Practical no-1

Q1. Create the table as workers and the details are:

Sr.	Name	Designation	BranchNo
1	Ram	Manager	Chennai
2	Santosh	Supervisor	Goa
3	Hari	Assistant	Delhi
4	Jagdesh	Peon	Mumbai

Perform the following queries:

1. Alter the table by adding a column Salary.

2. Alter the table by modifying the column Name.

3. Describe the table employee.

4. Copy the table employee as emp.

5. Truncate the table

6. Delete the second row from the table

7. Drop the table.

Code:-

1. Create the `workers` table

```
CREATE TABLE workers (
    Sr INT,
    Name VARCHAR(50),
    Designation VARCHAR(50),
    BranchNo VARCHAR(50)
);

INSERT INTO workers (Sr, Name, Designation, BranchNo)
VALUES
    (1, 'Ram', 'Manager', 'Chennai'),
    (2, 'Santosh', 'Supervisor', 'Goa'),
    (3, 'Hari', 'Assistant', 'Delhi'),
    (4, 'Jagdesh', 'Peon', 'Mumbai');
```

2. Alter the table by adding a column `Salary`

```
ALTER TABLE workers
ADD Salary DECIMAL(10, 2);
```

3. Alter the table by modifying the column `Name`

For instance, to increase the size of the Name column:

```
ALTER TABLE workers
MODIFY Name VARCHAR(100);
```

4. Describe the table `workers`

```
DESCRIBE workers;
```

5. Copy the table `workers` as `emp`

```
CREATE TABLE emp AS
SELECT * FROM workers;
```

6. Truncate the table

```
TRUNCATE TABLE workers;
```

7. Delete the second row from the table

If you've reinserted data after truncating:

```
DELETE FROM workers
WHERE Sr = 2;
```

8. Drop the table

```
DROP TABLE workers;
```

Practical no-2

Q2. Createthefollowingtables.AndSolvefollowingqueriesbySQL

🕐 Deposit (actno,cname,bname,amount,adate)

🕐 Branch (bname,city)

🕐 Customers (cname, city)

🕐 Borrow(loanno,cname,bname, amount)

- Add primary key and foreign key wherever applicable.

- Insert data into the above created tables.

Perform the following queries:

1. Display names of depositors having amount greater than 4000.

2. Display account date of customers Anil

3. Display account no. and deposit amount of customers having account opened between dates1-12-96 and 1-5-97

4. Find the names of all branches where the average account balance is more than 1,200.

5. Delete depositors having deposit less than 5000

6. Create a view on deposit table.

Code:-

Table Creation and Data Insertion

1. Create `Deposit` Table

```
CREATE TABLE Deposit (
    actno INT PRIMARY KEY,
    cname VARCHAR(50),
    bname VARCHAR(50),
    amount DECIMAL(10, 2),
    adate DATE,
    FOREIGN KEY (cname) REFERENCES Customers(cname),
    FOREIGN KEY (bname) REFERENCES Branch(bname)
);
```

2. Create `Branch` Table

```
CREATE TABLE Branch (
    bname VARCHAR(50) PRIMARY KEY,
    city VARCHAR(50)
);
```

3. Create `Customers` Table

```
CREATE TABLE Customers (
    cname VARCHAR(50) PRIMARY KEY,
    city VARCHAR(50)
);
```

4. Create `Borrow` Table

```
CREATE TABLE Borrow (
    loanno INT PRIMARY KEY,
    cname VARCHAR(50),
    bname VARCHAR(50),
    amount DECIMAL(10, 2),
    FOREIGN KEY (cname) REFERENCES Customers(cname),
    FOREIGN KEY (bname) REFERENCES Branch(bname)
);
```

Inserting Data into Tables

You can insert sample data into these tables as follows:

```
-- Insert into Branch
INSERT INTO Branch (bname, city) VALUES
    ('Branch1', 'Mumbai'),
    ('Branch2', 'Delhi'),
    ('Branch3', 'Goa');

-- Insert into Customers
INSERT INTO Customers (cname, city) VALUES
    ('Anil', 'Mumbai'),
    ('Sunita', 'Delhi'),
    ('Ravi', 'Goa');

-- Insert into Deposit
INSERT INTO Deposit (actno, cname, bname, amount, adate) VALUES
    (101, 'Anil', 'Branch1', 5000, '1996-12-10'),
    (102, 'Sunita', 'Branch2', 3000, '1997-02-15'),
    (103, 'Ravi', 'Branch3', 6000, '1997-04-20');

-- Insert into Borrow
INSERT INTO Borrow (loanno, cname, bname, amount) VALUES
    (201, 'Anil', 'Branch1', 1500),
    (202, 'Sunita', 'Branch2', 2500),
    (203, 'Ravi', 'Branch3', 3500);
```

Queries

1. **Display names of depositors having amount greater than 4000**

   ```
   SELECT cname
   FROM Deposit
   WHERE amount > 4000;
   ```

2. **Display account date of customers named Anil**

   ```
   SELECT adate
   FROM Deposit
   WHERE cname = 'Anil';
   ```

3. **Display account number and deposit amount of customers with accounts opened between '1-12-96' and '1-5-97'**

```
SELECT actno, amount
FROM Deposit
WHERE adate BETWEEN '1996-12-01' AND '1997-05-01';
```

4. **Find the names of all branches where the average account balance is more than 1,200**

```
SELECT bname
FROM Deposit
GROUP BY bname
HAVING AVG(amount) > 1200;
```

5. **Delete depositors having deposits less than 5000**

```
DELETE FROM Deposit
WHERE amount < 5000;
```

6. **Create a view on the `Deposit` table**

```
CREATE VIEW Deposit_View AS
SELECT actno, cname, bname, amount, adate
FROM Deposit;
```

Practical no-3

Q3. Create a database called Company consist of the following tables.

1.Emp(eno,ename,job,hiredate,salary, commission, deptno)

2.dept(deptno, deptname, location)

- eno is primary key in emp

- deptno is primary key in dept

PerformtheQueriesbySQL (At list 6 query)

1. List the maximum salary paid to salesman

2. List name of emp whose name start with „I"

3. List details of emp who have joined before "30-sept-81"

4. List the emp details in the of their basic salary

5. List of no. of emp & avg salary for emp in the dept no "20"

6. List the avg salary, minimum salary of the emp hiredatewise for dept no „10".

7. List emp name and its department

8. List total salary paid to each department

9. List details of employee working in "Development" department

10. Update salary of all employees in deptno 10 by 5 %.

Code:-

Database Creation and Table Structure

1. Create the `Company` Database

```
CREATE DATABASE Company;
USE Company;
```

2. Create the `Emp` Table

```
CREATE TABLE Emp (
    eno INT PRIMARY KEY,
    ename VARCHAR(50),
    job VARCHAR(50),
    hiredate DATE,
    salary DECIMAL(10, 2),
    commission DECIMAL(10, 2),
    deptno INT,
    FOREIGN KEY (deptno) REFERENCES Dept(deptno)
);
```

