DBMS Practical

Problem Statement 5 (JOINS & SUBQUERIES USING MYSQL)

Consider Following Schema

Employee (Employee_id, First_name, Last_name , Hire_date, Salary, Job_title, Manager_id, department_id)

Departments(Department_id, Department_name, Manager_id, Location_id)

Locations(Location_id , Street_address , Postal_code, city, state, Country_id)

Manager(Manager_id, Manager_name)

Create the tables with referential integrity.Solve following queries using joins and subqueries.

1. Write a query to find the names (first_name, last_name), the salary of the employees who earn more than the average salary and who works in any of the IT departments.
2. Write a query to find the names (first_name, last_name), the salary of the employees who earn the same salary as the minimum salary for all departments.
3. Write a query to display the employee ID, first name, last names, salary of all employees whose salary is above average for their departments.
4. Write a query to display the department name, manager name, and city.
5. Write a query to display the name (first_name, last_name), hire date, salary of all managers whose experience is more than 15 years.

Creating Table

```
-- Create the Employee table
CREATE TABLE Employee (
    Employee_id INT PRIMARY KEY,
    First_name VARCHAR(50),
    Last_name VARCHAR(50),
    Hire_date DATE,
    Salary DECIMAL(10, 2),
    Job_title VARCHAR(50),
    Manager_id INT,
    department_id INT
);
```

```
-- Create the Departments table
CREATE TABLE Departments (
    Department_id INT PRIMARY KEY,
    Department_name VARCHAR(50),
    Manager_id INT,
    Location_id INT
);
```

```
-- Create the Locations table
CREATE TABLE Locations (
    Location_id INT PRIMARY KEY,
    Street_address VARCHAR(100),
    Postal_code VARCHAR(10),
    City VARCHAR(50),
    State VARCHAR(50),
    Country_id VARCHAR(2)
);
```

```
-- Create the Manager table
CREATE TABLE Manager (
    Manager_id INT PRIMARY KEY,
    Manager_name VARCHAR(100)
);
```

Inserting Data Into Tables

```
-- Insert sample data into the Employee table
INSERT INTO Employee (Employee_id, First_name, Last_name, Hire_date, Salary,
Job_title, Manager_id, department_id)
VALUES
    (1, 'John', 'Doe', '2023-01-15', 60000.00, 'Manager', NULL, 1),
    (2, 'Jane', 'Smith', '2023-02-20', 50000.00, 'Developer', 1, 1),
    (3, 'Michael', 'Johnson', '2023-03-10', 55000.00, 'Developer', 1, 2),
    (4, 'Emily', 'Williams', '2023-04-05', 52000.00, 'Analyst', 1, 3),
    (5, 'David', 'Brown', '2023-05-12', 49000.00, 'Analyst', 1, 3);
```

```
-- Insert sample data into the Departments table
INSERT INTO Departments (Department_id, Department_name, Manager_id,
Location_id)
VALUES
    (1, 'HR', 1, 101),
    (2, 'IT', 2, 102),
    (3, 'Finance', 3, 103);
```

```
-- Insert sample data into the Locations table
INSERT INTO Locations (Location_id, Street_address, Postal_code, City, State,
Country_id)
VALUES
   (101, '123 Main St', '12345', 'New York', 'NY', 'US'),
   (102, '456 Elm St', '67890', 'Los Angeles', 'CA', 'US'),
   (103, '789 Oak St', '98765', 'Chicago', 'IL', 'US');
```

```
-- Insert sample data into the Manager table
INSERT INTO Manager (Manager_id, Manager_name)
VALUES
   (1, 'Alice Johnson');
```

1. Write a query to find the names (first_name, last_name), the salary of the employees who earn more than the average salary and who works in any of the IT departments.

```
SELECT * FROM Employee
WHERE Department_id IN
(SELECT Department_id FROM Departments WHERE Department_name = "IT")
AND Salary > (Select AVG(Salary) FROM Employee);
```

Using Inner Join

```
SELECT First_name, Last_name,Department_name
FROM Employee AS e
INNER JOIN Departments AS d
ON e.Department_id = d.Department_id
WHERE e.Salary > (SELECT AVG(Salary) FROM Employee)
AND d.Department_name = "Engineering";
```

2. Write a query to find the names (first_name, last_name), the salary of the employees who earn the same salary as the minimum salary for all departments.

```
SELECT First_name,Last_name,Salary
FROM Employee
WHERE Salary = (SELECT MIN(Salary) FROM Employee);
```

3. Write a query to display the employee ID, first name, last names, salary of all employees whose salary is above average for their departments.

```
SELECT Employee_id,First_name,Last_name,Salary FROM Employee AS e
WHERE Salary > (SELECT AVG(Salary)
FROM Employee WHERE Department_id = e.Department_id);
```

4. Write a query to display the department name, manager name, and city.

```
SELECT Department_name,Manager_name, City FROM Manager AS m
INNER JOIN Departments AS d ON m.Manager_id = d.Manager_id
INNER JOIN Locations AS l ON d.Location_id = l.Location_id;
```

5. Write a query to display the name (first_name, last_name), hire date, salary of all managers whose experience is more than 15 years.

```
SELECT * FROM Employee WHERE YEAR(Hire_date) < 2008;
```

```
SELECT First_name,Last_name,Hire_date,Salary,
(DATEDIFF(now(), Hire_date)) / 365 Experience
FROM Departments d JOIN Employee e ON (d.Manager_id = e.Employee_id)
WHERE (DATEDIFF(now(),Hire_date))/365 > 15;
```

Problem Statement 8 (DML USING MYSQL)

Create following tables using a given schema and insert appropriate data into the same:
Customer (CustID, Name, Cust_Address, Phone_no, Email_ID, Age)
Branch (Branch ID, Branch_Name, Address)
Account (Account_no, Branch ID, CustID, date_open, Account_type, Balance)
1. Modify the size of column “Email_Address” to 20 in Customer table.
2. Change the column “Email_Address” to Not Null in Customer table.
3. Display the total customers with the balance >50, 000 Rs.
4. Display average balance for account type=”Saving Account”.
5. Display the customer details that lives in Pune or name starts with ’A’.
6. Create a table Saving_Account with (Account_no, Branch ID, CustID, date_open, Balance) using
Account Table.
7. Display the customer details Age wise with balance>=20,000Rs
----

Create Tables

```
-- Create the Customer table
CREATE TABLE Customer (
    CustID INT PRIMARY KEY,
    Name VARCHAR(100),
    Cust_Address VARCHAR(255),
    Phone_no VARCHAR(20),
    Email_ID VARCHAR(100),
    Age INT
);
```

```
-- Create the Branch table
CREATE TABLE Branch (
    Branch_ID INT PRIMARY KEY,
    Branch_Name VARCHAR(100),
    Address VARCHAR(255)
);
```

```
-- Create the Account table
CREATE TABLE Account (
    Account_no INT PRIMARY KEY,
    Branch_ID INT,
    CustID INT,
    date_open DATE,
    Account_type VARCHAR(50),
    Balance DECIMAL(10, 2)
);
```

Insert into Tables

```
-- Insert sample data into the Customer table
INSERT INTO Customer (CustID, Name, Cust_Address, Phone_no, Email_ID,
Age)
VALUES
   (1, 'John Smith', '123 Main St, CityA', '123-456-7890', 'john@example.com', 35),
   (2, 'Alice Johnson', '456 Elm St, CityB', '987-654-3210', 'alice@example.com',
28),
   (3, 'David Brown', '789 Oak St, CityC', '567-890-1234', 'david@example.com',
42);

-- Insert sample data into the Branch table
INSERT INTO Branch (Branch_ID, Branch_Name, Address)
VALUES
   (101, 'Branch1', '123 Oak Ave, CityA'),
   (102, 'Branch2', '456 Maple St, CityB'),
   (103, 'Branch3', '789 Birch Rd, CityC');

-- Insert sample data into the Account table
INSERT INTO Account (Account_no, Branch_ID, CustID, date_open,
Account_type, Balance)
VALUES
   (1001, 101, 1, '2023-01-15', 'Savings', 5000.00),
   (1002, 102, 2, '2023-02-20', 'Checking', 7500.00),
   (1003, 103, 3, '2023-03-10', 'Savings', 6000.00);
```

1. Modify the size of column “Email_Address” to 20 in Customer table.

2. Change the column “Email_Address” to Not Null in Customer table.

```
ALTER TABLE Customer
MODIFY COLUMN Email_ID VARCHAR(20) NOT NULL;
```

3. Display the total customers with the balance >50, 000 Rs.

```
SELECT COUNT(*) AS Total_Customers
FROM Account
WHERE Balance > 50000.00;
```

4. Display average balance for account type=”Saving Account”.

```
SELECT AVG(Balance) AS Average_Balance
FROM Account
WHERE Account_type = 'Savings Account';
```

