S1.Consider following Relation Account (Acc\_no, branch\_name,balance) Branch(branch\_name,branch\_city,assets) Customer(cust\_name,cust\_street,cust\_city) Depositor(cust\_name,acc\_no) Loan(loan\_no,branch\_name,amount) Borrower(cust\_name,loan\_no) Create above tables with appropriate constraints like primary key, foreign key, not null etc. 1. Find the names of all branches in loan relation. 2. Find all loan numbers for loans made at ‘Wadia College’ Branch with loan amount > 12000. 3. Find all customers who have a loan from bank. Find their names,loan\_no and loan amount. 4. List all customers in alphabetical order who have loan from ‘Wadia College’ branch. 5. Display distinct cities of branch.

CREATE TABLE Account (

Acc\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

balance DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Branch (

branch\_name VARCHAR(50) PRIMARY KEY,

branch\_city VARCHAR(50) NOT NULL,

assets DECIMAL(15, 2) NOT NULL

);

CREATE TABLE Customer (

cust\_name VARCHAR(50) PRIMARY KEY,

cust\_street VARCHAR(100),

cust\_city VARCHAR(50) NOT NULL

);

CREATE TABLE Depositor (

cust\_name VARCHAR(50),

acc\_no INT,

PRIMARY KEY (cust\_name, acc\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (acc\_no) REFERENCES Account(acc\_no)

);

CREATE TABLE Loan (

loan\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

amount DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Borrower (

cust\_name VARCHAR(50),

loan\_no INT,

PRIMARY KEY (cust\_name, loan\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (loan\_no) REFERENCES Loan(loan\_no)

);

Find the names of all branches in the loan relation.

SELECT DISTINCT branch\_name FROM Loan;

Find all loan numbers for loans made at the ‘Wadia College’ branch with loan amount > 12000.

SELECT loan\_no FROM Loan

WHERE branch\_name = 'Wadia College' AND amount > 12000;

Find all customers who have a loan from the bank. Find their names, loan number, and loan amount.

SELECT Customer.cust\_name, Loan.loan\_no, Loan.amount

FROM Customer

JOIN Borrower ON Customer.cust\_name = Borrower.cust\_name

JOIN Loan ON Borrower.loan\_no = Loan.loan\_no;

List all customers in alphabetical order who have a loan from the ‘Wadia College’ branch.

SELECT DISTINCT Customer.cust\_name

FROM Customer

JOIN Borrower ON Customer.cust\_name = Borrower.cust\_name

JOIN Loan ON Borrower.loan\_no = Loan.loan\_no

WHERE Loan.branch\_name = 'Wadia College'

ORDER BY Customer.cust\_name;

Display distinct cities of branches.

SELECT DISTINCT branch\_city FROM Branch;

S2. Consider following Relation Account (Acc\_no, branch\_name,balance) Branch(branch\_name,branch\_city,assets) Customer(cust\_name,cust\_street,cust\_city) Depositor(cust\_name,acc\_no) Loan(loan\_no,branch\_name,amount) Borrower(cust\_name,loan\_no) Create above tables with appropriate constraints like primary key, foreign key, not null etc. 1. Find all customers who have both account and loan at bank. 2. Find all customers who have an account or loan or both at bank. 3. Find all customers who have account but no loan at the bank. 4. Find average account balance at ‘Wadia College’ branch. 5. Find no. of depositors at each branch

CREATE TABLE Account (

Acc\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

balance DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Branch (

branch\_name VARCHAR(50) PRIMARY KEY,

branch\_city VARCHAR(50) NOT NULL,

assets DECIMAL(15, 2) NOT NULL

);

CREATE TABLE Customer (

cust\_name VARCHAR(50) PRIMARY KEY,

cust\_street VARCHAR(100),

cust\_city VARCHAR(50) NOT NULL

);

CREATE TABLE Depositor (

cust\_name VARCHAR(50),

acc\_no INT,

PRIMARY KEY (cust\_name, acc\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (acc\_no) REFERENCES Account(acc\_no)

);

CREATE TABLE Loan (

loan\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

amount DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Borrower (

cust\_name VARCHAR(50),

loan\_no INT,

PRIMARY KEY (cust\_name, loan\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (loan\_no) REFERENCES Loan(loan\_no)

);

Find all customers who have both an account and a loan at the bank.

SELECT DISTINCT Depositor.cust\_name

FROM Depositor

INNER JOIN Borrower ON Depositor.cust\_name = Borrower.cust\_name;

Find all customers who have an account or a loan or both at the bank.

SELECT DISTINCT cust\_name

FROM (

SELECT cust\_name FROM Depositor

UNION

SELECT cust\_name FROM Borrower

) AS all\_customers;

Find all customers who have an account but no loan at the bank.

SELECT DISTINCT Depositor.cust\_name

FROM Depositor

LEFT JOIN Borrower ON Depositor.cust\_name = Borrower.cust\_name

WHERE Borrower.loan\_no IS NULL;

Find the average account balance at the ‘Wadia College’ branch.

SELECT AVG(balance) AS avg\_balance

FROM Account

WHERE branch\_name = 'Wadia College';

Find the number of depositors at each branch.

SELECT branch\_name, COUNT(DISTINCT Depositor.cust\_name) AS num\_depositors

FROM Account

JOIN Depositor ON Account.Acc\_no = Depositor.acc\_no

GROUP BY branch\_name;

Consider following Relation Account (Acc\_no, branch\_name,balance) Branch(branch\_name,branch\_city,assets) Customer(cust\_name,cust\_street,cust\_city) Depositor(cust\_name,acc\_no) Loan(loan\_no,branch\_name,amount) Borrower(cust\_name,loan\_no) Create above tables with appropriate constraints like primary key, foreign key, not null etc. 1. Find the branches where average account balance > 15000. 2. Find number of tuples in customer relation. 3. Calculate total loan amount given by bank. 4. Delete all loans with loan amount between 1300 and 1500. 5. Find the average account balance at each branch 6. Find name of Customer and city where customer name starts with Letter P.