3. Create the `Dept` Table

```
CREATE TABLE Dept (
    deptno INT PRIMARY KEY,
    deptname VARCHAR(50),
    location VARCHAR(50)
);
```

Sample Data Insertion

```
-- Insert sample data into Dept table
INSERT INTO Dept (deptno, deptname, location) VALUES
    (10, 'Accounting', 'New York'),
    (20, 'Research', 'Dallas'),
    (30, 'Sales', 'Chicago'),
    (40, 'Development', 'Boston');

-- Insert sample data into Emp table
INSERT INTO Emp (eno, ename, job, hiredate, salary, commission, deptno) VALUES
    (1, 'Alice', 'Manager', '1980-06-09', 5000, 500, 10),
    (2, 'Bob', 'Salesman', '1981-07-01', 3000, 300, 30),
    (3, 'Charlie', 'Analyst', '1981-09-20', 4000, NULL, 20),
    (4, 'Irene', 'Clerk', '1980-03-15', 1500, NULL, 20),
    (5, 'Ian', 'Salesman', '1983-12-17', 2500, 200, 30),
    (6, 'Emma', 'Developer', '1982-11-11', 2800, NULL, 40);
```

Queries

1. **List the maximum salary paid to salesmen**

   ```
   SELECT MAX(salary) AS MaxSalary
   FROM Emp
   WHERE job = 'Salesman';
   ```

2. **List the names of employees whose names start with "I"**

   ```
   SELECT ename
   FROM Emp
   WHERE ename LIKE 'I%';
   ```

3. **List details of employees who joined before '30-sept-81'**

   ```
   SELECT *
   FROM Emp
   WHERE hiredate < '1981-09-30';
   ```

4. **List employee details ordered by their basic salary**

   ```
   SELECT *
   FROM Emp
   ORDER BY salary;
   ```

5. **List the number of employees and average salary for employees in dept no "20"**

   ```
   SELECT COUNT(eno) AS NumEmployees, AVG(salary) AS AvgSalary
   FROM Emp
   WHERE deptno = 20;
   ```

6. **List the average salary and minimum salary of employees hiredate-wise for dept no "10"**

```
SELECT hiredate, AVG(salary) AS AvgSalary, MIN(salary) AS MinSalary
FROM Emp
WHERE deptno = 10
GROUP BY hiredate;
```

7. **List employee names and their departments**

```
SELECT Emp.ename, Dept.deptname
FROM Emp
JOIN Dept ON Emp.deptno = Dept.deptno;
```

8. **List the total salary paid to each department**

```
SELECT Dept.deptname, SUM(Emp.salary) AS TotalSalary
FROM Emp
JOIN Dept ON Emp.deptno = Dept.deptno
GROUP BY Dept.deptname;
```

9. **List details of employees working in the "Development" department**

```
SELECT Emp.*
FROM Emp
JOIN Dept ON Emp.deptno = Dept.deptno
WHERE Dept.deptname = 'Development';
```

10. **Update salary of all employees in deptno 10 by 5%**

```
UPDATE Emp
SET salary = salary * 1.05
WHERE deptno = 10;
```

Practical no-4

Q4. Perform Following

1. Create a table Employee with fields-

🕐 Eno - Primary key and apply sequence starts with 101.

🕐 Ename – not null

🕐 Address – default „Nashik“

🕐 Join_Date

🕐 Post

🕐 Salary -check>5000

2. Create another table Emp_Project with field-

🕐 Eno – Foriegn key

🕐 Project_name

🕐 LOC(i.e Line of code)

3. Create index on Ename field of employee table.

4. Create View on Employee table to show only Employee name, its post and salary.

Perform the following querries:

1. To find the employeename whose salary is greater than or equal to 6000.

2. Find Employee name whose salary is 3000.

3. Find name and Join_Date from table having salary<5000.

Code:-

Table Creation

1. Create the `Employee` Table

- `Eno` as the primary key with a sequence starting at 101.
- `Ename` as a NOT NULL field.
- `Address` with a default value of "Nashik."
- `Salary` with a check constraint ensuring it is greater than 5000.

```
CREATE TABLE Employee (
    Eno INT PRIMARY KEY AUTO_INCREMENT,
    Ename VARCHAR(50) NOT NULL,
    Address VARCHAR(100) DEFAULT 'Nashik',
    Join_Date DATE,
    Post VARCHAR(50),
    Salary DECIMAL(10, 2) CHECK (Salary > 5000)
);
```

```
-- Set the initial value of Eno to start from 101
ALTER TABLE Employee AUTO_INCREMENT = 101;
```

2. Create the `Emp_Project` Table

- `Eno` as a foreign key referencing the `Employee` table.
- `Project_name` and `LOC` (Line of Code).

```
CREATE TABLE Emp_Project (
    Eno INT,
    Project_name VARCHAR(100),
    LOC INT,
    FOREIGN KEY (Eno) REFERENCES Employee(Eno)
);
```

3. Create Index on `Ename` Field of `Employee` Table

```
CREATE INDEX idx_ename ON Employee(Ename);
```

4. Create a View on the `Employee` Table

- This view will display only `Ename`, `Post`, and `Salary`.

```
CREATE VIEW Employee_View AS
SELECT Ename, Post, Salary
FROM Employee;
```

Queries

1. **Find the employee names whose salary is greater than or equal to 6000**

   ```
   SELECT Ename
   FROM Employee
   WHERE Salary >= 6000;
   ```

2. **Find the employee names whose salary is 3000**

   ```
   SELECT Ename
   FROM Employee
   WHERE Salary = 3000;
   ```

3. **Find name and `Join_Date` from the table where salary is less than 5000**

   ```
   SELECT Ename, Join_Date
   FROM Employee
   WHERE Salary < 5000;
   ```

Practical no-5

Q5. Consider the following two tables:

a) Persons table is as follow:

Pid	First name	Last name	Address	City
1	Amit	Tiwari	kharadi	Pune
2	Suresh	Mishra	gangapur	Nashik
3	Shalu	Choudhary	savedi	Ahmednagar

a) Order Table is as follows:

O_id	Order_Date	Order_Price	P_id
1	2023-02-11	1000	3
2	2023-03-12	1600	3
3	2023-04-10	700	1
4	2022-07-24	300	1
5	2022-08-30	2000	NULL

a. Apply inner Join on above tables and show the result

b. Apply Left Join (or left outer join) on above tables and show the result.

c. Apply Right Join (or right outer join) on above tables and show the result.

Code:-

Table Creation and Sample Data Insertion

```
-- Creating the Persons table
CREATE TABLE Persons (
    Pid INT PRIMARY KEY,
    LastName VARCHAR(50),
    FirstName VARCHAR(50),
    Address VARCHAR(100),
    City VARCHAR(50)
);

-- Inserting data into Persons table
INSERT INTO Persons (Pid, LastName, FirstName, Address, City) VALUES
    (1, 'Tiwari', 'Amit', 'kharadi', 'Pune'),
    (2, 'Mishra', 'Suresh', 'gangapur', 'Nashik'),
    (3, 'Choudhary', 'Shalu', 'savedi', 'Ahmednagar');

-- Creating the Order table
CREATE TABLE Orders (
    O_id INT PRIMARY KEY,
```

```
    Order_Date DATE,
    Order_Price DECIMAL(10, 2),
    P_id INT
);

-- Inserting data into Order table
INSERT INTO Orders (O_id, Order_Date, Order_Price, P_id) VALUES
    (1, '2023-02-11', 1000, 3),
    (2, '2023-03-12', 1600, 3),
    (3, '2023-04-10', 700, 1),
    (4, '2022-07-24', 300, 1),
    (5, '2022-08-30', 2000, NULL);
```

Queries with Different Joins

1. **Inner Join**: Shows records with matching `Pid` in both `Persons` and `Orders` tables.