5. Display the customer details that lives in Pune or name starts with ’A’.

```
SELECT *
FROM Customer
WHERE Cust_Address LIKE '%Pune%' OR Name LIKE 'A%';
```

6. Create a table Saving_Account with (Account_no, Branch ID, CustID, date_open, Balance) using Account Table.

```
CREATE TABLE Saving_Account AS
SELECT Account_no, Branch_ID, CustID, date_open, Balance
FROM Account
WHERE Account_type = 'Savings Account';
```

7. Display the customer details Age wise with balance>=20,000Rs

```
SELECT Age, COUNT(*) AS Total_Customers
FROM Customer c
INNER JOIN Account a ON c.CustID = a.CustID
WHERE a.Balance >= 20000.00
GROUP BY Age
ORDER BY Age ASC;
```

Problem Statement 11 (DDL USING MYSQL)

Create following tables using a given schema and insert appropriate data into the same:

Customer (CustID, Name, Cust_Address, Phone_no, Email_ID, Age)

Branch (Branch ID, Branch_Name, Address)

Account (Account_no, Branch ID, CustID, date_open, Account_type, Balance)

1. Create the tables with referential integrity.
2. Draw the ER diagram for the same.
3. Create an Index on primary key column of table Account
4. Create the view as Customer_Info displaying the customer details for age less than 45.
5. Update the View with open date as 16/4/2017
6. Create a sequence on Branch able.
7. Create synonym ‘Branch_info’ for branch table.

Create Table

```
-- Create the Customer table
CREATE TABLE Customer (
   CustID INT PRIMARY KEY,
   Name VARCHAR(100),
   Cust_Address VARCHAR(255),
   Phone_no VARCHAR(20),
   Email_ID VARCHAR(100) UNIQUE,
   Age INT
);
```

```
-- Create the Branch table
CREATE TABLE Branch (
   Branch_ID INT PRIMARY KEY,
   Branch_Name VARCHAR(100),
   Address VARCHAR(255)
);
```

```
-- Create the Account table with foreign keys for Branch and Customer
CREATE TABLE Account (
   Account_no INT PRIMARY KEY,
   Branch_ID INT,
   CustID INT,
   date_open DATE,
   Account_type VARCHAR(50),
   Balance DECIMAL(10, 2),
   FOREIGN KEY (Branch_ID) REFERENCES Branch(Branch_ID),
   FOREIGN KEY (CustID) REFERENCES Customer(CustID)
);
```

Insert Values

```
-- Insert sample data into the Customer table
INSERT INTO Customer (CustID, Name, Cust_Address, Phone_no, Email_ID,
Age)
VALUES
   (1, 'John Smith', '123 Main St, CityA', '123-456-7890', 'john@example.com', 35),
   (2, 'Alice Johnson', '456 Elm St, CityB', '987-654-3210', 'alice@example.com',
28),
   (3, 'David Brown', '789 Oak St, CityC', '567-890-1234', 'david@example.com',
42);

-- Insert sample data into the Branch table
INSERT INTO Branch (Branch_ID, Branch_Name, Address)
VALUES
   (101, 'Branch1', '123 Oak Ave, CityA'),
   (102, 'Branch2', '456 Maple St, CityB'),
   (103, 'Branch3', '789 Birch Rd, CityC');

-- Insert sample data into the Account table
INSERT INTO Account (Account_no, Branch_ID, CustID, date_open,
Account_type, Balance)
VALUES
   (1001, 101, 1, '2023-01-15', 'Savings', 5000.00),
   (1002, 102, 2, '2023-02-20', 'Checking', 7500.00),
   (1003, 103, 3, '2023-03-10', 'Savings', 6000.00);
```

2. Draw the ER diagram for the same.

3. Create an Index on primary key column of table Account

```
CREATE INDEX account_no_idx ON Account (Account_no);
```

4. Create the view as Customer_Info displaying the customer details for age less than 45.

```
CREATE OR REPLACE VIEW Customer_Info AS
SELECT c.CustID, c.Name, c.Cust_Address, c.Phone_no, c.Email_ID, c.Age,
a.Account_no, a.date_open, a.Account_type, a.Balance
FROM Customer AS c INNER JOIN Account AS a ON c.CustID = a.CustID
WHERE Age < 40;
```

5. Update the View with open date as 16/4/2017

```
UPDATE VIEW Customer_Info SET date_open = '2017-04-16';
```

6. Create a sequence on Branch table.

```
CREATE SEQUENCE Branch_seq START WITH 1 INCREMENT BY 1;
```

7. Create synonym ‘Branch_info’ for branch table.

```
CREATE SYNONYM Branch_info FOR Branch;
```

Problem Statement 14 (JOINS & SUBQUERIES USING MYSQL)

Consider Following Schema
Employee (Employee_id, First_name, last_name , hire_date, salary, Job_title, manager_id, department_id)
Departments(Department_id, Department_name, Manager_id, Location_id)
Locations(location_id ,street_address ,postal_code, city, state, country_id)
Manager(Manager_id, Manager_name)
Create the tables with referential integrity. Solve following queries using joins and subqueries.

1. Write a query to find the names (first_name, last_name) and the salaries of the employees who have a higher salary than the employee whose last_name="Singh”.
2. Write a query to find the names (first_name, last_name) of the employees who have a manager and work for a department based in the United States.
3. Write a query to find the names (first_name, last_name), the salary of the employees whose salary is greater than the average salary.
4. Write a query to find the employee id, name (last_name) along with their manager_id, manager name (last_name).
5. Find the names and hire date of the employees who were hired after 'Jones'.

1. Write a query to find the names (first_name, last_name) and the salaries of the employees who have a higher salary than the employee whose last_name="Singh”.

```
SELECT First_name, Last_name, Salary FROM Employee
WHERE Salary > (SELECT Salary FROM Employee WHERE Last_name =
"Singh");
```

2. Write a query to find the names (first_name, last_name) of the employees who have a manager and work for a department based in the United States.

```
SELECT First_name,Last_name FROM Employee AS e INNER JOIN Departments
AS d ON e.Department_id = d.Department_id INNER JOIN Locations AS l ON
d.Location_id = l.Location_id WHERE City = "New York";
```

3. Write a query to find the names (first_name, last_name), the salary of the employees whose salary is greater than the average salary.

```
SELECT First_name, Last_name, Salary
FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```

4. Write a query to find the employee id, name (last_name) along with their manager_id, manager name (last_name).

```
SELECT e.Employee_id, e.Last_name, e.Manager_id, m.Manager_name
FROM Employee e
LEFT JOIN Manager m ON e.Manager_id = m.Manager_id;
```

5. Find the names and hire date of the employees who were hired after 'Jones'.

```
SELECT First_name, Last_name, Hire_date
FROM Employee
WHERE Hire_date > (SELECT Hire_date FROM Employee WHERE Last_name
= 'Jones');
```

Problem Statement 17 (DML USING MYSQL)

Create following tables using a given schema and insert appropriate data into the same:

Customer (CustID, Name, Cust_Address, Phone_no, Age)

Branch (Branch ID, Branch_Name, Address)

Account (Account_no, Branch ID, CustID, date_open, Account_type, Balance)

1. Add the column “Email_Address” in Customer table.
2. Change the name of column “Email_Address” to “Email_ID” in Customer table.
3. Display the customer details with highest balance in the account.
4. Display the customer details with lowest balance for account type= “Saving Account”.
5. Display the customer details that live in Pune and have age greater than 35.
6. Display the Cust_ID, Name and Age of the customer in ascending order of their age.
7. Display the Name and Branch ID of the customer group by the Account_type.

1. Add the column “Email_Address” in Customer table.

```
ALTER TABLE Customer
ADD Email_Address VARCHAR(50);
```

2. Change the name of column “Email_Address” to “Email_ID” in Customer table.

```
ALTER TABLE Customer
CHANGE COLUMN Email_Address Email_ID VARCHAR(50);
```

3. Display the customer details with highest balance in the account.

```
SELECT * FROM Customer
WHERE CustID
IN (SELECT CustID FROM Account WHERE Balance = (SELECT
MAX(Balance) FROM Account));
```

4. Display the customer details with lowest balance for account type= “Saving Account”.

```
SELECT * FROM Customer WHERE CustID IN (Select CUSTID FROM
Account WHERE Balance = (SELECT Min(Balance) FROM Account WHERE
Account_type = "Savings"));
```

5. Display the customer details that live in Pune and have age greater than 35.

```
SELECT * FROM Customer
WHERE Cust_Address
LIKE '%Pine%' AND Age > 35;
```

6. Display the Cust_ID, Name and Age of the customer in ascending order of their age.

```
SELECT CustID,Name,Age FROM Customer ORDER BY Age ASC;
```

7. Display the Name and Branch ID of the customer group by the Account_type.