CREATE TABLE Account (

Acc\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

balance DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Branch (

branch\_name VARCHAR(50) PRIMARY KEY,

branch\_city VARCHAR(50) NOT NULL,

assets DECIMAL(15, 2) NOT NULL

);

CREATE TABLE Customer (

cust\_name VARCHAR(50) PRIMARY KEY,

cust\_street VARCHAR(100) NOT NULL,

cust\_city VARCHAR(50) NOT NULL

);

CREATE TABLE Depositor (

cust\_name VARCHAR(50),

acc\_no INT,

PRIMARY KEY (cust\_name, acc\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (acc\_no) REFERENCES Account(Acc\_no)

);

CREATE TABLE Loan (

loan\_no INT PRIMARY KEY,

branch\_name VARCHAR(50) NOT NULL,

amount DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (branch\_name) REFERENCES Branch(branch\_name)

);

CREATE TABLE Borrower (

cust\_name VARCHAR(50),

loan\_no INT,

PRIMARY KEY (cust\_name, loan\_no),

FOREIGN KEY (cust\_name) REFERENCES Customer(cust\_name),

FOREIGN KEY (loan\_no) REFERENCES Loan(loan\_no)

);

**Find the branches where average account balance > 15000.**

SELECT branch\_name

FROM Account

GROUP BY branch\_name

HAVING AVG(balance) > 15000;

**Find the number of tuples in the Customer relation.**

SELECT COUNT(\*) AS num\_tuples

FROM Customer;

**Calculate the total loan amount given by the bank.**

SELECT SUM(amount) AS total\_loan\_amount

FROM Loan;

**Delete all loans with loan amount between 1300 and 1500.**

DELETE FROM Loan

WHERE amount BETWEEN 1300 AND 1500;

**Find the average account balance at each branch.**

SELECT branch\_name, AVG(balance) AS average\_balance

FROM Account

GROUP BY branch\_name;

**Find the name of the customer and city where the customer’s name starts with the letter 'P'.**

SELECT cust\_name, cust\_city

FROM Customer

WHERE cust\_name LIKE 'P%';

S4. SQL Queries: Create following tables with suitable constraints (primary key, foreign key, not null etc). Insert record and solve the following queries: Create table Cust\_Master(Cust\_no, Cust\_name, Cust\_addr) Create table Order(Order\_no, Cust\_no, Order\_date, Qty\_Ordered) Create Product (Product\_no, Product\_name, Order\_no) 1. List names of customers having 'A' as second letter in their name. 2. Display order from Customer no C1002, C1005, C1007 and C1008 3. List Clients who stay in either 'Banglore or 'Manglore' 4. Display name of customers& the product\_name they have purchase 5. Create view View1 consisting of Cust\_name, Product\_name. 6. Disply product\_name and quantity purchase by each customer 7. Perform different joint operation.

CREATE TABLE Cust\_Master (

Cust\_no VARCHAR(10) PRIMARY KEY,

Cust\_name VARCHAR(50) NOT NULL,

Cust\_addr VARCHAR(100) NOT NULL

);

CREATE TABLE Order (

Order\_no INT PRIMARY KEY,

Cust\_no VARCHAR(10) NOT NULL,

Order\_date DATE NOT NULL,

Qty\_Ordered INT NOT NULL,

FOREIGN KEY (Cust\_no) REFERENCES Cust\_Master(Cust\_no)

);

CREATE TABLE Product (

Product\_no INT PRIMARY KEY,

Product\_name VARCHAR(50) NOT NULL,

Order\_no INT NOT NULL,

FOREIGN KEY (Order\_no) REFERENCES Order(Order\_no)

);

-- Inserting records into Cust\_Master

INSERT INTO Cust\_Master (Cust\_no, Cust\_name, Cust\_addr)

VALUES ('C1001', 'Alice', 'Mumbai'),

('C1002', 'Anil', 'Bangalore'),

('C1003', 'Ravi', 'Chennai'),

('C1004', 'Pankaj', 'Manglore'),

('C1005', 'Amit', 'Pune'),

('C1006', 'Pranav', 'Manglore');

-- Inserting records into Order

INSERT INTO Order (Order\_no, Cust\_no, Order\_date, Qty\_Ordered)

VALUES (101, 'C1001', '2024-10-01', 2),

(102, 'C1002', '2024-10-05', 1),

(103, 'C1003', '2024-10-10', 3),

(104, 'C1004', '2024-10-15', 5),

(105, 'C1005', '2024-10-20', 2),

(106, 'C1006', '2024-10-25', 4);

-- Inserting records into Product

INSERT INTO Product (Product\_no, Product\_name, Order\_no)

VALUES (1, 'Laptop', 101),

(2, 'Tablet', 102),

(3, 'Smartphone', 103),

(4, 'Monitor', 104),

(5, 'Keyboard', 105),

(6, 'Mouse', 106);

List names of customers having 'A' as the second letter in their name.

SELECT Cust\_name

FROM Cust\_Master

WHERE Cust\_name LIKE '\_A%';

Display orders from Customer no C1002, C1005, C1007, and C1008.

SELECT \*

FROM Order

WHERE Cust\_no IN ('C1002', 'C1005', 'C1007', 'C1008');

List clients who stay in either 'Bangalore' or 'Mangalore'.

SELECT Cust\_name

FROM Cust\_Master

WHERE Cust\_addr IN ('Bangalore', 'Manglore');

Display the name of customers and the product\_name they have purchased.

SELECT Cust\_Master.Cust\_name, Product.Product\_name

FROM Cust\_Master

JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no

JOIN Product ON Order.Order\_no = Product.Order\_no;

Create view View1 consisting of Cust\_name, Product\_name.

CREATE VIEW View1 AS

SELECT Cust\_Master.Cust\_name, Product.Product\_name

FROM Cust\_Master

JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no

JOIN Product ON Order.Order\_no = Product.Order\_no;

**Display product\_name and quantity purchased by each customer.**

SELECT Cust\_Master.Cust\_name, Product.Product\_name, Order.Qty\_Ordered

FROM Cust\_Master

JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no

JOIN Product ON Order.Order\_no = Product.Order\_no;

Perform different join operations.