   ```
   SELECT Persons.Pid, Persons.LastName, Persons.FirstName, Orders.O_id,
   Orders.Order_Date, Orders.Order_Price
   FROM Persons
   INNER JOIN Orders ON Persons.Pid = Orders.P_id;
   ```

   This query will only return rows where `Pid` from `Persons` matches `P_id` from `Orders`.

2. **Left Join (Left Outer Join)**: Returns all records from the `Persons` table and matching records from the `Orders` table. If there's no match, `NULL` values will appear for columns from `Orders`.

   ```
   SELECT Persons.Pid, Persons.LastName, Persons.FirstName, Orders.O_id,
   Orders.Order_Date, Orders.Order_Price
   FROM Persons
   LEFT JOIN Orders ON Persons.Pid = Orders.P_id;
   ```

   This query includes all records from `Persons`, even if they don't have corresponding orders.

3. **Right Join (Right Outer Join)**: Returns all records from the `Orders` table and matching records from the `Persons` table. If there's no match, `NULL` values will appear for columns from `Persons`.

   ```
   SELECT Persons.Pid, Persons.LastName, Persons.FirstName, Orders.O_id,
   Orders.Order_Date, Orders.Order_Price
   FROM Persons
   RIGHT JOIN Orders ON Persons.Pid = Orders.P_id;
   ```

   This query includes all records from `Orders`, even if there's no corresponding person in `Persons`.

Practical no-6

Q6.Consider the following two tables:

a. Stud_Info Table is as follows

Rno	Name	Address
1	Abhay	Nagar
2	Sarika	Nashik
3	Riya	Pune
4	Sachin	Mumbai

b. Stud_Marks Table is as follows

Rno	DSD	IOT
1	50	45
2	67	65
3	76	55
4	70	50

a. Apply inner Join on above tables and show the result

b. Apply Left Join (or left outer join) on above tables and show the result.

c. Apply Right Join (or right outer join) on above tables and show the result.

Code:-

Table Creation and Data Insertion

```
sql
Copy code
-- Creating Stud_Info Table
CREATE TABLE Stud_Info (
    Rno INT PRIMARY KEY,
    Name VARCHAR(50),
    Address VARCHAR(50)
);

-- Inserting data into Stud_Info Table
INSERT INTO Stud_Info (Rno, Name, Address) VALUES
    (1, 'Abhay', 'Nagar'),
```

```
    (2, 'Sarika', 'Nashik'),
    (3, 'Riya', 'Pune'),
    (4, 'Sachin', 'Mumbai');

-- Creating Stud_Marks Table
CREATE TABLE Stud_Marks (
    Rno INT PRIMARY KEY,
    DSD INT,
    IOT INT
);

-- Inserting data into Stud_Marks Table
INSERT INTO Stud_Marks (Rno, DSD, IOT) VALUES
    (1, 50, 45),
    (2, 67, 65),
    (3, 76, 55),
    (5, 70, 50);
```

Queries with Different Joins

1. **Inner Join**: Shows records with matching `Rno` in both `Stud_Info` and `Stud_Marks` tables.

   ```
   SELECT Stud_Info.Rno, Stud_Info.Name, Stud_Info.Address, Stud_Marks.DSD,
   Stud_Marks.IOT
   FROM Stud_Info
   INNER JOIN Stud_Marks ON Stud_Info.Rno = Stud_Marks.Rno;
   ```

   This query will return only students who have records in both tables (`Rno` in both `Stud_Info` and `Stud_Marks`).

2. **Left Join (Left Outer Join)**: Returns all records from the `Stud_Info` table and matching records from the `Stud_Marks` table. If there's no match, `NULL` values will appear for columns from `Stud_Marks`.

   ```
   SELECT Stud_Info.Rno, Stud_Info.Name, Stud_Info.Address, Stud_Marks.DSD,
   Stud_Marks.IOT
   FROM Stud_Info
   LEFT JOIN Stud_Marks ON Stud_Info.Rno = Stud_Marks.Rno;
   ```

   This query includes all records from `Stud_Info`, even if they don't have corresponding marks in `Stud_Marks`.

3. **Right Join (Right Outer Join)**: Returns all records from the `Stud_Marks` table and matching records from the `Stud_Info` table. If there's no match, `NULL` values will appear for columns from `Stud_Info`.

   ```
   SELECT Stud_Info.Rno, Stud_Info.Name, Stud_Info.Address, Stud_Marks.DSD,
   Stud_Marks.IOT
   FROM Stud_Info
   RIGHT JOIN Stud_Marks ON Stud_Info.Rno = Stud_Marks.Rno;
   ```

   This query includes all records from `Stud_Marks`, even if there's no corresponding student in `Stud_Info`.

Practical no-7

Q7. Implement Unnamed PL/SQLcode block: Use of Control structure and Exception handling is mandatory.

Consider Tables:

1. Borrower(Roll_no, Name, Date of Issue, Name of Book, Status)

2. Fine(Roll_no, Date, Amt)

🕐 Accept Roll_no and Name of Book from user.

🕐 Check the number of days (from date of issue).

🕐 If days are between 15 to 30 then fine amount will be Rs 5per day.

🕐 If no. of days>30, per day fine will be Rs 50 per day and for days less than 30, Rs. 5 per day.

🕐 After submitting the book, status will change from I to R.

If condition of fine is true, then details will be stored into fine table.

Code:-

Below is a PL/SQL anonymous block that meets your requirements. It accepts `Roll_no` and `Name of Book` as inputs, calculates the fine based on the number of days since the issue date, and updates the `Borrower` table accordingly. If a fine is applicable, it inserts a record into the `Fine` table.