```
SELECT Name,Branch_ID,Account_type
FROM Customer AS c
INNER JOIN Account AS a ON c.CustID = a.CustID
GROUP BY Name, Branch_ID, Account_type;
```

```
SELECT COUNT(Name), a.Account_type
FROM Customer c
JOIN Account a ON c.CustID = a.CustID
GROUP BY  a.Account_type;
```

Problem Statement 20 (DDL USING MYSQL)

Create following tables using a given schema and insert appropriate data into the same:

Customer (CustID, Name, Cust_Address, Phone_no, Email_ID, Age)

Branch (Branch ID, Branch_Name, Address)

Account (Account_no, Branch ID, CustID, open_date, Account_type, Balance)

1. Create the tables with referential integrity.
2. Draw the ER diagram for the same.
3. Create a View as Saving account displaying the customer details with the open date as 16/8/2018.
4. Update the View with Cust_Address as Pune for CustID =103.
5. Create a View as Loan account displaying the customer details with the open date as 16/2/2018.
6. Create an Index on primary key column of table Customer.
7. Create an Index on primary key column of table Branch.
8. Create a sequence on Customer Table.
9. Create synonym 'Cust_info' for branch table.

1. Create the tables with referential integrity.

```
-- Create the Customer table
CREATE TABLE Customer (
    CustID INT PRIMARY KEY,
    Name VARCHAR(100),
    Cust_Address VARCHAR(255),
    Phone_no VARCHAR(20),
    Email_ID VARCHAR(100) UNIQUE,
    Age INT
);

-- Create the Branch table
CREATE TABLE Branch (
    Branch_ID INT PRIMARY KEY,
    Branch_Name VARCHAR(100),
    Address VARCHAR(255)
);

-- Create the Account table with foreign keys for Branch and Customer
CREATE TABLE Account (
    Account_no INT PRIMARY KEY,
    Branch_ID INT,
    CustID INT,
    open_date DATE,
    Account_type VARCHAR(50),
    Balance DECIMAL(10, 2),
    FOREIGN KEY (Branch_ID) REFERENCES Branch(Branch_ID),
    FOREIGN KEY (CustID) REFERENCES Customer(CustID)
);
```

2. Draw the ER diagram for the same.

3. Create a View as Saving account displaying the customer details with the open date as 16/8/2018.

```
CREATE VIEW Saving_Account AS
SELECT c.*
FROM Customer c
JOIN Account a ON c.CustID = a.CustID
WHERE a.Account_type = 'Saving Account' AND a.open_date = '2018-08-16';
```

4. Update the View with Cust_Address as Pune for CustID =103.

```
UPDATE Customer
SET Cust_Address = 'Pune'
WHERE CustID = 103;
```

5. Create a View as Loan account displaying the customer details with the open date as 16/2/2018.

```
CREATE VIEW Loan_Account AS
SELECT c.*
FROM Customer c
JOIN Account a ON c.CustID = a.CustID
WHERE a.Account_type = 'Loan Account' AND a.open_date = '2018-02-16';
```

6. Create an Index on primary key column of table Customer.

```
CREATE INDEX customer_pk_index ON Customer (CustID);
```

7. Create an Index on primary key column of table Branch.

```
CREATE INDEX branch_pk_index ON Branch (Branch_ID);
```

8. Create a sequence on Customer Table.

```
CREATE SEQUENCE customer_sequence
START WITH 1
INCREMENT BY 1;
```

9. Create synonym ‘Cust_info’ for branch table.

```
CREATE SYNONYM Cust_info FOR Branch;
```

Problem Statement 1 (Triggers)

Employee(emp_id, emp_name, salary, designation)

Salary_Backup(emp_id, old_salary, new_salary, salary_difference)

```
CREATE TABLE Employee (
   emp_id INT PRIMARY KEY,
   emp_name VARCHAR(255),
   salary DECIMAL(10, 2),
   designation VARCHAR(50)
);

CREATE TABLE Salary_Backup (
   emp_id INT,
   old_salary DECIMAL(10, 2),
   new_salary DECIMAL(10, 2),
   salary_difference DECIMAL(10, 2)
);
```

1. Create a Trigger to record salary change of the employee. Whenever salary is updated insert the details in

```
CREATE OR REPLACE TRIGGER Salary_Change_Trigger
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
  IF :OLD.salary != :NEW.salary THEN
    INSERT INTO Salary_Backup(emp_id, old_salary, new_salary,
salary_difference)
    VALUES(:OLD.emp_id, :OLD.salary, :NEW.salary, :NEW.salary - :OLD.salary);
  END IF;
END;
```

2. Create a Trigger that will prevent deleting the employee record having designation as CEO.

```
CREATE OR REPLACE TRIGGER Prevent_Deletion_Trigger
BEFORE DELETE ON Employee
FOR EACH ROW
BEGIN
  IF :OLD.designation = 'CEO' THEN
    RAISE_APPLICATION_ERROR(-20001, 'You cannot delete an employee with
the designation "CEO".');
  END IF;
END;
```

Problem Statement 4 (Procedures / Functions)

Consider following schema for Bank database.

Account(Account_No, Cust_Name, Balance, NoOfYears)

Earned_Interest(Account_No, Interest_Amt)

```
-- Create the Account table
CREATE TABLE Account (
  Account_No NUMBER PRIMARY KEY,
  Cust_Name VARCHAR2(100),
  Balance NUMBER,
  NoOfYears NUMBER
);

-- Create the Earned_Interest table
CREATE TABLE Earned_Interest (
  Account_No NUMBER PRIMARY KEY,
  Interest_Amt NUMBER
);
```

1. Write a PL/SQL procedure for following requirement. Take as input Account_No and Interest Rate from User. Calculate the Interest_Amt as simple interest for the given Account_No and store it in Earned_Interest table. Display all the details of Earned_Interest Table.

```
CREATE OR REPLACE PROCEDURE calculate(
    acc_no IN INTEGER,
    interest_rate IN INTEGER
) AS
interest_amt integer;
principle_amt integer;
interest_time integer;
BEGIN
SELECT Balance INTO principle_amt FROM ACCOUNT WHERE Account_no = acc_no;
SELECT Noofyears INTO interest_time FROM ACCOUNT WHERE Account_no = acc_no;
interest_amt := (principle_amt*interest_rate*interest_time)/100;
INSERT INTO earned_interest VALUES (acc_no,interest_amt);
END;
```

```
EXEC calculate(1,5);
```

2. Write a PLSQL function to display all records from Account table having Balance greater than 50,000.

```
CREATE OR REPLACE FUNCTION GetHighBalanceAccounts RETURN
SYS_REFCURSOR AS
   v_cursor SYS_REFCURSOR;
BEGIN
   OPEN v_cursor FOR
   SELECT * FROM Account WHERE Balance > 50000;

   RETURN v_cursor;
END GetHighBalanceAccounts;
/
```

Call and Display The Values

```
DECLARE
   v_cursor SYS_REFCURSOR;
   v_Account_No Account.Account_No%TYPE;
   v_Cust_Name Account.Cust_Name%TYPE;
   v_Balance Account.Balance%TYPE;
   v_NoOfYears Account.NoOfYears%TYPE;
BEGIN
   -- Call the function to get a cursor with the result set
   v_cursor := GetHighBalanceAccounts;

   -- Loop through the cursor and display the values
   LOOP
      FETCH v_cursor INTO v_Account_No, v_Cust_Name, v_Balance,
v_NoOfYears;
      EXIT WHEN v_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE('Account_No: ' || v_Account_No || ',
Cust_Name: ' || v_Cust_Name || ', Balance: ' || v_Balance || ', NoOfYears: ' ||
v_NoOfYears);
   END LOOP;

   -- Close the cursor
   CLOSE v_cursor;
END;
/
```

Problem Statement 10 (Triggers)

Employee(emp_id, dept_idemp_name, DoJ, salary, commission,job_title)

Consider the schema given above for Write a PLSQL Program to

1. Create a Trigger to ensure the salary of the employee is not decreased.

```
CREATE OR REPLACE TRIGGER Salary_Not_Decreased
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
   IF :NEW.salary < :OLD.salary THEN
      RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be decreased.');
   END IF;
END;
/
```

2. Whenever the job title is changed for an employee write the following details into job_history table. Employee

ID, old job title, old department ID, DoJ for start date, system date for end date.