**Inner Join**: Lists only matching records.

SELECT Cust\_Master.Cust\_name, Order.Order\_no

FROM Cust\_Master

INNER JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no;

**Left Join**: Lists all customers and their orders if any, showing NULL for customers without orders.

SELECT Cust\_Master.Cust\_name, Order.Order\_no

FROM Cust\_Master

LEFT JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no;

**Right Join**: Lists all orders and their customers, showing NULL for orders without a matching customer (if any).

SELECT Cust\_Master.Cust\_name, Order.Order\_no

FROM Cust\_Master

RIGHT JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no;

**Full Outer Join**: Lists all customers and all orders, with NULLs for any missing matches on either side (only works if supported by your SQL system).

SELECT Cust\_Master.Cust\_name, Order.Order\_no

FROM Cust\_Master

FULL OUTER JOIN Order ON Cust\_Master.Cust\_no = Order.Cust\_no;

S5. Consider following Relation Employee(emp\_id,employee\_name,street,city) Works(employee\_name,company\_name,salary) Company(company\_name,city) Manages(employee\_name,manager\_name) Create above tables with appropriate constraints like primary key, foreign key, not null etc. 1. Find the names of all employees who work for ‘TCS’. 2. Find the names and company names of all employees sorted in ascending order of company name and descending order of employee names of that company. 3. Change the city of employee working with InfoSys to ‘Bangalore’ 4. Find the names, street address, and cities of residence for all employees who work for 'TechM' and earn more than $10,000. 5. Add Column Asset to Company table.

CREATE TABLE Employee (

emp\_id INT PRIMARY KEY,

employee\_name VARCHAR(50) NOT NULL,

street VARCHAR(100),

city VARCHAR(50)

);

CREATE TABLE Company (

company\_name VARCHAR(50) PRIMARY KEY,

city VARCHAR(50) NOT NULL

);

CREATE TABLE Works (

employee\_name VARCHAR(50) NOT NULL,

company\_name VARCHAR(50) NOT NULL,

salary DECIMAL(10, 2),

FOREIGN KEY (employee\_name) REFERENCES Employee(employee\_name),

FOREIGN KEY (company\_name) REFERENCES Company(company\_name)

);

CREATE TABLE Manages (

employee\_name VARCHAR(50) NOT NULL,

manager\_name VARCHAR(50) NOT NULL,

FOREIGN KEY (employee\_name) REFERENCES Employee(employee\_name),

FOREIGN KEY (manager\_name) REFERENCES Employee(employee\_name)

);

INSERT INTO Employee (emp\_id, employee\_name, street, city)

VALUES

(1, 'Alice', '123 Maple St', 'Mumbai'),

(2, 'Bob', '456 Oak St', 'Delhi'),

(3, 'Charlie', '789 Pine St', 'Pune'),

(4, 'Diana', '101 Birch St', 'Hyderabad'),

(5, 'Eve', '202 Cedar St', 'Bangalore');

INSERT INTO Company (company\_name, city)

VALUES

('TCS', 'Mumbai'),

('InfoSys', 'Bangalore'),

('TechM', 'Pune'),

('Wipro', 'Hyderabad');

INSERT INTO Works (employee\_name, company\_name, salary)

VALUES

('Alice', 'TCS', 12000.00),

('Bob', 'InfoSys', 15000.00),

('Charlie', 'TechM', 9500.00),

('Diana', 'TechM', 10500.00),

('Eve', 'Wipro', 13000.00);

INSERT INTO Manages (employee\_name, manager\_name)

VALUES

('Alice', 'Bob'),

('Charlie', 'Alice'),

('Diana', 'Charlie'),

('Eve', 'Alice');

SELECT employee\_name

FROM Works

WHERE company\_name = 'TCS';

SELECT employee\_name, company\_name

FROM Works

ORDER BY company\_name ASC, employee\_name DESC;

UPDATE Employee

SET city = 'Bangalore'

WHERE employee\_name IN (

SELECT employee\_name

FROM Works

WHERE company\_name = 'InfoSys'

);

SELECT e.employee\_name, e.street, e.city

FROM Employee e

JOIN Works w ON e.employee\_name = w.employee\_name

WHERE w.company\_name = 'TechM' AND w.salary > 10000;

ALTER TABLE Company

ADD COLUMN Asset DECIMAL(15, 2);

S6.Consider following Relation Employee(emp\_id,employee\_name,street,city) Works(employee\_name,company\_name,salary) Company(company\_name,city) Manages(employee\_name,manager\_name) Create above tables with appropriate constraints like primary key, foreign key, not null etc. 1. Change the city of employee working with InfoSys to ‘Bangalore’ 2. Find the names of all employees who earn more than the average salary of all employees of their company. Assume that all people work for at most one company. 3. Find the names, street address, and cities of residence for all employees who work for 'TechM' and earn more than $10,000. 4. Change name of table Manages to Management. 5. Create Simple and Unique index on employee table. 6. Display index Information

CREATE TABLE Employee (

emp\_id INT PRIMARY KEY,

employee\_name VARCHAR(50) NOT NULL UNIQUE,

street VARCHAR(100),

city VARCHAR(50)

);

CREATE TABLE Company (

company\_name VARCHAR(50) PRIMARY KEY,

city VARCHAR(50) NOT NULL

);

CREATE TABLE Works (

employee\_name VARCHAR(50) NOT NULL,

company\_name VARCHAR(50) NOT NULL,

salary DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (employee\_name) REFERENCES Employee(employee\_name),

FOREIGN KEY (company\_name) REFERENCES Company(company\_name)

);

CREATE TABLE Manages (

employee\_name VARCHAR(50) NOT NULL,

manager\_name VARCHAR(50) NOT NULL,

FOREIGN KEY (employee\_name) REFERENCES Employee(employee\_name),

FOREIGN KEY (manager\_name) REFERENCES Employee(employee\_name)