PL/SQL Code Block

```
DECLARE
    v_roll_no       INT;
    v_name_of_book  VARCHAR2(100);
    v_issue_date    DATE;
    v_status        CHAR(1);
    v_days          INT;
    v_fine_amount   INT := 0;
    v_current_date  DATE := SYSDATE;

    -- Custom exception for when no record is found
    e_no_borrower EXCEPTION;
BEGIN
    -- Accept inputs for Roll_no and Name of Book
    v_roll_no := &Roll_no;
    v_name_of_book := '&Name_of_Book';

    -- Fetch Date of Issue and Status from Borrower table for given Roll_no and
Book Name
    SELECT Date_of_Issue, Status
    INTO v_issue_date, v_status
    FROM Borrower
    WHERE Roll_no = v_roll_no
      AND Name_of_Book = v_name_of_book
      AND Status = 'I';

    -- Calculate the number of days since the book was issued
```

```
    v_days := v_current_date - v_issue_date;

    -- Determine fine based on the number of days
    IF v_days > 15 AND v_days <= 30 THEN
        v_fine_amount := (v_days - 15) * 5;
    ELSIF v_days > 30 THEN
        v_fine_amount := (30 - 15) * 5 + (v_days - 30) * 50;
    END IF;

    -- Update the Borrower table to mark the book as returned (Status 'R')
    UPDATE Borrower
    SET Status = 'R'
    WHERE Roll_no = v_roll_no
      AND Name_of_Book = v_name_of_book;

    -- If there's a fine, insert a record in the Fine table
    IF v_fine_amount > 0 THEN
        INSERT INTO Fine (Roll_no, Date, Amt)
        VALUES (v_roll_no, v_current_date, v_fine_amount);
    END IF;

    DBMS_OUTPUT.PUT_LINE('Book returned successfully.');
    DBMS_OUTPUT.PUT_LINE('Fine Amount: ' || v_fine_amount);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE e_no_borrower;
    WHEN e_no_borrower THEN
        DBMS_OUTPUT.PUT_LINE('Error: No matching borrower record found, or book
already returned.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
/
```

Explanation of the Code

1. **Input Acceptance**: The PL/SQL block takes `Roll_no` and `Name_of_Book` as input.
2. **Data Retrieval**: It retrieves the `Date_of_Issue` and `Status` of the book using `Roll_no` and `Name_of_Book` from the `Borrower` table. The `Status` is checked to ensure the book is issued (`Status = 'I'`).
3. **Date Calculation and Fine Determination**:
    - The difference in days between `SYSDATE` and `Date_of_Issue` is calculated.
    - If days are between 15 and 30, a fine of Rs. 5 per extra day is calculated.
    - For days greater than 30, Rs. 5 per day is applied up to 30 days, and Rs. 50 per day is applied thereafter.
4. **Status Update**: If the book is returned, `Status` is updated to 'R'.
5. **Fine Insertion**: If a fine is applicable, an entry is added to the `Fine` table.
6. **Exception Handling**: Handles cases where no matching record is found or other errors.

Execution

This PL/SQL code block includes both control structures and exception handling to manage errors and conditions specified in the problem statement.

Practical no-8

Q8. Implement a Named PL/SQL Block: PL/SQL Stored Procedure and Stored Function.

Write a Stored Procedure namely proc_Grade for the categorization of student. If marks scored by students in examination is <=1500 and marks>=990 then student will be placed in distinction category if marks scored are between 989 and 900 category is first class, if marks 899and 825 category is Higher Second Class.

Write a PL/SQLblock to use procedure created with above requirement.

1. Stud_Marks(name, total_marks)

2. Result(Roll,Name, Class).

Code:-

Table Creation

```
-- Creating Stud_Marks Table
CREATE TABLE Stud_Marks (
    name VARCHAR2(50),
    total_marks INT
);

-- Creating Result Table
CREATE TABLE Result (
    Roll INT PRIMARY KEY,
    Name VARCHAR2(50),
    Class VARCHAR2(30)
);
```

Stored Procedure: `proc_Grade`

The stored procedure `proc_Grade` accepts a student's `name` and `total_marks`, determines their grade based on the provided marks ranges, and updates the `Result` table with the calculated grade.

```
CREATE OR REPLACE PROCEDURE proc_Grade (
    p_name       IN VARCHAR2,
    p_total_marks IN INT
) AS
    v_class VARCHAR2(30);
BEGIN
    -- Determine category based on total marks
    IF p_total_marks >= 990 AND p_total_marks <= 1500 THEN
        v_class := 'Distinction';
    ELSIF p_total_marks BETWEEN 900 AND 989 THEN
        v_class := 'First Class';
    ELSIF p_total_marks BETWEEN 825 AND 899 THEN
        v_class := 'Higher Second Class';
    ELSE
        v_class := 'No Category';
    END IF;
```

```
    -- Update or insert into the Result table based on the Roll number
    INSERT INTO Result (Roll, Name, Class)
    VALUES ((SELECT COALESCE(MAX(Roll), 0) + 1 FROM Result), p_name, v_class);

    DBMS_OUTPUT.PUT_LINE('Student ' || p_name || ' categorized as: ' ||
v_class);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END proc_Grade;
/
```

Explanation

1. **Procedure Parameters**: `p_name` (name of student) and `p_total_marks` (total marks).
2. **Grade Calculation**:
    - If marks are between 990 and 1500, the student is categorized as “Distinction.”
    - If marks are between 900 and 989, the student is categorized as “First Class.”
    - If marks are between 825 and 899, the student is categorized as “Higher Second Class.”
    - If marks are below 825, they fall under “No Category.”
3. **Updating the Result Table**:
    - A new `Roll` number is generated, and the `name` and `Class` are inserted into the `Result` table.
4. **Exception Handling**: Captures any unexpected errors.

PL/SQL Block to Use the Procedure

```
DECLARE
    v_name       VARCHAR2(50);
    v_total_marks INT;
BEGIN
    -- Example student data to call the procedure
    v_name := 'Abhay';
    v_total_marks := 980;

    -- Call the proc_Grade procedure with the example data
    proc_Grade(v_name, v_total_marks);

    -- Additional calls as needed
    -- proc_Grade('Sarika', 1050);
    -- proc_Grade('Riya', 860);
    -- proc_Grade('Sachin', 900);
END;
/
```

Explanation of the PL/SQL Block

- **Variable Declaration**: Variables `v_name` and `v_total_marks` are declared to store example values.
- **Procedure Call**: The procedure `proc_Grade` is called with these values, categorizing each student.
- **Sample Data**: Multiple calls can be made with different names and marks to categorize various students.

Practical no-9

Q9. Implement a PL/SQL block of code using non-parameterized Cursor that will merge the data available in the newly created table N_Roll Call with the data available in the table O_RollCall. If thedata in the first table already exist in the second table then that data should be skipped. (Apply Explicit, For loop,Non- Parameterized Cursor)

code:-

In this example, we assume both tables (`N_RollCall` and `O_RollCall`) have a column `roll_no` which is unique in each table and is used to check for existing data.

```
DECLARE
    -- Define a cursor to select data from the N_RollCall table
    CURSOR n_rollcall_cur IS
        SELECT roll_no, student_name, attendance_date
        FROM N_RollCall;

    -- Define variables to hold each row's data from N_RollCall
    v_roll_no        N_RollCall.roll_no%TYPE;
    v_student_name   N_RollCall.student_name%TYPE;
    v_attendance_date N_RollCall.attendance_date%TYPE;

BEGIN
    -- Open the cursor and iterate through each record in N_RollCall
    FOR n_record IN n_rollcall_cur LOOP
        -- Check if the record already exists in O_RollCall
        BEGIN
            SELECT roll_no INTO v_roll_no
            FROM O_RollCall
            WHERE roll_no = n_record.roll_no;

            -- If no exception, it means record already exists
            -- Skip this record by continuing to the next iteration
            CONTINUE;

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                -- If NO_DATA_FOUND, it means the record does not exist in
O_RollCall
                -- Insert the record into O_RollCall
                INSERT INTO O_RollCall (roll_no, student_name, attendance_date)
                VALUES (n_record.roll_no, n_record.student_name,
n_record.attendance_date);
        END;
    END LOOP;

    -- Commit the transaction to save changes
    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Data merge completed successfully.');
END;
```

Explanation:

1. **Cursor Declaration**: The cursor `n_rollcall_cur` fetches all data from `N_RollCall`.