```
CREATE OR REPLACE TRIGGER Log_Job_Title_Changes
AFTER UPDATE ON Employee
FOR EACH ROW
WHEN (NEW.job_title <> OLD.job_title)
DECLARE
   v_Start_Date DATE;
BEGIN
   v_Start_Date := :OLD.DoJ;

   INSERT INTO Job_History (emp_id, old_job_title, old_dept_id, start_date,
end_date)
   VALUES (:OLD.emp_id, :OLD.job_title, :OLD.dept_id, v_Start_Date,
SYSDATE);
END;
/
```

Problem Statement 16 (Procedures / Functions)

Employee(emp_id, dept_idemp_name, DoJ, salary, commission,job_title)

1. Consider the schema given above. Keep the commission column empty initially. Write a Stored Procedure to record the employee commission based on following conditions. If salary is more than 10000 then commission is 0.4%, if Salary is less than 10000 but experience is more than 10 years then 0.35%, if salary is less than 3000 then commission is 0.25%. In the remaining cases commission is 0.15%.

```
CREATE OR REPLACE PROCEDURE RecordEmployeeCommission(
    emp_id IN NUMBER,
    salary IN NUMBER,
    DoJ IN DATE
) AS
    commission_rate NUMBER;
BEGIN
    IF salary > 10000 THEN
        commission_rate := 0.004; -- 0.4%
    ELSIF salary < 10000 AND (SYSDATE - DoJ) > 3650 THEN -- 10 years in days
        commission_rate := 0.0035; -- 0.35%
    ELSIF salary < 3000 THEN
        commission_rate := 0.0025; -- 0.25%
    ELSE
        commission_rate := 0.0015; -- 0.15%
    END IF;

    -- Update the commission column in the Employee table for the given emp_id
    UPDATE Employee
    SET commission = salary * commission_rate
    WHERE emp_id = emp_id;
END;
/
```

2. Write a PLSQL Function that takes department ID and returns the name of the manager of the department.

```
CREATE OR REPLACE FUNCTION GetDepartmentManager(
   dept_id IN NUMBER
) RETURN VARCHAR2 AS
   manager_name VARCHAR2(100);
BEGIN
   SELECT emp_name INTO manager_name
   FROM Employee
   WHERE dept_id = dept_id AND job_title = 'Manager';

   RETURN manager_name;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN NULL; -- Department has no manager
END;
/
```

Problem Statement 7 (Cursors)

Consider the following schema for Products table.

Products(Product_id, Product_Name, Product_Type, Price)

1. Write a parameterized cursor to display all products in the given price range of price and type 'Apparel'.

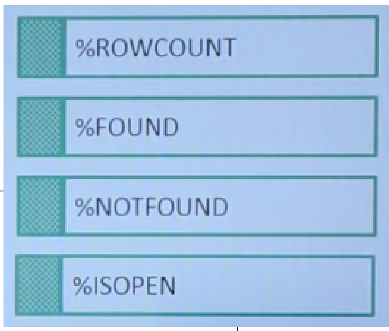
Hint: Take the user input for minimum and maximum price for price range.

```
DECLARE
prod_name Products.Product_Name%TYPE;
prod_type Products.Product_Type%TYPE;
prod_price Products.Price%TYPE;
CURSOR cur(max NUMBER,min NUMBER) IS SELECT
product_name,product_type,price FROM Products WHERE Product_Type =
'Apparel' AND price BETWEEN min AND max;

BEGIN

        OPEN cur(10,50);
        LOOP
        FETCH cur INTO prod_name,prod_type,prod_price;
        EXIT WHEN cur%NOTFOUND;
        dbms_output.put_line('Product Name : '||prod_name||' Product Type : '||
prod_type||' Price : '||prod_price);
        END LOOP;
CLOSE cur;
END;
```

3. Write an implicit cursor to display the number of records affected by the update operation incrementing Price of all products by 1000.



```
DECLARE
   rows_updated NUMBER;
BEGIN
   UPDATE Products SET Price = Price + 1000;
   rows_updated := SQL%ROWCOUNT;  -- Get the number of rows affected
   DBMS_OUTPUT.PUT_LINE('Number of records updated: ' || rows_updated);
END;
```

Problem Statement 13 (Cursors)

Consider a table Employee with schema as Employee (Emp_id, Emp_Name,Salary).

1. Write an explicit cursor to display records of all employees with salary greater than 50,000.

```
DECLARE
   employee_name VARCHAR2(50);
        employee_salary INT;
        CURSOR cur IS SELECT emp_name,salary FROM Employee WHERE
salary > 50000;
BEGIN
        OPEN cur;
        LOOP
        FETCH cur INTO employee_name,employee_salary;
        EXIT WHEN cur%NOTFOUND;
        dbms_output.put_line('Name : '||employee_name||' Salary : '||
employee_salary);
        END LOOP;
CLOSE cur;
END;
```

2. Write a PL/SQL block of code using Implicit Cursor that will display total number of tuples in Employee table.

```
DECLARE
        total NUMBER;
BEGIN
        SELECT COUNT(*) INTO total FROM Employee;
        dbms_output.put_line(total);
END;
```

3. Write a PL/SQL block of code using Parameterized Cursor that will display salary of employee id entered by the user.

```
DECLARE
   salary NUMBER;
        CURSOR cur(id NUMBER) IS SELECT salary FROM Employee WHERE
emp_id = id;
BEGIN
        OPEN cur(1);
        LOOP
        FETCH cur INTO salary;
        EXIT WHEN cur%NOTFOUND;
        dbms_output.put_line('Salary : '||salary);
        END LOOP;
        CLOSE cur;
END;
```

Problem Statement 19 (Unnamed Block)

Employee(emp_id, dept_idemp_name, DoJ, salary, commission,job_title)

Salary_Increment(emp_id, new_salary)

Consider the schema given above. Write a PLSQL Unnamed Block of code to increase the salary of employee

115 based on the following conditions:

Accept emp_id from user. If experience of employee is more than 10 years, increase salary by 20%. If experience

is greater than 5 years, increase salary by 10% Otherwise 5%. (Hint: Find the years of experience from Date of

Joining (DoJ)). Store the incremented salary in Salary_Increment table.

Also handle the exception by named exception handler or user defined exception handler.

Problem Statement 19 (Unnamed Block)

Employee(emp_id, dept_idemp_name, DoJ, salary, commission,job_title)

Salary_Increment(emp_id, new_salary)

Consider the schema given above. Write a PLSQL Unnamed Block of code to increase the salary of employee 115 based on the following conditions:

Accept emp_id from user. If experience of employee is more than 10 years, increase salary by 20%. If experience is greater than 5 years, increase salary by 10% Otherwise 5%. (Hint: Find the years of experience from Date of

Joining (DoJ)). Store the incremented salary in Salary_Increment table.

Also handle the exception by named exception handler or user defined exception handler.

```
DECLARE
   v_emp_id NUMBER;
   v_doj DATE;
   v_experience NUMBER;
   v_increment_percentage NUMBER;
   v_new_salary NUMBER;

   -- User-defined exception
   InvalidEmployee EXCEPTION;

BEGIN
   -- Accept emp_id from the user
   v_emp_id := &emp_id; -- You can also use a prompt for user input

   -- Retrieve Date of Joining (DoJ) and salary of the employee
   SELECT DoJ, salary
   INTO v_doj, v_new_salary
   FROM Employee
   WHERE emp_id = v_emp_id;

   -- Calculate years of experience
   v_experience := TRUNC(MONTHS_BETWEEN(SYSDATE, v_doj) / 12);

   -- Determine the increment percentage based on experience
   IF v_experience > 10 THEN
      v_increment_percentage := 0.20; -- 20% increment for more than 10 years
   ELSIF v_experience > 5 THEN
      v_increment_percentage := 0.10; -- 10% increment for more than 5 years
   ELSE
```

```
        v_increment_percentage := 0.05; -- 5% increment for less than or equal to 5
years
    END IF;

    -- Calculate the new salary with the increment
    v_new_salary := v_new_salary + (v_new_salary * v_increment_percentage);

    -- Insert the incremented salary into the Salary_Increment table
    INSERT INTO Salary_Increment (emp_id, new_salary)
    VALUES (v_emp_id, v_new_salary);

    DBMS_OUTPUT.PUT_LINE('Salary increment for employee ' || v_emp_id || ': ' ||
v_new_salary);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'Employee with ID ' || v_emp_id || '
not found.');
    WHEN InvalidEmployee THEN
        DBMS_OUTPUT.PUT_LINE('Invalid employee data.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
        ROLLBACK;
END;
/
```

Problem Statement 12 (CRUD Using MongoDB)

Create a collection Social_Media having fields as User_Id, User_Name, No_of_Posts, No_of_Friends,

Friends_List, Interests. (Hint: Friends_List and Interests can be of array type)

Insert 20 documents in the collection Social_Media. Write queries for following.

```
db.createCollection("Social_Media")

db.Social_Media.insertMany([
    {
        User_Id: 1,
        User_Name: "user1",
        No_of_Posts: 75,
        No_of_Friends: 10,
        Friends_List: ["friend1", "friend2", "friend3"],
        Interests: ["interest1", "interest2"]
    },
    {
        User_Id: 2,
        User_Name: "user2",
        No_of_Posts: 120,
        No_of_Friends: 8,
        Friends_List: ["friend4", "friend5", "friend6"],
        Interests: ["interest3", "interest4"]
    },
    // ... Insert 18 more documents with similar structure
])
```

1. List all the users from collection Social_Media in formatted manner.