);

**Change the city of employees working with InfoSys to 'Bangalore':**

UPDATE Employee

SET city = 'Bangalore'

WHERE employee\_name IN (

SELECT employee\_name

FROM Works

WHERE company\_name = 'InfoSys'

);

Find names of employees earning more than the average salary of employees in their company:

SELECT w.employee\_name

FROM Works w

JOIN (

SELECT company\_name, AVG(salary) AS avg\_salary

FROM Works

GROUP BY company\_name

) AS avg\_salaries

ON w.company\_name = avg\_salaries.company\_name

WHERE w.salary > avg\_salaries.avg\_salary;

Find names, street addresses, and cities for employees working for 'TechM' and earning more than $10,000:

SELECT e.employee\_name, e.street, e.city

FROM Employee e

JOIN Works w ON e.employee\_name = w.employee\_name

WHERE w.company\_name = 'TechM' AND w.salary > 10000;

Change the name of the Manages table to Management:

ALTER TABLE Manages

RENAME TO Management;

Create a simple and unique index on the Employee table:

-- Simple index on employee\_name

CREATE INDEX idx\_employee\_name ON Employee(employee\_name);

-- Unique index on emp\_id

CREATE UNIQUE INDEX idx\_emp\_id ON Employee(emp\_id);

Display index information:

SHOW INDEX FROM Employee;

S7. Consider following Relation Account (Acc\_no, branch\_name,balance) Branch(branch\_name,branch\_city,assets) Customer(cust\_name,cust\_street,cust\_city) Depositor(cust\_name,acc\_no) Loan(loan\_no,branch\_name,amount) Borrower(cust\_name,loan\_no) Execute the following query: 1. Create a View1 to display List all customers in alphabetical order who have loan from Pune\_Station branch. 2. Create View2 on branch table by selecting any two columns and perform insert update delete operations. 3. Create View3 on borrower and depositor table by selecting any one column from each table perform insert update delete operations. 4. Create Union of left and right joint for all customers who have an account or loan or both at bank 5. Create Simple and Unique index. 6. Display index Information.

Create View1 for Customers with Loans from Pune\_Station Branch

CREATE VIEW View1 AS

SELECT c.cust\_name

FROM Customer c

JOIN Borrower b ON c.cust\_name = b.cust\_name

JOIN Loan l ON b.loan\_no = l.loan\_no

WHERE l.branch\_name = 'Pune\_Station'

ORDER BY c.cust\_name ASC;

Create View2 on Branch Table and Perform Insert, Update, Delete Operations

CREATE VIEW View2 AS

SELECT branch\_name, branch\_city

FROM Branch;

**Insert Operation:**

INSERT INTO Branch (branch\_name, branch\_city)

VALUES ('New\_Branch', 'Mumbai');

Update Operation:

UPDATE Branch

SET branch\_city = 'Pune'

WHERE branch\_name = 'New\_Branch';

Delete Operation:

DELETE FROM Branch

WHERE branch\_name = 'New\_Branch';

. Create View3 on Borrower and Depositor Tables and Perform Insert, Update, Delete Operations

CREATE VIEW View3 AS

SELECT b.cust\_name AS borrower\_name, d.acc\_no AS depositor\_acc\_no

FROM Borrower b

JOIN Depositor d ON b.cust\_name = d.cust\_name;

**Insert Operation:**

INSERT INTO Depositor (cust\_name, acc\_no)

VALUES ('New\_Customer', 12345);

INSERT INTO Borrower (cust\_name, loan\_no)

VALUES ('New\_Customer', 56789);

Update Operation:

UPDATE Depositor

SET acc\_no = 54321

WHERE cust\_name = 'New\_Customer';

Delete Operation:

DELETE FROM Borrower

WHERE cust\_name = 'New\_Customer';

DELETE FROM Depositor

WHERE cust\_name = 'New\_Customer';

Create Union of Left and Right Joins for All Customers Who Have an Account or Loan or Both

SELECT DISTINCT cust\_name

FROM Depositor

LEFT JOIN Borrower ON Depositor.cust\_name = Borrower.cust\_name

UNION

SELECT DISTINCT cust\_name

FROM Borrower

RIGHT JOIN Depositor ON Borrower.cust\_name = Depositor.cust\_name;

Create Simple and Unique Index

CREATE INDEX idx\_balance ON Account(balance);

CREATE UNIQUE INDEX idx\_cust\_name ON Customer(cust\_name);

Display Index Information

SHOW INDEX FROM Account;

SHOW INDEX FROM Customer;

SHOW INDEX FROM Branch;

SHOW INDEX FROM Loan;

SHOW INDEX FROM Depositor;

SHOW INDEX FROM Borrower;

S8. Consider following Relation: Companies (comp\_id, name, cost, year) Orders (comp\_id, domain, quantity) Execute the following query: 1. Find names, costs, domains and quantities for companies using inner join. 2. Find names, costs, domains and quantities for companies using left outer join. 3. Find names, costs, domains and quantities for companies using right outer join. 4. Find names, costs, domains and quantities for companies using Union operator. 5. Create View View1 by selecting both tables to show company name and quantities. 6. Create View View2 by selecting any two columns and perform insert update delete operations. 7. Display content of View1, View2.

CREATE TABLE Companies (

comp\_id INT PRIMARY KEY,

name VARCHAR(50) NOT NULL,

cost DECIMAL(10, 2),

year INT

);

CREATE TABLE Orders (

comp\_id INT,

domain VARCHAR(50),

quantity INT,

FOREIGN KEY (comp\_id) REFERENCES Companies(comp\_id)

);

Inner Join: Find names, costs, domains, and quantities for companies.

SELECT c.name, c.cost, o.domain, o.quantity

FROM Companies c

INNER JOIN Orders o ON c.comp\_id = o.comp\_id;

Left Outer Join: Find names, costs, domains, and quantities for companies.