2. **For Loop**: The loop iterates through each record in `N_RollCall`.
3. **Data Check and Insertion**:
    - The block checks if the `roll_no` from `N_RollCall` already exists in `O_RollCall`.
    - If it exists, the `CONTINUE` statement skips the insertion.
    - If `NO_DATA_FOUND` is raised, the record does not exist in `O_RollCall`, and it proceeds with insertion.
4. **Commit**: Saves all changes made to `O_RollCall`.

Practical no-10

Q10. Implement a PLSQL Block of code using with and without parameterized cursor for displaying the inserted rows and also update the salary by 5000 of those employee salary is above 45000 and increase rest of the others by 1000.

(Apply Explicit, Loops with Parameterized cursor)

code:-

This example assumes a table called `Employee` with columns `emp_id`, `emp_name`, and `salary`.

PL/SQL Block

```
DECLARE
    -- Non-parameterized cursor to fetch all employee details
    CURSOR all_employees_cur IS
        SELECT emp_id, emp_name, salary FROM Employee;

    -- Parameterized cursor to update salaries based on the condition
    CURSOR salary_update_cur (p_salary_threshold NUMBER) IS
        SELECT emp_id, salary
        FROM Employee
        WHERE salary > p_salary_threshold;

    -- Variables to hold employee data
    v_emp_id   Employee.emp_id%TYPE;
    v_emp_name Employee.emp_name%TYPE;
    v_salary   Employee.salary%TYPE;

BEGIN
    -- Display inserted rows using the non-parameterized cursor
    DBMS_OUTPUT.PUT_LINE('List of all employees:');
    FOR employee_rec IN all_employees_cur LOOP
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || employee_rec.emp_id ||
                             ', Name: ' || employee_rec.emp_name ||
                             ', Salary: ' || employee_rec.salary);
    END LOOP;

    -- Update salary using the parameterized cursor for employees with salary >
45000
    FOR salary_rec IN salary_update_cur(45000) LOOP
        UPDATE Employee
        SET salary = salary + 5000
        WHERE emp_id = salary_rec.emp_id;
    END LOOP;

    -- Update salary for employees with salary <= 45000
    FOR employee_rec IN all_employees_cur LOOP
        IF employee_rec.salary <= 45000 THEN
            UPDATE Employee
            SET salary = salary + 1000
            WHERE emp_id = employee_rec.emp_id;
        END IF;
    END LOOP;

    -- Commit changes to save the updates
```

```
    COMMIT;

    -- Display the updated salary information
    DBMS_OUTPUT.PUT_LINE('Updated salary details:');
    FOR employee_rec IN all_employees_cur LOOP
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || employee_rec.emp_id ||
                             ', Name: ' || employee_rec.emp_name ||
                             ', Updated Salary: ' || employee_rec.salary);
    END LOOP;

END;
```

Explanation:

1. **Non-Parameterized Cursor (`all_employees_cur`)**: This cursor fetches all employees, used for displaying initial records and applying updates to employees with a salary of 45,000 or less.
2. **Parameterized Cursor (`salary_update_cur`)**: Accepts a salary threshold (45,000 in this case) and selects employees whose salaries are above this threshold.
3. **Salary Update Logic**:
    - The first loop uses the parameterized cursor to increase the salary by 5,000 for employees earning more than 45,000.
    - The second loop iterates through `all_employees_cur`, increasing the salary by 1,000 for employees with a salary of 45,000 or less.
4. **Commit**: Finalizes the changes by committing the transactions.
5. **Display Updated Information**: A loop at the end displays updated salary details for each employee.

Practical no-11

Q11. Write a before trigger for Insert, update event considering following

requirement:Emp(e_no, e_name, salary)

I) Trigger action should be initiated when salary is tried to be inserted is less than Rs. 50,000/-

II) Trigger action should be initiated when salary is tried to be updated for value less than Rs. 50,000/- Action should be rejection of update or Insert operation by displaying appropriate errormessage.

Also the new values expected to be inserted will be stored in new table Tracking (e_no, salary).

Code:-

Requirements:

- `Emp` table with columns: `e_no`, `e_name`, and `salary`.
- `Tracking` table with columns: `e_no` and `salary`.

Trigger Code

```
CREATE OR REPLACE TRIGGER trg_check_salary
BEFORE INSERT OR UPDATE ON Emp
FOR EACH ROW
DECLARE
    -- Custom exception for salary below the allowed limit
    salary_below_limit EXCEPTION;
BEGIN
    -- Check if the new salary is less than 50,000 for both INSERT and UPDATE
events
    IF :NEW.salary < 50000 THEN
        -- Insert the attempted values into the Tracking table
        INSERT INTO Tracking (e_no, salary)
        VALUES (:NEW.e_no, :NEW.salary);

        -- Raise the custom exception with an error message
        RAISE salary_below_limit;
    END IF;

EXCEPTION
    WHEN salary_below_limit THEN
        -- Raise an error with a descriptive message for salary restriction
        RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be less than Rs. 50,000.
Operation rejected.');
END;
```

Explanation:

1. **Trigger Timing and Event**: The trigger is set to execute `BEFORE INSERT OR UPDATE` on the `Emp` table.
2. **Condition Check**:
    - `:NEW.salary < 50000` checks if the new salary value is less than 50,000.
    - If the condition is met, the trigger performs two actions:
        - It logs the attempted insertion or update into the `Tracking` table.
        - It raises a custom exception `salary_below_limit`.
3. **Exception Handling**:
    - The `salary_below_limit` exception triggers an error message using `RAISE_APPLICATION_ERROR` to reject the operation, displaying a descriptive message: "Salary cannot be less than Rs. 50,000. Operation rejected."
4. **Insertion in `Tracking` Table**:
    - The `Tracking` table stores the attempted `e_no` and `salary` values for audit purposes.

Practical no-12

Q12. Create Institute Database and Create Student Collection using Mongodb.

1. RollNo
2. Name
3. Age
4. Branch
5. Address:{City, State}
6. Hobbies(Array)

Perform Following Operations (Any 8)

1. Create database institute
2. Create collection students
3. Insert 10 documents with above mentioned structure.
4. Display all students information
5. Display student‟s information whose age is greater than 15.
6. Display student information sorted on name fields.
7. Update student branch Computer of RollNo 3.
8. Remove documents with roll no 1.
9. Display student‟s information whose name starts with A.
10. Display total number of documents available in collection.
11. Display only first 5 documents.
12. Display all documents instead of 3.
13. Display the name of students who leave in Pune City.
14. Display the list of different cities from where students are coming.
15. Display the list of different cities with number of students from belonging to that city.
16. Display only name of all students.