```
db.Social_Media.find({}, { _id: 0, User_Id: 1, User_Name: 1 })
```

2. Find all users having number of posts greater than 100.

```
db.Social_Media.find({ No_of_Posts: { $gt: 100 } })
```

3. List the user names and their respective Friens_List

```
db.Social_Media.find({}, { _id: 0, User_Name: 1, Friends_List: 1 })
```

4. Display the user ids and Friends list of users who have more than 5 friends.

```
db.Social_Media.find({ No_of_Friends: { $gt: 5 } }, { _id: 0, User_Id: 1,
Friends_List: 1 })
```

5. Display all users with no of posts in descending order.

```
db.Social_Media.find({}, { _id: 0, User_Name: 1, No_of_Posts:
1 }).sort({ No_of_Posts: -1 })
```

Problem Statement 21 (CRUD)

Create collection Student with fields as Roll_No, Name, Class, Marks, Address, Enrolled_Courses.

(Hint: One student can enrol in multiple courses. Use Array to store the names of courses enrolled)

Insert 10 documents in the collection Student. Write the queries for following.

```
db.createCollection("Student")

db.Student.insertMany([
   {
      Roll_No: "A1",
      Name: "Alice",
      Class: "SE",
      Marks: 75,
      Address: "123 Main St",
      Enrolled_Courses: ["DBMS", "TOC"]
   },
   {
      Roll_No: "A2",
      Name: "Bob",
      Class: "TE",
      Marks: 45,
      Address: "456 Elm St",
      Enrolled_Courses: ["DBMS", "AI"]
   },
   {
      Roll_No: "A3",
      Name: "Charlie",
      Class: "SE",
      Marks: 60,
      Address: "789 Oak St",
      Enrolled_Courses: ["TOC", "AI"]
   },
   // Insert 7 more documents with similar structure
])
```

1. List the names of students who have enrolled in the course “DBMS”, “TOC”.

```
db.Student.find({ Enrolled_Courses: { $all: ["DBMS", "TOC"] }, }, { _id: 0, Name:
1 })
```

2. List the Roll numbers and class of students who have marks more than 50 or class as TE.

```
db.Student.find({ $or: [{ Marks: { $gt: 50 } }, { Class: "TE" } ] }, { _id: 0, Roll_No:
1, Class: 1 })
```

3. Update the entire record of roll_no A10.

```
db.Student.update({ Roll_No: "A10" }, { /* Your updated document here */ })
```

4. Display the names of students having 3rd and 4th highest marks.

```
db.Student.find({}, { _id: 0, Name: 1, Marks: 1 }).sort({ Marks: -
1 }).skip(2).limit(2)
```

5. Delete the records of students having marks less than 20.

```
db.Student.deleteMany({ Marks: { $lt: 20 } })
```

6. Delete only first record from the collection.

```
db.Student.deleteOne({})
```

Problem Statement 3 (Aggregation & Indexing)

Create the Collection Movies_Data(Movie_ID, Movie_Name, Director, Genre, BoxOfficeCollection) and solve the following:

```
db.createCollection("Movies_Data")

db.Movies_Data.insertMany([
  {
    Movie_ID: 1,
    Movie_Name: "Movie 1",
    Director: "Director A",
    Genre: "Action",
    BoxOfficeCollection: 1000000
  },
  {
    Movie_ID: 2,
    Movie_Name: "Movie 2",
    Director: "Director A",
    Genre: "Drama",
    BoxOfficeCollection: 1200000
  },
  {
    Movie_ID: 3,
    Movie_Name: "Movie 3",
    Director: "Director B",
    Genre: "Action",
    BoxOfficeCollection: 800000
  },
  {
    Movie_ID: 4,
    Movie_Name: "Movie 4",
    Director: "Director B",
    Genre: "Comedy",
    BoxOfficeCollection: 1500000
  }
])
```

1. Display a list stating how many Movies are directed by each “Director”.

```
db.Movies_Data.aggregate([
  {
    $group: {
      _id: "$Director",
      totalMovies: { $sum: 1 }
    }
  }
])
```

This MongoDB aggregation query groups movies in the "Movies_Data" collection by their directors and counts the total number of movies directed by each director, creating a result that lists directors and their respective movie counts.

2. Display list of Movies with the highest BoxOfficeCollection in each Genre.

```
db.Movies_Data.aggregate([
  {
    $group: {
      _id: "$Genre",
      maxCollection: { $max: "$BoxOfficeCollection" }
    }
  },
  {
    $lookup: {
      from: "Movies_Data",
      let: { maxCollection: "$maxCollection", genre: "$_id" },
      pipeline: [
        {
          $match: {
            $expr: {
              $and: [
                { $eq: ["$Genre", "$$genre"] },
                { $eq: ["$BoxOfficeCollection", "$$maxCollection"] }
              ]
            }
          }
        },
        {
          $project: {
            _id: 0,
            Movie_Name: 1,
            Director: 1,
            Genre: 1,
            BoxOfficeCollection: 1
          }
        }
      ],
      as: "movies"
    }
  }
])
```

3. Display list of Movies with the highest BoxOfficeCollection in each Genre in ascending order of BoxOfficeCollection.

```
db.Movies_Data.aggregate([
 {
  $group: {
    _id: "$Genre",
    maxCollection: { $max: "$BoxOfficeCollection" }
  }
 },
 {
  $lookup: {
    from: "Movies_Data",
    let: { maxCollection: "$maxCollection", genre: "$_id" },
    pipeline: [
      {
        $match: {
          $expr: {
            $and: [
              { $eq: ["$Genre", "$$genre"] },
              { $eq: ["$BoxOfficeCollection", "$$maxCollection"] }
            ]
          }
        }
      },
      {
        $project: {
          _id: 0,
          Movie_Name: 1,
          Director: 1,
          Genre: 1,
          BoxOfficeCollection: 1
        }
      }
    ],
    as: "movies"
  }
 },
 {
  $unwind: "$movies"
 },
 {
  $sort: {
    "movies.BoxOfficeCollection": 1
  }
 }
])
```

4. Create an index on field Movie_ID.

```
db.Movies_Data.createIndex({ Movie_ID: 1 })
```

5. Create an index on fields ” Movie_Name” and ” Director”.

```
db.Movies_Data.createIndex({ Movie_Name: 1, Director: 1 })
```

6. Drop an index on field Movie_ID.

```
db.Movies_Data.dropIndex("Movie_ID_1")
```

7. Drop an index on fields ” Movie_Name” and ” Director”.

```
db.Movies_Data.dropIndex({ Movie_Name: 1, Director: 1 })
```

What is aggregation

Aggregation in MongoDB refers to the process of transforming and processing data from a MongoDB collection to produce structured and meaningful results. MongoDB's aggregation framework provides a powerful and flexible set of tools to perform a wide range of data manipulations, transformations, and computations on the data stored in collections.

Key features and components of MongoDB's aggregation framework include:

1. **Aggregation Pipeline**: The core concept of MongoDB's aggregation is the aggregation pipeline. The aggregation pipeline is a sequence of stages, each of

which performs a specific data transformation or operation on the input documents and passes the results to the next stage. Stages can include filtering, grouping, sorting, projecting, and more.

2. **Aggregation Operators**: MongoDB provides a wide range of aggregation operators that can be used within the pipeline stages. These operators include `$match` for filtering, `$group` for grouping, `$project` for reshaping documents, `$sort` for sorting, `$lookup` for joining data from other collections, and many more.
3. **Expression Operators**: Aggregation expressions allow you to perform complex operations within the pipeline stages. Operators like `$add`, `$subtract`, `$multiply`, and `$divide` are used for arithmetic operations, while others like `$eq`, `$ne`, `$gt`, and `$lt` are used for comparisons.
4. **Accumulators**: Accumulators like `$sum`, `$avg`, `$max`, and `$min` are used in the `$group` stage to perform calculations on grouped data.
5. **Aggregation Functions**: MongoDB provides various aggregation functions to work with data, such as `$group`, `$project`, `$match`, and `$sort`. These functions enable you to shape and analyze the data according to your requirements.
6. **Dynamic Pipelines**: Aggregation pipelines can be dynamic, meaning that the stages and operations can change based on the data or conditions.

Aggregation in MongoDB is useful for a wide range of data analysis tasks, including:

- Summarizing and reporting on data.
- Transforming data into a desired format for reporting or visualization.
- Grouping and counting data.
- Joining data from multiple collections.
- Calculating statistics and metrics.

Overall, MongoDB's aggregation framework is a powerful tool for performing complex data manipulations and deriving insights from your data stored in MongoDB collections.

1. `$group` Stage:
    - It groups the documents in the collection by the "Genre" field (`_id: "$Genre"`).
    - Within each group, it calculates the maximum "BoxOfficeCollection" value and names it "maxCollection" using the `$max` aggregation operator.
2. `$lookup` Stage:
    - This stage performs a lookup operation to retrieve related documents from the same "Movies_Data" collection.
    - `from: "Movies_Data"` specifies the same collection as the source for the lookup.
    - `let` defines variables (`maxCollection` and `genre`) to be used in the pipeline.
    - The `pipeline` field specifies an array of aggregation stages to apply to the documents in the "Movies_Data" collection for the lookup operation.
3. `$match` Stage (inside the `$lookup` pipeline):
    - It filters documents within the same genre and with the same "BoxOfficeCollection" as the maximum value found in the previous `$group` stage.
    - The `$expr` operator allows you to use expressions to compare field values.
    - `$eq` is used to check equality for both the "Genre" and "BoxOfficeCollection" fields between the current document and the variables defined in the `let` clause.
4. `$project` Stage (inside the `$lookup` pipeline):
    - It shapes the output of the lookup operation by including or excluding specific fields.
    - In this case, it includes the "Movie_Name," "Director," "Genre," and "BoxOfficeCollection" fields while excluding the default `_id` field.
5. `as: "movies"` (inside the `$lookup` stage):

- It assigns the resulting documents from the lookup operation to a new array field called "movies."

6. the $**expr** operator is used to compare the values of "field1" and "field2" in each document,

Problem Statement 18 (AGGREGATION & INDEXING USING MONGODB)

Create the Collection Student_Data(Student _ID, Student _Name, Department, Marks)and solve the following:

```
db.createCollection("Student_Data")

db.Student_Data.insertMany([
  {
    Student_ID: 1,
    Student_Name: "Alice",
    Department: "Math",
    Marks: 90
  },
  {
    Student_ID: 2,
    Student_Name: "Bob",
    Department: "Math",
    Marks: 88
  },
  {
    Student_ID: 3,
    Student_Name: "Charlie",
    Department: "Physics",
    Marks: 95
  },
  {
    Student_ID: 4,
    Student_Name: "David",
    Department: "Physics",
    Marks: 92
  }
])
```

1. Display all Students based on their departments along with an average Marks of a particular department.

```
db.Student_Data.aggregate([
  {
    $group: {
      _id: "$Department",
      Students: { $push: "$$ROOT" },
      AvgMarks: { $avg: "$Marks" }
    }
  }
])
```

2. Displays the number of Students associated along with a particular department.

```
db.Student_Data.aggregate([
  {
    $group: {
      _id: "$Department",
      NumStudents: { $sum: 1 }
    }
  }
])
```

3. Display list of Students with the highest Marks in each Department in descending order of Marks.

```
db.Student_Data.aggregate([
  {
    $sort: {
      Marks: -1
    }
  },
  {
    $group: {
      _id: "$Department",
      topStudent: { $first: "$$ROOT" }
    }
  },
  {
    $replaceRoot: { newRoot: "$topStudent" }
  }
])
```

4. Create an index on field Student_ID.

```
db.Student_Data.createIndex({ Student_ID: 1 })
```

5. Create an index on fields “Student_Name‘ and “Department”.

```
db.Student_Data.createIndex({ Student_Name: 1, Department: 1 })
```

6. Drop an index on field Student_ID.

```
db.Student_Data.dropIndex("Student_ID_1")
```

7. Drop an index on fields “Student_Name‘ and “Department”.

```
db.Student_Data.dropIndex({ Student_Name: 1, Department: 1 })
```

Problem Statement 9 (Map Reduce)

Create collection for Student{roll_no, name, class, dept, aggregate_marks}. Write Map Reduce Functions for following requirements.

```
db.createCollection("Student")

db.Student.insertMany([
  {
    roll_no: 1,
    name: "Alice",
    class: "TE",
    dept: "Math",
    aggregate_marks: 90
  },
  {
    roll_no: 2,
    name: "Bob",
    class: "SE",
    dept: "Physics",
    aggregate_marks: 88
  },
  {
    roll_no: 3,
    name: "Charlie",
    class: "TE",
    dept: "Math",
    aggregate_marks: 85
  },
  {
    roll_no: 4,
    name: "David",
    class: "BE",
    dept: "Physics",
    aggregate_marks: 92
  },
  {
    roll_no: 5,
    name: "Eve",
    class: "BE",
    dept: "Math",
    aggregate_marks: 95
  }
])
```

1. Finding the total marks of students of “TE” class department-wise.

```
var mapFunction = function() {
  if (this.class === "TE") {
    emit(this.dept, this.aggregate_marks);
  }
};

var reduceFunction = function(key, values) {
  return Array.sum(values);
};

db.Student.mapReduce(mapFunction, reduceFunction, { out: "TE_TotalMarks" });
db.TE_TotalMarks.find();
```

The code you provided is a basic example of how to use the map-reduce functionality in MongoDB to process data and store the results in a new collection. Let's break it down step by step:

1. **Map Function**:
   - `var mapFunction = function() { ... }`: This is a JavaScript function that defines the "map" part of the map-reduce process.
   - `if (this.class === "TE") { ... }`: This condition checks if the "class" field of the current document is equal to "TE."
   - `emit(this.dept, this.aggregate_marks);`: If the condition is met, it emits (outputs) a key-value pair. The "dept" field is used as the key, and the "aggregate_marks" field is used as the value for the emitted pair.
2. **Reduce Function**:
   - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.
   - `return Array.sum(values);`: It takes a key and an array of values and reduces them by calculating the sum of all the values associated with that key.
3. **Map-Reduce Operation**:

- `db.Student.mapReduce(mapFunction, reduceFunction, { out: "TE_TotalMarks" });`: This line initiates the map-reduce operation on the "Student" collection. It uses the map function defined earlier to map the data and the reduce function to reduce it. The results are stored in a new collection named "TE_TotalMarks."

4. **Result Retrieval**:
   - `db.TE_TotalMarks.find();`: This line retrieves the results from the "TE_TotalMarks" collection, which now contains the aggregated data.

2. Finding the highest marks of students of “SE” class department-wise.

```
var mapFunction = function() {
  if (this.class === "SE") {
    emit(this.dept, this.aggregate_marks);
  }
};

var reduceFunction = function(key, values) {
  return Math.max.apply(null, values);
};

db.Student.mapReduce(mapFunction, reduceFunction, { out:
"SE_HighestMarks" });
db.SE_HighestMarks.find();
```

This code is another example of using the map-reduce functionality in MongoDB to process data and store the results in a new collection. Let's break down what each part of the code is doing:

1. **Map Function**:
   - `var mapFunction = function() { ... }`: This is the JavaScript function that defines the "map" part of the map-reduce process.
   - `if (this.class === "SE") { ... }`: This condition checks if the "class" field of the current document is equal to "SE."

  - `emit(this.dept, this.aggregate_marks);`: If the condition is met, it emits (outputs) a key-value pair. The "dept" field is used as the key, and the "aggregate_marks" field is used as the value for the emitted pair.

2. **Reduce Function**:
   - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.
   - `return Math.max.apply(null, values);`: It takes a key and an array of values and reduces them by finding the maximum value in the array of values using the `Math.max` function.

3. **Map-Reduce Operation**:
   - `db.Student.mapReduce(mapFunction, reduceFunction, { out: "SE_HighestMarks" });`: This line initiates the map-reduce operation on the "Student" collection. It uses the map function defined earlier to map the data and the reduce function to reduce it. The results are stored in a new collection named "SE_HighestMarks."

4. **Result Retrieval**:
   - `db.SE_HighestMarks.find();`: This line retrieves the results from the "SE_HighestMarks" collection, which now contains the maximum aggregate marks for each department for students with the "class" value equal to "SE" (presumably "SE" represents a specific class or program).

3. Find Average marks of students of “BE” class department-wise.