SELECT c.name, c.cost, o.domain, o.quantity

FROM Companies c

LEFT JOIN Orders o ON c.comp\_id = o.comp\_id;

Right Outer Join: Find names, costs, domains, and quantities for companie

SELECT c.name, c.cost, o.domain, o.quantity

FROM Companies c

RIGHT JOIN Orders o ON c.comp\_id = o.comp\_id;

Union Operator: Find names, costs, domains, and quantities for companies.

SELECT c.name, c.cost, o.domain, o.quantity

FROM Companies c

LEFT JOIN Orders o ON c.comp\_id = o.comp\_id

UNION

SELECT c.name, c.cost, o.domain, o.quantity

FROM Companies c

RIGHT JOIN Orders o ON c.comp\_id = o.comp\_id;

Create View1 to show company name and quantities.

CREATE VIEW View1 AS

SELECT c.name, o.quantity

FROM Companies c

JOIN Orders o ON c.comp\_id = o.comp\_id;

Create View2 by selecting any two columns and perform insert, update, delete operations.

Create View2:

CREATE VIEW View2 AS

SELECT name, cost

FROM Companies;

Insert Operation:

INSERT INTO Companies (comp\_id, name, cost, year)

VALUES (4, 'NewCompany', 5000.00, 2022);

Update Operation:

UPDATE Companies

SET cost = 5500.00

WHERE name = 'NewCompany';

Delete Operation:

DELETE FROM Companies

WHERE name = 'NewCompany';

Display content of View1 and View2.

SELECT \* FROM View1;

SELECT \* FROM View2;

S9. SQL Queries Create following tables with suitable constraints. Insert data and solve the following queries: CUSTOMERS(CNo, Cname, Ccity, CMobile) ITEMS(INo, Iname, Itype, Iprice, Icount) PURCHASE(PNo, Pdate, Pquantity, Cno, INo) 1. List all stationary items with price between 400/- to 1000/- 2. Change the mobile number of customer “Gopal” 3. Display the item with maximum price 4. Display all purchases sorted from the most recent to the oldest 5. Count the number of customers in every city 6. Display all purchased quantity of Customer Maya 7. Create view which shows Iname, Price and Count of all stationary items in descending order of price.

CREATE TABLE CUSTOMERS (

CNo INT PRIMARY KEY,

Cname VARCHAR(50) NOT NULL,

Ccity VARCHAR(50),

CMobile VARCHAR(15)

);

CREATE TABLE ITEMS (

INo INT PRIMARY KEY,

Iname VARCHAR(50) NOT NULL,

Itype VARCHAR(50),

Iprice DECIMAL(10, 2),

Icount INT

);

CREATE TABLE PURCHASE (

PNo INT PRIMARY KEY,

Pdate DATE,

Pquantity INT,

CNo INT,

INo INT,

FOREIGN KEY (CNo) REFERENCES CUSTOMERS(CNo),

FOREIGN KEY (INo) REFERENCES ITEMS(INo)

);

-- Inserting sample data into CUSTOMERS table

INSERT INTO CUSTOMERS (CNo, Cname, Ccity, CMobile) VALUES

(1, 'Gopal', 'Delhi', '9876543210'),

(2, 'Maya', 'Mumbai', '9123456789'),

(3, 'Ravi', 'Delhi', '9234567890');

-- Inserting sample data into ITEMS table

INSERT INTO ITEMS (INo, Iname, Itype, Iprice, Icount) VALUES

(1, 'Pen', 'Stationary', 500, 100),

(2, 'Notebook', 'Stationary', 700, 50),

(3, 'Calculator', 'Electronics', 1500, 30),

(4, 'Eraser', 'Stationary', 300, 200);

-- Inserting sample data into PURCHASE table

INSERT INTO PURCHASE (PNo, Pdate, Pquantity, CNo, INo) VALUES

(1, '2024-10-01', 2, 1, 1),

(2, '2024-10-05', 1, 2, 2),

(3, '2024-10-07', 3, 2, 4),

(4, '2024-10-10', 2, 3, 3);

List all stationary items with a price between 400 and 1000.

SELECT Iname, Iprice

FROM ITEMS

WHERE Itype = 'Stationary' AND Iprice BETWEEN 400 AND 1000;

Change the mobile number of customer “Gopal”.

UPDATE CUSTOMERS

SET CMobile = '9999999999'

WHERE Cname = 'Gopal';

**Display the item with the maximum price.**

SELECT Iname, Iprice

FROM ITEMS

WHERE Iprice = (SELECT MAX(Iprice) FROM ITEMS);

Display all purchases sorted from the most recent to the oldest.

SELECT \*

FROM PURCHASE

ORDER BY Pdate DESC;

Count the number of customers in every city.

SELECT Ccity, COUNT(\*) AS CustomerCount

FROM CUSTOMERS

GROUP BY Ccity;

Display all purchased quantity of Customer Maya.

SELECT p.Pquantity, i.Iname

FROM PURCHASE p

JOIN CUSTOMERS c ON p.CNo = c.CNo

JOIN ITEMS i ON p.INo = i.INo

WHERE c.Cname = 'Maya';

Create a view that shows Iname, Price, and Count of all stationary items in descending order of price.

CREATE VIEW StationaryView AS

SELECT Iname, Iprice, Icount

FROM ITEMS

WHERE Itype = 'Stationary'

ORDER BY Iprice DESC;

To display the content of StationaryView:

SELECT \* FROM StationaryView;

P1. Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 5 to 9. Store the radius and the corresponding values of calculated area in an empty table named areas, consisting of two columns, radius and area.