Code:-

```
// 1. Create the database "institute"
use institute;

// 2. Create the "students" collection
db.createCollection("students");

// 3. Insert 10 documents into the "students" collection
db.students.insertMany([
    { RollNo: 1, Name: "Aarav", Age: 16, Branch: "Science", Address: { City:
"Pune", State: "Maharashtra" }, Hobbies: ["Cricket", "Reading"] },
    { RollNo: 2, Name: "Bhavya", Age: 17, Branch: "Commerce", Address: { City:
"Mumbai", State: "Maharashtra" }, Hobbies: ["Dance", "Music"] },
    { RollNo: 3, Name: "Chaitanya", Age: 18, Branch: "Science", Address: { City:
"Delhi", State: "Delhi" }, Hobbies: ["Chess", "Reading"] },
    { RollNo: 4, Name: "Diya", Age: 15, Branch: "Computer", Address: { City:
"Bangalore", State: "Karnataka" }, Hobbies: ["Painting", "Gaming"] },
    { RollNo: 5, Name: "Esha", Age: 14, Branch: "Arts", Address: { City: "Pune",
State: "Maharashtra" }, Hobbies: ["Music", "Dancing"] },
    { RollNo: 6, Name: "Farhan", Age: 19, Branch: "Science", Address: { City:
"Hyderabad", State: "Telangana" }, Hobbies: ["Reading", "Coding"] },
    { RollNo: 7, Name: "Gauri", Age: 16, Branch: "Commerce", Address: { City:
"Pune", State: "Maharashtra" }, Hobbies: ["Singing", "Sketching"] },
    { RollNo: 8, Name: "Harsh", Age: 15, Branch: "Computer", Address: { City:
"Chennai", State: "Tamil Nadu" }, Hobbies: ["Gaming", "Coding"] },
    { RollNo: 9, Name: "Isha", Age: 17, Branch: "Arts", Address: { City: "Pune",
State: "Maharashtra" }, Hobbies: ["Photography", "Traveling"] },
    { RollNo: 10, Name: "Jay", Age: 18, Branch: "Computer", Address: { City:
"Mumbai", State: "Maharashtra" }, Hobbies: ["Drawing", "Music"] }
]);

// 4. Display all students' information
db.students.find().pretty();

// 5. Display students' information whose age is greater than 15
db.students.find({ Age: { $gt: 15 } }).pretty();

// 6. Display students' information sorted by name
db.students.find().sort({ Name: 1 }).pretty();

// 7. Update branch to "Computer" for student with RollNo 3
db.students.updateOne({ RollNo: 3 }, { $set: { Branch: "Computer" } });

// 8. Remove document with RollNo 1
db.students.deleteOne({ RollNo: 1 });

// 9. Display students' information whose name starts with "A"
db.students.find({ Name: { $regex: /^A/ } }).pretty();

// 10. Display total number of documents in the collection
db.students.countDocuments();

// 11. Display only the first 5 documents
db.students.find().limit(5).pretty();

// 12. Display all documents except the third one
db.students.find().skip(2).pretty();

// 13. Display the names of students who live in Pune City
```

```
db.students.find({ "Address.City": "Pune" }, { Name: 1, _id: 0 }).pretty();

// 14. Display the list of different cities students come from
db.students.distinct("Address.City");

// 15. Display the list of different cities with the number of students from
each
db.students.aggregate([
    { $group: { _id: "$Address.City", studentCount: { $sum: 1 } } },
    { $sort: { studentCount: -1 } }
]);

// 16. Display only the names of all students
db.students.find({}, { Name: 1, _id: 0 }).pretty();
```

Explanation of Operations:

1. **Create Database**: `use institute` creates or switches to the database.
2. **Create Collection**: `db.createCollection("students")` creates the `students` collection.
3. **Insert Documents**: `db.students.insertMany([...])` inserts 10 documents into the `students` collection.
4. **Display All Documents**: `db.students.find().pretty()` retrieves and displays all student documents.
5. **Filter by Age**: `db.students.find({ Age: { $gt: 15 } }).pretty()` retrieves students older than 15.
6. **Sort by Name**: `db.students.find().sort({ Name: 1 }).pretty()` displays students sorted alphabetically by `Name`.
7. **Update Branch**: `db.students.updateOne({ RollNo: 3 }, { $set: { Branch: "Computer" } })` updates the branch for `RollNo 3`.
8. **Delete Document**: `db.students.deleteOne({ RollNo: 1 })` removes the document with `RollNo 1`.
9. **Name Starts with "A"**: `db.students.find({ Name: { $regex: /^A/ } }).pretty()` displays students with names starting with "A".
10. **Count Documents**: `db.students.countDocuments()` returns the total number of documents.
11. **Limit Documents**: `db.students.find().limit(5).pretty()` limits the display to the first 5 documents.
12. **Skip Documents**: `db.students.find().skip(2).pretty()` skips the first 2 documents and displays the rest.
13. **Filter by City**: `db.students.find({ "Address.City": "Pune" }, { Name: 1, _id: 0 }).pretty()` displays names of students in Pune.
14. **Unique Cities**: `db.students.distinct("Address.City")` lists unique cities from which students come.
15. **Count by City**: Aggregation pipeline groups students by city and counts them.
16. **Display Names Only**: `db.students.find({}, { Name: 1, _id: 0 }).pretty()` displays only student names.

Practical no-13

Q13. Create Database TECSD using MongoDB Create following Collections.

Teachers (Tname, dno, dname, experience, salary, date_of_joining) //dno- department no, Tname-Tearcher name

Students(Sname,roll_no,class)

At list perform 8 queries)

1.Find the information about all teachers

2.Find the information about all teachers of CSD department

3.Find the information about all teachers of computer, IT, and CSD department

4.Find the information about all teachers of computer, IT,and CSD department having salary

to10000/-

5.Find the student information having roll_no = 2 or Sname= Hitesh

6. Update the experience of teacher-praveen to 10years, if the entry is not available in

databaseconsider the entry as new entry.