```
var mapFunction = function() {
  if (this.class === "BE") {
    emit(this.dept, { count: 1, total_marks: this.aggregate_marks });
  }
};

var reduceFunction = function(key, values) {
  var reducedVal = { count: 0, total_marks: 0 };
  for (var i = 0; i < values.length; i++) {
    reducedVal.count += values[i].count;
    reducedVal.total_marks += values[i].total_marks;
  }
  return reducedVal;
};

var finalizeFunction = function(key, reducedVal) {
  return reducedVal.total_marks / reducedVal.count;
};

db.Student.mapReduce(mapFunction, reduceFunction, {
  out: { merge: "BE_AvgMarks" },
  finalize: finalizeFunction
});

db.BE_AvgMarks.find();
```

This code demonstrates the use of map-reduce in MongoDB to calculate the average marks for students with the "class" value equal to "BE" (presumably representing a specific class or program) for each department. Let's break it down step by step:

1. **Map Function**:
    - `var mapFunction = function() { ... }`: This is the JavaScript function that defines the "map" part of the map-reduce process.
    - `if (this.class === "BE") { ... }`: This condition checks if the "class" field of the current document is equal to "BE."
    - `emit(this.dept, { count: 1, total_marks: this.aggregate_marks });`: If the condition is met, it emits (outputs) a key-value pair. The "dept" field is used as the key, and an object with "count" (set to 1) and "total_marks" (containing the aggregate marks) is used as the value. This will help in calculating the average.
2. **Reduce Function**:
    - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.

- It initializes a `reducedVal` object with `count` and `total_marks` set to 0.
- It iterates through the array of values, which are objects containing "count" and "total_marks," and sums these values to calculate the total count and total marks for the department.

3. **Finalize Function**:
   - `var finalizeFunction = function(key, reducedVal) { ... }`: This is an optional function that is applied to the reduced values after the reduce step.
   - It calculates the average marks by dividing the "total_marks" by the "count" for each department.
4. **Map-Reduce Operation**:
   - `db.Student.mapReduce(mapFunction, reduceFunction, { out: { merge: "BE_AvgMarks" }, finalize: finalizeFunction });`: This line initiates the map-reduce operation on the "Student" collection. It uses the map function to map the data, the reduce function to reduce it, and the finalize function to calculate the average marks. The results are stored in a new collection named "BE_AvgMarks."
5. **Result Retrieval**:
   - `db.BE_AvgMarks.find();`: This line retrieves the results from the "BE_AvgMarks" collection, which now contains the calculated average marks for each department for students with the "class" value equal to "BE."

What is MapReduce?

Map-Reduce in MongoDB is a data processing technique that allows you to perform complex data analysis and transformations on the data stored in your MongoDB collections. It can be thought of as a way to extract and process data in a distributed and parallel manner.

Here's a simplified explanation in easy words:

1. **Map**: In the "Map" step, you define a JavaScript function (the "map" function) that processes each document in your collection and extracts the data you want. This function takes each document, does something with it, and emits (outputs) a key-value pair for each document. Think of it as a way to extract specific information from each document.
2. **Shuffle and Sort**: MongoDB takes all the key-value pairs emitted by the "map" function, groups them by key, and sorts them. This is done to prepare the data for the "reduce" step.
3. **Reduce**: In the "Reduce" step, you define another JavaScript function (the "reduce" function) that takes all the values associated with a specific key and performs some aggregation or calculation on those values. For example, you can calculate totals, averages, or any other summary statistics for the grouped data.
4. **Finalize (Optional)**: You can also include a "finalize" function to perform additional processing on the output of the "reduce" step.
5. **Result**: The final result is a new collection with the aggregated or processed data.

In simple terms, think of "map" as the step where you pick the information you want from each document, "reduce" as the step where you crunch and combine that information for each key, and then you have your processed data.

Map-Reduce is particularly useful when you need to perform more complex data analysis tasks on your MongoDB data that can't be achieved through standard queries or the aggregation framework. However, it's important to note that for many use cases, MongoDB's aggregation framework is preferred because it's more efficient and easier to work with. Map-Reduce is considered a lower-level and more powerful, but less user-friendly, data processing tool.

Problem Statement 15 (Map Reduce using MongoDB)

Create Book Collection with (Title, Author_name, Borrowed_status) as fields. Write Map Reduce Functions for following requirements.

```
db.createCollection("Book")

db.Book.insertMany([
  {
    Title: "Book 1",
    Author_name: "Author A",
    Borrowed_status: false,
    Price: 250
  },
  {
    Title: "Book 2",
    Author_name: "Author A",
    Borrowed_status: true,
    Price: 350
  },
  {
    Title: "Book 3",
    Author_name: "Author B",
    Borrowed_status: true,
    Price: 400
  },
  {
    Title: "Book 4",
    Author_name: "Author C",
    Borrowed_status: true,
    Price: 200
  },
  {
    Title: "Book 5",
    Author_name: "Author B",
    Borrowed_status: false,
    Price: 500
  }
])
```

1. Display Author wise list of books.

```
var mapFunction = function() {
  emit(this.Author_name, { books: [this.Title] });
};

var reduceFunction = function(key, values) {
  var reducedVal = { books: [] };
  values.forEach(function(value) {
    reducedVal.books = reducedVal.books.concat(value.books);
  });
  return reducedVal;
};

db.Book.mapReduce(mapFunction, reduceFunction, { out: "AuthorWiseBooks" });
db.AuthorWiseBooks.find();
```

This code demonstrates how to use the map-reduce functionality in MongoDB to organize a collection of books into an "AuthorWiseBooks" collection where each author is associated with a list of their books. Here's a step-by-step breakdown:

1. **Map Function**:
   - `var mapFunction = function() { ... }`: This is the JavaScript function that defines the "map" part of the map-reduce process.
   - `emit(this.Author_name, { books: [this.Title] });`: The map function emits (outputs) key-value pairs based on the "Author_name" field. The key is the author's name, and the value is an object with a "books" field that contains an array with the book's title.
2. **Reduce Function**:
   - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.
   - It initializes a `reducedVal` object with an empty "books" array.
   - It iterates through the array of values and concatenates all the book titles into the "books" array for the same author.

3. **Map-Reduce Operation**:
   - `db.Book.mapReduce(mapFunction, reduceFunction, { out: "AuthorWiseBooks" });`: This line initiates the map-reduce operation on the "Book" collection. It uses the map function to group books by author and the reduce function to concatenate the book titles. The results are stored in a new collection named "AuthorWiseBooks."
4. **Result Retrieval**:
   - `db.AuthorWiseBooks.find();`: This line retrieves the results from the "AuthorWiseBooks" collection, which now contains author names as keys, and for each author, a list of their books.

2. Display Author wise list of books having Borrowed status as “True”.

```
var mapFunction = function() {
  if (this.Borrowed_status === true) {
    emit(this.Author_name, { books: [this.Title] });
  }
};

var reduceFunction = function(key, values) {
  var reducedVal = { books: [] };
  values.forEach(function(value) {
    reducedVal.books = reducedVal.books.concat(value.books);
  });
  return reducedVal;
};

db.Book.mapReduce(mapFunction, reduceFunction, { out:
"AuthorWiseBorrowedBooks" });
db.AuthorWiseBorrowedBooks.find();
```

This code uses the map-reduce functionality in MongoDB to organize a collection of books into an "AuthorWiseBorrowedBooks" collection, where each author is associated with a list of books that have a "Borrowed_status" set to `true`. Here's a step-by-step explanation:

1. **Map Function**:
    - `var mapFunction = function() { ... }`: This is the JavaScript function that defines the "map" part of the map-reduce process.
    - `if (this.Borrowed_status === true) { ... }`: This condition checks if the "Borrowed_status" field of the current document is `true`.
    - `emit(this.Author_name, { books: [this.Title] });`: If the condition is met, it emits a key-value pair. The key is the author's name, and the value is an object with a "books" field that contains an array with the book's title.
2. **Reduce Function**:
    - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.
    - It initializes a `reducedVal` object with an empty "books" array.
    - It iterates through the array of values and concatenates all the book titles into the "books" array for the same author.
3. **Map-Reduce Operation**:
    - `db.Book.mapReduce(mapFunction, reduceFunction, { out: "AuthorWiseBorrowedBooks" });`: This line initiates the map-reduce operation on the "Book" collection. It uses the map function to group books by author and the reduce function to concatenate the book titles. The results are stored in a new collection named "AuthorWiseBorrowedBooks."
4. **Result Retrieval**:
    - `db.AuthorWiseBorrowedBooks.find();`: This line retrieves the results from the "AuthorWiseBorrowedBooks" collection, which now contains author names as keys, and for each author, a list of books with "Borrowed_status" set to `true`.

3. Display Author wise list of books having price greater than 300.