-- Create the areas table if it does not exist

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE areas (

radius NUMBER,

area NUMBER

)';

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE = -955 THEN

NULL; -- Table already exists, ignore error

ELSE

RAISE;

END IF;

END;

/

-- PL/SQL block to calculate and insert area values

DECLARE

radius\_val NUMBER;

area\_val NUMBER;

BEGIN

-- Loop through radius values from 5 to 9

FOR radius\_val IN 5..9 LOOP

-- Calculate the area of the circle

area\_val := 3.14159 \* radius\_val \* radius\_val;

-- Insert the radius and area into the areas table

INSERT INTO areas (radius, area) VALUES (radius\_val, area\_val);

END LOOP;

-- Commit the transaction to save the data

COMMIT;

END;

/

P2. Write an Unnamed PL/SQL of code for the following requirements: - Schema: Borrower (Rollin, Name, DateofIssue, NameofBook, Status) Fine (Roll\_no,Date,Amt) Accept roll\_no & name of book from user. Check the number of days (from date of issue). 1. If days are between 15 to 30 then fine amounts will be Rs 5 per day. 2. If no. of days>30, per day fine will be Rs 50 per day & for days less than 30, Rs. 5 per day. 3. After submitting the book, status will change from I to R. 4. If condition of fine is true, then details will be stored into fine table.

DECLARE

v\_roll\_no Borrower.Rollin%TYPE;

v\_nameofbook Borrower.NameofBook%TYPE;

v\_dateofissue Borrower.DateofIssue%TYPE;

v\_status Borrower.Status%TYPE;

v\_fine\_amt NUMBER := 0;

v\_days NUMBER;

BEGIN

-- Accept input for roll number and name of book

v\_roll\_no := &roll\_no;

v\_nameofbook := '&name\_of\_book';

-- Retrieve DateofIssue and Status for the given roll number and book name

SELECT DateofIssue, Status

INTO v\_dateofissue, v\_status

FROM Borrower

WHERE Rollin = v\_roll\_no AND NameofBook = v\_nameofbook;

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

v\_days := TRUNC(SYSDATE - v\_dateofissue);

-- Calculate the fine based on the number of days

IF v\_days > 30 THEN

-- Fine calculation for more than 30 days

v\_fine\_amt := (30 \* 5) + ((v\_days - 30) \* 50);

ELSIF v\_days BETWEEN 15 AND 30 THEN

-- Fine calculation for 15 to 30 days

v\_fine\_amt := v\_days \* 5;

END IF;

-- Update the status in the Borrower table if the book is being returned

IF v\_status = 'I' THEN

UPDATE Borrower

SET Status = 'R'

WHERE Rollin = v\_roll\_no AND NameofBook = v\_nameofbook;

END IF;

-- Insert a record into the Fine table if there is a fine

IF v\_fine\_amt > 0 THEN

INSERT INTO Fine (Roll\_no, Date, Amt)

VALUES (v\_roll\_no, SYSDATE, v\_fine\_amt);

END IF;

-- Commit the transaction

COMMIT;

-- Output the result

DBMS\_OUTPUT.PUT\_LINE('Days overdue: ' || v\_days);

DBMS\_OUTPUT.PUT\_LINE('Fine amount: ' || v\_fine\_amt);

DBMS\_OUTPUT.PUT\_LINE('Status updated to Returned (R) in Borrower table.');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No record found for the given roll number and book name.');

WHEN OTHERS THEN

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

END;

/

P3. Write a PL/SQL block of code using 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 the data in the first table already exist in the second table, then that data should be skipped.

DECLARE

CURSOR n\_rollcall\_cursor IS

SELECT roll\_no, name, date FROM N\_RollCall;

v\_roll\_no N\_RollCall.roll\_no%TYPE;

v\_name N\_RollCall.name%TYPE;

v\_date N\_RollCall.date%TYPE;

BEGIN

FOR record IN n\_rollcall\_cursor LOOP

-- Attempt to insert only if roll\_no doesn't exist in O\_RollCall

BEGIN

INSERT INTO O\_RollCall (roll\_no, name, date)

SELECT record.roll\_no, record.name, record.date

FROM dual

WHERE NOT EXISTS (

SELECT 1 FROM O\_RollCall WHERE roll\_no = record.roll\_no

);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

NULL; -- Skip duplicates

END;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Merge completed successfully.');

END;

/

P4.Write a PL/SQL block for following requirements and handle the exceptions. Roll no. of students will be entered by the user. Attendance of roll no. entered by user will be checked in the Stud table. If attendance is less than 75% then display the message “Term not granted” and set the status in stud table as “Detained”. Otherwise display message “Term granted” and set the status in stud table as “Not Detained”. Student (Roll, Name, Attendance, Status)

DECLARE

v\_roll Stud.Roll%TYPE; -- Variable for roll number

v\_attendance Stud.Attendance%TYPE; -- Variable for attendance

v\_status VARCHAR2(20); -- Variable for status message

BEGIN

-- Prompt the user to enter the roll number

v\_roll := &Enter\_Roll\_No;

-- Fetch attendance for the given roll number

SELECT Attendance INTO v\_attendance

FROM Stud

WHERE Roll = v\_roll;

-- Check attendance and update status

IF v\_attendance < 75 THEN

v\_status := 'Detained';

UPDATE Stud

SET Status = v\_status

WHERE Roll = v\_roll;

DBMS\_OUTPUT.PUT\_LINE('Term not granted');

ELSE

v\_status := 'Not Detained';

UPDATE Stud

SET Status = v\_status

WHERE Roll = v\_roll;

DBMS\_OUTPUT.PUT\_LINE('Term granted');

END IF;

COMMIT; -- Commit the transaction

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: No student found with Roll No. ' || v\_roll);

WHEN OTHERS THEN

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

END;

/

CREATE TABLE Stud (

Roll NUMBER PRIMARY KEY,

Name VARCHAR2(50),

Attendance NUMBER(5,2), -- Store attendance as a percentage (e.g., 80.50)

Status VARCHAR2(20) -- Store status as "Detained" or "Not Detained"

);

INSERT INTO Stud (Roll, Name, Attendance, Status) VALUES (1, 'John Doe', 80, NULL);

INSERT INTO Stud (Roll, Name, Attendance, Status) VALUES (2, 'Jane Smith', 70, NULL);

INSERT INTO Stud (Roll, Name, Attendance, Status) VALUES (3, 'Mike Johnson', 65, NULL);