7.Update the deparment of all the teachers working in IT department to

CSD 8.find the teachers name and their experience from teachers

collection

9. Insert one entry in department collection

10. Change the dept of teacher Rajesh to IT

11. Delete all the documents from teacher’s collection having IT dept.

12. Display with pretty() method, the first 3 documents in teachers collection in ascending order

Code:-

```
javascript
Copy code
// Create the database "TECSD"
use TECSD;

// Create the "Teachers" collection and insert some initial documents
db.Teachers.insertMany([
    { Tname: "Praveen", dno: 1, dname: "Computer", experience: 8, salary: 15000,
date_of_joining: new Date("2018-06-12") },
    { Tname: "Rajesh", dno: 2, dname: "IT", experience: 5, salary: 12000,
date_of_joining: new Date("2019-08-10") },
    { Tname: "Anita", dno: 3, dname: "CSD", experience: 7, salary: 18000,
date_of_joining: new Date("2017-05-21") },
```

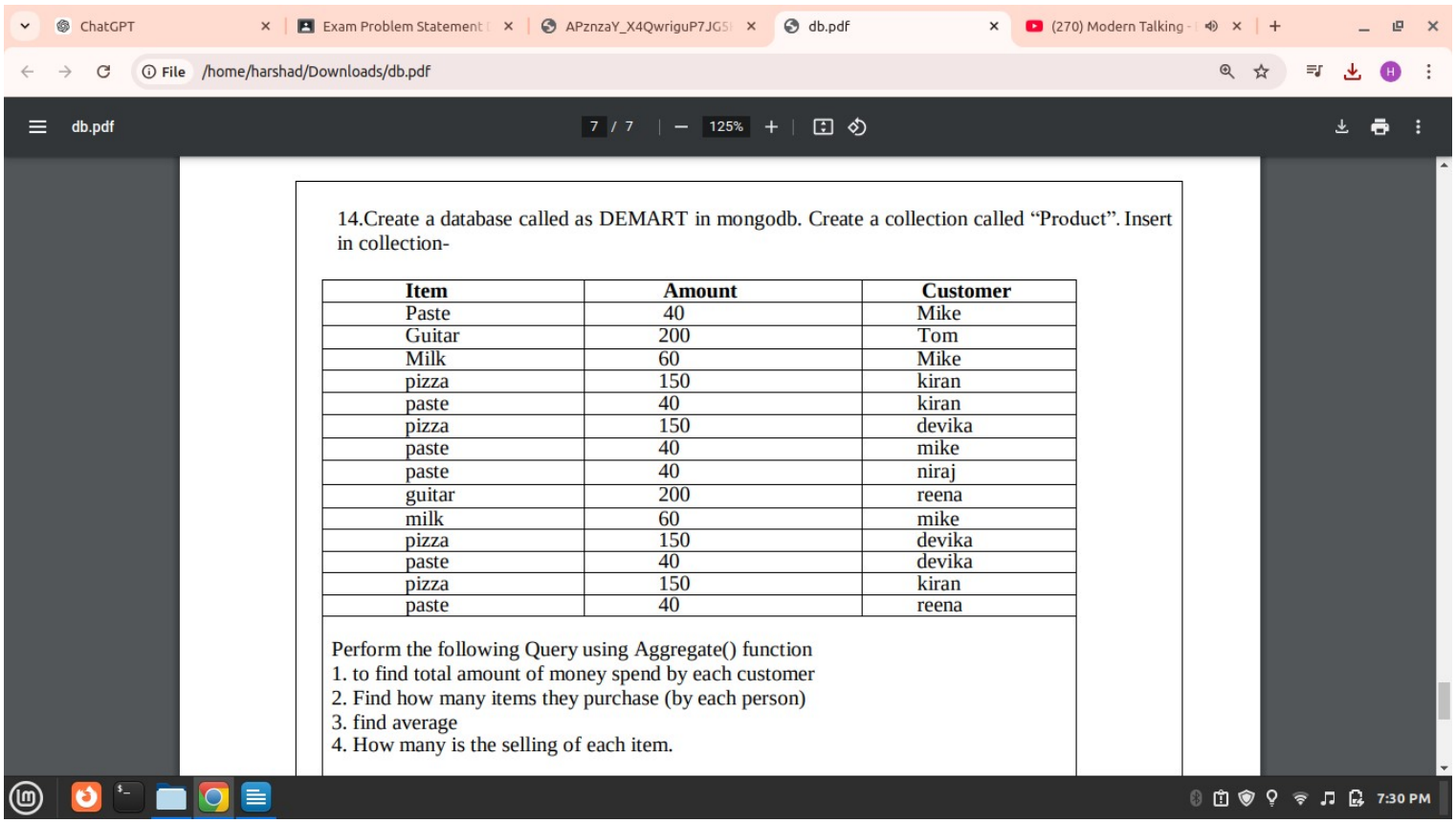
```
    { Tname: "Manisha", dno: 1, dname: "Computer", experience: 6, salary: 10000,
date_of_joining: new Date("2020-01-15") }
]);

// Create the "Students" collection and insert some initial documents
db.Students.insertMany([
    { Sname: "Hitesh", roll_no: 1, class: "10th" },
    { Sname: "Aarav", roll_no: 2, class: "10th" },
    { Sname: "Bhavya", roll_no: 3, class: "11th" }
]);

// Queries:

// 1. Find the information about all teachers
db.Teachers.find().pretty();

// 2. Find the information about all teachers of CSD department
db.Teachers.find({ dname: "CSD" }).pretty();

// 3. Find the information about all teachers of Computer, IT, and CSD
departments
db.Teachers.find({ dname: { $in: ["Computer", "IT", "CSD"] } }).pretty();

// 4. Find the information about all teachers of Computer, IT, and CSD
departments with a salary of 10,000
db.Teachers.find({ dname: { $in: ["Computer", "IT", "CSD"] }, salary:
10000 }).pretty();

// 5. Find the student information where roll_no is 2 or Sname is "Hitesh"
db.Students.find({ $or: [{ roll_no: 2 }, { Sname: "Hitesh" }] }).pretty();

// 6. Update the experience of teacher "Praveen" to 10 years. If "Praveen" is
not found, insert a new document
db.Teachers.updateOne(
    { Tname: "Praveen" },
    { $set: { experience: 10 } },
    { upsert: true }
);

// 7. Update the department of all teachers in the IT department to CSD
db.Teachers.updateMany(
    { dname: "IT" },
    { $set: { dname: "CSD" } }
);

// 8. Find the names and experience of all teachers
db.Teachers.find({}, { Tname: 1, experience: 1, _id: 0 }).pretty();

// 9. Insert one entry into a new "Department" collection
db.Department.insertOne({ dno: 4, dname: "Mathematics", hod: "Dr. Sharma" });

// 10. Change the department of teacher "Rajesh" to "IT"
db.Teachers.updateOne(
    { Tname: "Rajesh" },
    { $set: { dname: "IT" } }
);

// 11. Delete all documents from the Teachers collection with the department
"IT"
db.Teachers.deleteMany({ dname: "IT" });

// 12. Display with pretty() method, the first 3 documents in the Teachers
collection in ascending order by name
db.Teachers.find().sort({ Tname: 1 }).limit(3).pretty();
```

Explanation of Queries:

1. **Find all teachers**: `db.Teachers.find().pretty()` displays all documents in the `Teachers` collection.
2. **Find teachers in CSD department**: `db.Teachers.find({ dname: "CSD" }).pretty()` retrieves teachers in the `CSD` department.
3. **Find teachers in multiple departments**: `db.Teachers.find({ dname: { $in: ["Computer", "IT", "CSD"] } }).pretty()` fetches teachers from `Computer`, `IT`, and `CSD`.
4. **Find teachers with salary of 10,000 in specific departments**: Combines department and salary conditions.
5. **Find specific student by roll number or name**: `$or` operator fetches either condition.
6. **Update teacher experience**: Uses `upsert` option to either update or insert if not found.
7. **Change IT department teachers to CSD**: `updateMany` changes all matching documents.
8. **Display teachers' names and experience only**: Field projection shows only specified fields.
9. **Insert new department entry**: Inserts a new document in a new `Department` collection.
10. **Update Rajesh's department**: Changes `dname` to `IT` for the teacher "Rajesh".
11. **Delete IT department teachers**: `deleteMany` removes all teachers with `dname` of `IT`.
12. **Display first 3 documents in ascending order**: `sort` and `limit` control order and count.