```
var mapFunction = function() {
  if (this.Price > 300) {
    emit(this.Author_name, { books: [this.Title] });
  }
};

var reduceFunction = function(key, values) {
  var reducedVal = { books: [] };
  values.forEach(function(value) {
    reducedVal.books = reducedVal.books.concat(value.books);
  });
  return reducedVal;
};

db.Book.mapReduce(mapFunction, reduceFunction, { out:
"AuthorWiseExpensiveBooks" });
db.AuthorWiseExpensiveBooks.find();
```

This code uses the map-reduce functionality in MongoDB to organize a collection of books into an "AuthorWiseExpensiveBooks" collection, where each author is associated with a list of books that have a "Price" greater than 300. Here's a step-by-step explanation:

1. **Map Function**:
   - `var mapFunction = function() { ... }`: This is the JavaScript function that defines the "map" part of the map-reduce process.
   - `if (this.Price > 300) { ... }`: This condition checks if the "Price" field of the current document is greater than 300.
   - `emit(this.Author_name, { books: [this.Title] });`: If the condition is met, it emits a key-value pair. The key is the author's name, and the value is an object with a "books" field that contains an array with the book's title.
2. **Reduce Function**:
   - `var reduceFunction = function(key, values) { ... }`: This is the "reduce" part of the map-reduce process.

    - It initializes a `reducedVal` object with an empty "books" array.
    - It iterates through the array of values and concatenates all the book titles into the "books" array for the same author.
3. **Map-Reduce Operation**:
    - `db.Book.mapReduce(mapFunction, reduceFunction, { out: "AuthorWiseExpensiveBooks" });`: This line initiates the map-reduce operation on the "Book" collection. It uses the map function to group books by author and the reduce function to concatenate the book titles. The results are stored in a new collection named "AuthorWiseExpensiveBooks."
4. **Result Retrieval**:
    - `db.AuthorWiseExpensiveBooks.find();`: This line retrieves the results from the "AuthorWiseExpensiveBooks" collection, which now contains author names as keys, and for each author, a list of books with a "Price" greater than 300.

In summary, this code uses map-reduce to organize a collection of books by author, but only for books with a price greater than 300. The result is a new collection named "AuthorWiseExpensiveBooks" where each author is associated with a list of expensive books they have authored.

Problem Statement 2

Implement MYSQL database connectivity with PHP.

Set up the database: Create a MySQL database and a table to store your data. Here's a SQL schema example for a "products" table:

```
CREATE DATABASE crud_example;

USE crud_example;


CREATE TABLE products (

    id INT AUTO_INCREMENT PRIMARY KEY,

    name VARCHAR(255) NOT NULL,

    description TEXT,

    price DECIMAL(10, 2) NOT NULL

);
```

Create the PHP files:

a. **config.php**: Set up your database connection.

```
<?php

$host = "localhost";

$username = "your_username";

$password = "your_password";

$database = "crud_example";


$connection = new mysqli($host, $username, $password,
$database);


if ($connection->connect_error) {

    die("Connection failed: " . $connection-
>connect_error);
}
?>
```

b. **index.php**: Display a list of products.

```
<?php
include("config.php");

$query = "SELECT * FROM products";
$result = $connection->query($query);
?>

<!DOCTYPE html>
<html>
<head>
    <title>CRUD Example</title>
</head>
<body>
    <h1>Product List</h1>
    <a href="create.php">Add Product</a>

    <table>
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Description</th>
            <th>Price</th>
            <th>Actions</th>
        </tr>
        <?php while ($row = $result->fetch_assoc()) { ?>
            <tr>
                <td><?= $row['id'] ?></td>
                <td><?= $row['name'] ?></td>
                <td><?= $row['description'] ?></td>
                <td><?= $row['price'] ?></td>
                <td>
                    <a href="edit.php?id=<?=
$row['id'] ?>">Edit</a>
                    <a href="delete.php?id=<?=
$row['id'] ?>">Delete</a>
                </td>
            </tr>
        <?php } ?>
    </table>
</body>
</html>
```

c. **create.php**: Create a new product record.

```
<?php
include("config.php");

if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $name = $_POST["name"];
    $description = $_POST["description"];
    $price = $_POST["price"];

    $query = "INSERT INTO products (name, description,
price) VALUES ('$name', '$description', $price)";
```

```
    $connection->query($query);
    header("Location: index.php");
}
?>

<!DOCTYPE html>
<html>
<head>
    <title>Create Product</title>
</head>
<body>
    <h1>Create Product</h1>
    <a href="index.php">Back to List</a>

    <form method="post">
        <label for="name">Name:</label>
        <input type="text" name="name" required><br>

        <label for="description">Description:</label>
        <textarea name="description"></textarea><br>

        <label for="price">Price:</label>
        <input type="number" step="0.01" name="price"
required><br>

        <input type="submit" value="Create">
    </form>
</body>
```

```
</html>
```

d. **edit.php**: Edit an existing product record.

```
<?php
include("config.php");

if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $id = $_POST["id"];
    $name = $_POST["name"];
    $description = $_POST["description"];
    $price = $_POST["price"];

    $query = "UPDATE products SET name='$name',
description='$description', price=$price WHERE id=$id";
    $connection->query($query);
    header("Location: index.php");
} else {
    $id = $_GET["id"];
    $query = "SELECT * FROM products WHERE id=$id";
    $result = $connection->query($query);
    $row = $result->fetch_assoc();
}
?>

<!DOCTYPE html>
<html>
```

```
<head>
    <title>Edit Product</title>
</head>
<body>
    <h1>Edit Product</h1>
    <a href="index.php">Back to List</a>

    <form method="post">
        <input type="hidden" name="id" value="<?=
$row['id'] ?>">
        <label for="name">Name:</label>
        <input type="text" name="name" value="<?=
$row['name'] ?>" required><br>

        <label for="description">Description:</label>
        <textarea name="description"><?=
$row['description'] ?></textarea><br>

        <label for="price">Price:</label>
        <input type="number" step="0.01" name="price"
value="<?= $row['price'] ?>" required><br>

        <input type="submit" value="Update">
    </form>
</body>
</html>
```

e. **delete.php**: Delete a product record.

```
<?php
```

```
include("config.php");


if ($_SERVER["REQUEST_METHOD"] == "GET" &&
isset($_GET["id"])) {
    $id = $_GET["id"];
    $query = "DELETE FROM products WHERE id=$id";
    $connection->query($query);
}


header("Location: index.php");
?>
```

Remember to customize these files to match your database and entity structure. This is a basic example, and you may want to enhance it by adding validation, authentication, and error handling. Also, consider using prepared statements to prevent SQL injection.

Problem Statement 6

Implement MongoDB database connectivity with Java.

Add MongoDB Java Driver Dependency

For Maven, add the following dependency to your pom.xml:

```
<dependency>
   <groupId>org.mongodb</groupId>
   <artifactId>mongo-java-driver</artifactId>
   <version>3.8.1</version> <!-- Use the appropriate version -->
</dependency>
```

```
import com.mongodb.client.*;
import org.bson.Document;
import java.util.Scanner;

public class MongoDBCRUDMenu {
   public static void main(String[] args) {
      MongoClient mongoClient =
MongoClients.create("mongodb://localhost:27017");
      MongoDatabase database = mongoClient.getDatabase("mydb");
      MongoCollection<Document> collection =
database.getCollection("mycollection");

      Scanner scanner = new Scanner(System.in);

      int choice;
      do {
         System.out.println("Menu:");
         System.out.println("1. Create (Insert) Document");
         System.out.println("2. Read (Find) Document");
         System.out.println("3. Update (Edit) Document");
         System.out.println("4. Delete Document");
         System.out.println("5. Exit");
         System.out.print("Enter your choice: ");
         choice = scanner.nextInt();

         switch (choice) {
            case 1:
               // Create (Insert) Document
               System.out.print("Enter name: ");
               String name = scanner.next();
               System.out.print("Enter age: ");
```

```
                int age = scanner.nextInt();
                System.out.print("Enter city: ");
                String city = scanner.next();

                Document newDocument = new Document("name", name)
                        .append("age", age)
                        .append("city", city);

                collection.insertOne(newDocument);
                System.out.println("Document inserted successfully.");
                break;

            case 2:
                // Read (Find) Document
                System.out.print("Enter name to search: ");
                String searchName = scanner.next();

                Document searchQuery = new Document("name", searchName);
                FindIterable<Document> foundDocuments =
collection.find(searchQuery);

                for (Document document : foundDocuments) {
                    System.out.println(document);
                }
                break;

            case 3:
                // Update (Edit) Document
                System.out.print("Enter name to update: ");
                String updateName = scanner.next();

                Document updateQuery = new Document("name", updateName);

                System.out.print("Enter new age: ");
                int newAge = scanner.nextInt();
                System.out.print("Enter new city: ");
                String newCity = scanner.next();

                Document updateData = new Document("age", newAge)
                        .append("city", newCity);

                collection.updateOne(updateQuery, new Document("$set",
updateData));
                System.out.println("Document updated successfully.");
```

```
                    break;

                case 4:
                    // Delete Document
                    System.out.print("Enter name to delete: ");
                    String deleteName = scanner.next();

                    Document deleteQuery = new Document("name", deleteName);
                    collection.deleteOne(deleteQuery);
                    System.out.println("Document deleted successfully.");
                    break;

                case 5:
                    // Exit the program
                    System.out.println("Exiting the program.");
                    break;

                default:
                    System.out.println("Invalid choice. Please try again.");
                    break;
            }

        } while (choice != 5);

        mongoClient.close();
        scanner.close();
    }
}
```