call the procedure
CALL merge_rollcalls();

-- 6️⃣ Verify the merged table
SELECT * FROM O_RollCall ORDER BY roll_no;
```