Practical no-14

Q.14



14.Create a database called as DEMART in mongodb. Create a collection called "Product". Insert in collection-

Item	Amount	Customer
Paste	40	Mike
Guitar	200	Tom
Milk	60	Mike
pizza	150	kiran
paste	40	kiran
pizza	150	devika
paste	40	mike
paste	40	niraj
guitar	200	reena
milk	60	mike
pizza	150	devika
paste	40	devika
pizza	150	kiran
paste	40	reena

Perform the following Query using Aggregate() function
1. to find total amount of money spend by each customer
2. Find how many items they purchase (by each person)
3. find average
4. How many is the selling of each item.

Code:-

```
// 1. Create the database "DEMART"
use DEMART;

// 2. Create the "Product" collection and insert the documents
db.Product.insertMany([
    { Item: "Paste", Amount: 40, Customer: "Mike" },
    { Item: "Guitar", Amount: 200, Customer: "Tom" },
    { Item: "Milk", Amount: 60, Customer: "Mike" },
    { Item: "Pizza", Amount: 150, Customer: "Kiran" },
    { Item: "Paste", Amount: 40, Customer: "Kiran" },
    { Item: "Pizza", Amount: 150, Customer: "Devika" },
    { Item: "Paste", Amount: 40, Customer: "Mike" },
    { Item: "Paste", Amount: 40, Customer: "Niraj" },
    { Item: "Guitar", Amount: 200, Customer: "Reena" },
    { Item: "Milk", Amount: 60, Customer: "Mike" },
    { Item: "Pizza", Amount: 150, Customer: "Devika" },
    { Item: "Paste", Amount: 40, Customer: "Devika" },
    { Item: "Pizza", Amount: 150, Customer: "Kiran" },
    { Item: "Paste", Amount: 40, Customer: "Reena" }
]);

// Queries:

// 1. Find the total amount of money spent by each customer
db.Product.aggregate([
```

```
    { $group: { _id: "$Customer", totalSpent: { $sum: "$Amount" } } }
]);

// 2. Find how many items each customer purchased
db.Product.aggregate([
    { $group: { _id: "$Customer", itemsPurchased: { $sum: 1 } } }
]);

// 3. Find the average amount spent by each customer
db.Product.aggregate([
    { $group: { _id: "$Customer", averageSpent: { $avg: "$Amount" } } }
]);

// 4. Find how many times each item was sold
db.Product.aggregate([
    { $group: { _id: "$Item", totalSold: { $sum: 1 } } }
]);
```

Explanation of Queries:

1. **Total amount spent by each customer**: This query groups the documents by `Customer` and sums the `Amount` spent by each customer.
2. **Total items purchased by each customer**: This query counts the number of documents for each `Customer`, indicating the number of items purchased.
3. **Average amount spent by each customer**: This query calculates the average `Amount` spent by each `Customer`.
4. **Total selling count for each item**: This query groups by `Item` and counts how many times each item was sold.

Practical no-15

Q15.You are tasked with analyzing student performance data stored in a MongoDB collection. Each document represents a student's name and their marks. Your goal is to utilize MongoDB's MapReduce functionality to derive meaningful insights from this data.You have the following records in the students collection:

- Amit: 80, 90 🕐
- Shreya: 40 🕐
- Neha: 80, 35 🕐

Perform following queries.

1. Counting Student Occurrences

2. Retrieving Occurrence Counts

3. Calculating Total Marks

4. Sorting Results by Total Marks

5. Limiting Results

6. Conditional-Aggregation

Code:-

Step 1: Insert Data

First, let's create the `students` collection and insert the documents with student names and marks.

```
use SchoolDB;

db.students.insertMany([
    { name: "Amit", marks: [80, 90] },
    { name: "Shreya", marks: [40] },
    { name: "Neha", marks: [80, 35] }
]);
```

Step 2: Define `map` and `reduce` Functions

Define the `map` function to emit each student's name and their total marks. The `reduce` function will be used to sum up the marks if necessary.

```
// Map function to emit each student's name and their total marks
var mapFunction = function() {
    var totalMarks = Array.isArray(this.marks) ? this.marks.reduce((a, b) => a +
b, 0) : this.marks;
    emit(this.name, totalMarks);
};

// Reduce function to sum up marks in case there are multiple entries for the
same student
var reduceFunction = function(key, values) {
    return Array.sum(values);
```

```
};
```

Step 3: Perform Queries

1. Counting Student Occurrences

To count the number of occurrences (documents) for each student:

```
db.students.mapReduce(
    function() { emit(this.name, 1); },  // map: emit 1 for each student name
    function(key, values) { return Array.sum(values); },  // reduce: sum up the
occurrences
    { out: "student_occurrences" }
);
db.student_occurrences.find().pretty();
```

2. Retrieving Occurrence Counts

This is similar to the first query; it retrieves the count of each student's occurrences from the `student_occurrences` collection.

```
db.student_occurrences.find().pretty();
```

3. Calculating Total Marks

To calculate the total marks for each student:

```
db.students.mapReduce(
    mapFunction,  // map function defined earlier
    reduceFunction,  // reduce function defined earlier
    { out: "total_marks" }
);
db.total_marks.find().pretty();
```

4. Sorting Results by Total Marks

To sort the results by total marks in descending order:

```
db.total_marks.find().sort({ value: -1 }).pretty();
```

5. Limiting Results

To limit the number of results to, for example, the top 2 students with the highest marks:

```
db.total_marks.find().sort({ value: -1 }).limit(2).pretty();
```

6. Conditional Aggregation

To perform conditional aggregation, such as calculating total marks only for students who have a score above 50 in at least one subject:

```
var mapFunctionConditional = function() {
    var totalMarks = Array.isArray(this.marks) ? this.marks.reduce((a, b) => a +
b, 0) : this.marks;
    if (this.marks.some(mark => mark > 50)) {
        emit(this.name, totalMarks);
    }
};
```

```
db.students.mapReduce(
    mapFunctionConditional,
    reduceFunction,
    { out: "conditional_aggregation" }
);
db.conditional_aggregation.find().pretty();
```

Explanation of Each Query:

1. **Counting Student Occurrences**: This counts how many times each student's record appears in the collection.
2. **Retrieving Occurrence Counts**: Displays the count of occurrences for each student from the `student_occurrences` collection.
3. **Calculating Total Marks**: Calculates the total marks for each student by summing up their scores.
4. **Sorting Results by Total Marks**: Sorts students by total marks in descending order.
5. **Limiting Results**: Limits the output to the top students by total marks.
6. **Conditional Aggregation**: Calculates the total marks for students who have at least one subject mark above 50.