INSERT INTO Stud (Roll, Name, Attendance, Status) VALUES (4, 'Emily Davis', 85, NULL);

COMMIT;

P5.Write a PL/SQL Block to increase the salary of employees by 10% of existing salary, who are having salary less than average salary of organization, whenever such salary updates take place, a record for same is maintained in the increment\_salary table. emp(emp\_no, salary) increment\_salary(emp\_no, salary)

CREATE TABLE emp (

emp\_no NUMBER PRIMARY KEY,

salary NUMBER(10,2)

);

CREATE TABLE increment\_salary (

emp\_no NUMBER,

salary NUMBER(10,2),

increment\_date DATE DEFAULT SYSDATE

);

INSERT INTO emp (emp\_no, salary) VALUES (1, 30000);

INSERT INTO emp (emp\_no, salary) VALUES (2, 40000);

INSERT INTO emp (emp\_no, salary) VALUES (3, 25000);

INSERT INTO emp (emp\_no, salary) VALUES (4, 45000);

COMMIT;

DECLARE

v\_avg\_salary NUMBER(10,2); -- Variable to store the average salary

BEGIN

-- Calculate the average salary of all employees

SELECT AVG(salary) INTO v\_avg\_salary FROM emp;

-- Update salaries and log increments for employees earning below the average salary

FOR employee IN (SELECT emp\_no, salary FROM emp WHERE salary < v\_avg\_salary) LOOP

-- Increase salary by 10%

UPDATE emp

SET salary = salary \* 1.1

WHERE emp\_no = employee.emp\_no;

-- Insert record of salary increment in increment\_salary table

INSERT INTO increment\_salary (emp\_no, salary)

VALUES (employee.emp\_no, employee.salary \* 1.1);

END LOOP;

COMMIT; -- Commit all updates and inserts

DBMS\_OUTPUT.PUT\_LINE('Salary updates completed successfully.');

EXCEPTION

WHEN OTHERS THEN

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

END;

/

P6.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 categories is first class, if marks 899 and 825 category is Higher Second Class. Write a PL/SQL block for using procedure created with above requirement. Stud\_Marks(name, total\_marks), Result (Roll,Name, Class)

CREATE TABLE Stud\_Marks (

name VARCHAR2(50),

total\_marks NUMBER

);

CREATE TABLE Result (

Roll NUMBER PRIMARY KEY,

name VARCHAR2(50),

Class VARCHAR2(20)

);

INSERT INTO Stud\_Marks (name, total\_marks) VALUES ('John Doe', 1450);

INSERT INTO Stud\_Marks (name, total\_marks) VALUES ('Jane Smith', 960);

INSERT INTO Stud\_Marks (name, total\_marks) VALUES ('Mike Johnson', 880);

INSERT INTO Stud\_Marks (name, total\_marks) VALUES ('Emily Davis', 850);

COMMIT;

CREATE OR REPLACE PROCEDURE proc\_Grade (

p\_name IN VARCHAR2,

p\_total\_marks IN NUMBER

) AS

v\_class VARCHAR2(20);

BEGIN

-- Determine the category based on total marks

IF p\_total\_marks BETWEEN 990 AND 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;

-- Insert the result into the Result table

INSERT INTO Result (Roll, name, Class)

VALUES ((SELECT NVL(MAX(Roll), 0) + 1 FROM Result), p\_name, v\_class);

DBMS\_OUTPUT.PUT\_LINE('Result added for student: ' || p\_name || ' with class: ' || v\_class);

EXCEPTION

WHEN OTHERS THEN

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

END proc\_Grade;

/

BEGIN

FOR student IN (SELECT name, total\_marks FROM Stud\_Marks) LOOP

-- Call proc\_Grade for each student

proc\_Grade(student.name, student.total\_marks);

END LOOP;

END;

/

P7.Create a stored function titled 'Age\_calc'. Accept the date of birth of a person as a parameter. Calculate the age of the person in years, months and days e.g. 3 years, 2months, 10 days. Return the age in years directly (with the help of Return statement). The months and days are to be returned indirectly in the form of OUT parameters.

CREATE OR REPLACE FUNCTION Age\_calc (

p\_dob DATE,

p\_months OUT NUMBER,

p\_days OUT NUMBER

) RETURN NUMBER AS

v\_years NUMBER;

v\_dob DATE := p\_dob;

v\_today DATE := SYSDATE;

BEGIN

-- Calculate the difference in years

v\_years := FLOOR(MONTHS\_BETWEEN(v\_today, v\_dob) / 12);

-- Calculate remaining months after full years

p\_months := FLOOR(MONTHS\_BETWEEN(v\_today, v\_dob) - (v\_years \* 12));

-- Calculate days remaining after full months

p\_days := v\_today - ADD\_MONTHS(v\_dob, v\_years \* 12 + p\_months);

RETURN v\_years; -- Return the age in years

END Age\_calc;

/

DECLARE

v\_years NUMBER;

v\_months NUMBER;

v\_days NUMBER;

BEGIN

-- Call the Age\_calc function with a sample date of birth

v\_years := Age\_calc(DATE '2000-05-15', v\_months, v\_days);

-- Display the results

DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_years || ' years, ' || v\_months || ' months, ' || v\_days || ' days');

END;

/

P8. Write a Row Level Before and After Trigger on Library table. The System should keep track of the records that are being updated or deleted. The old value of updated or deleted records should be added in Library\_Audit table.

CREATE TABLE Library (

book\_id NUMBER PRIMARY KEY,

book\_name VARCHAR2(100),

author VARCHAR2(100),

published\_year NUMBER

);

CREATE TABLE Library\_Audit (

audit\_id NUMBER PRIMARY KEY,

action\_type VARCHAR2(10),

book\_id NUMBER,

book\_name VARCHAR2(100),

author VARCHAR2(100),

published\_year NUMBER,

action\_date DATE

);

CREATE SEQUENCE audit\_seq START WITH 1 INCREMENT BY 1;

CREATE OR REPLACE TRIGGER before\_update\_delete\_Library

BEFORE UPDATE OR DELETE ON Library

FOR EACH ROW

BEGIN

INSERT INTO Library\_Audit (

audit\_id,

action\_type,

book\_id,

book\_name,

author,

published\_year,

action\_date

)

VALUES (

audit\_seq.NEXTVAL,

CASE WHEN DELETING THEN 'DELETE' ELSE 'UPDATE' END,

:OLD.book\_id,

:OLD.book\_name,

:OLD.author,

:OLD.published\_year,

SYSDATE

);

END;

/

P9. Trigger: Create a row level trigger for the CUSTOMERS table that would fire INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values.

CREATE TABLE CUSTOMERS (

customer\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100),

salary NUMBER

);

CREATE OR REPLACE TRIGGER trg\_salary\_diff

BEFORE INSERT OR UPDATE OR DELETE ON CUSTOMERS

FOR EACH ROW

BEGIN

-- Display the salary for an INSERT operation

IF INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('New salary (INSERT): ' || :NEW.salary);

-- Display the salary difference for an UPDATE operation

ELSIF UPDATING THEN

IF :NEW.salary IS NOT NULL AND :OLD.salary IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Old salary: ' || :OLD.salary);

DBMS\_OUTPUT.PUT\_LINE('New salary: ' || :NEW.salary);

DBMS\_OUTPUT.PUT\_LINE('Salary difference: ' || (:NEW.salary - :OLD.salary));

ELSE

DBMS\_OUTPUT.PUT\_LINE('Salary update, but no previous salary available.');

END IF;

-- Display the salary for a DELETE operation

ELSIF DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('Deleted salary: ' || :OLD.salary);

END IF;

END;

/

SET SERVEROUTPUT ON;

-- Insert example

INSERT INTO CUSTOMERS (customer\_id, customer\_name, salary) VALUES (1, 'John Doe', 5000);

-- Update example

UPDATE CUSTOMERS SET salary = 6000 WHERE customer\_id = 1;

-- Delete example

DELETE FROM CUSTOMERS WHERE customer\_id = 1;

P10.Trigger: Write a after trigger for Insert, update and delete event considering following requirement: Emp(Emp\_no, Emp\_name, Emp\_salary) a) Trigger should be initiated when salary tried to be inserted is less than Rs.50,000/- b) Trigger should be initiated when salary tried to be updated for value less than Rs. 50,000/- Also the new values expected to be inserted will be stored in new table Tracking(Emp\_no,Emp\_salary).

CREATE TABLE Emp (

Emp\_no NUMBER PRIMARY KEY,

Emp\_name VARCHAR2(100),

Emp\_salary NUMBER

);

CREATE TABLE Tracking (

Emp\_no NUMBER,

Emp\_salary NUMBER

);

CREATE OR REPLACE TRIGGER trg\_track\_low\_salary

AFTER INSERT OR UPDATE OR DELETE ON Emp

FOR EACH ROW

BEGIN

-- Insert condition: When inserting a new record with salary < 50,000

IF INSERTING THEN

IF :NEW.Emp\_salary < 50000 THEN

INSERT INTO Tracking (Emp\_no, Emp\_salary)

VALUES (:NEW.Emp\_no, :NEW.Emp\_salary);

DBMS\_OUTPUT.PUT\_LINE('Trigger fired: Insert with salary < 50,000. Record added to Tracking.');

END IF;

-- Update condition: When updating an existing record to a salary < 50,000

ELSIF UPDATING THEN

IF :NEW.Emp\_salary < 50000 THEN

INSERT INTO Tracking (Emp\_no, Emp\_salary)

VALUES (:NEW.Emp\_no, :NEW.Emp\_salary);

DBMS\_OUTPUT.PUT\_LINE('Trigger fired: Update with salary < 50,000. Record added to Tracking.');

END IF;

-- Delete condition: Track if you need to manage deleted data (optional, based on requirements)

ELSIF DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('Delete operation detected, but no action required in this trigger.');

END IF;

END;

/

SET SERVEROUTPUT ON;

-- Test Insert

INSERT INTO Emp (Emp\_no, Emp\_name, Emp\_salary) VALUES (1, 'Alice', 45000);

-- Test Update

UPDATE Emp SET Emp\_salary = 40000 WHERE Emp\_no = 1;

-- Test Delete (no action in Tracking table for delete)

DELETE FROM Emp WHERE Emp\_no = 1;

M1. Design and Develop MongoDB Queries using CRUD operations:

Create Employee collection by considering following Fields:

i. Name: Embedded Doc (FName, LName)

ii. Company Name: String

iii. Salary: Number

iv. Designation: String

v. Age: Number

vi. Expertise: Array

vii. DOB: String or Date

viii. Email id: String

ix. Contact: String

x. Address: Array of Embedded Doc (PAddr, LAddr)

Insert at least 5 documents in collection by considering above

attribute and execute following queries:

1. Select all documents where the Designation field has the value

"Programmer" and the value of the salary field is greater than

30000.

2. Creates a new document if no document in the employee collection

contains

{Designation: "Tester", Company\_name: "TCS", Age: 25}

3. Increase salary of each Employee working with “Infosys" 10000.

4. Finds all employees working with "TCS" and reduce their salary

by 5000.

5. Return documents where Designation is not equal to "Tester".

6. Find all employee with Exact Match on an Array having Expertise:

['Mongodb','Mysql','Cassandra']

db.Employee.insertMany([

{

Name: { FName: "John", LName: "Doe" },

CompanyName: "TCS",

Salary: 35000,

Designation: "Programmer",

Age: 28,

Expertise: ["Mongodb", "Node.js", "React"],

DOB: new Date("1995-04-10"),

Email: "john.doe@example.com",

Contact: "1234567890",

Address: [{ PAddr: "123 Main St", LAddr: "456 Elm St" }]

},

{

Name: { FName: "Jane", LName: "Smith" },

CompanyName: "Infosys",

Salary: 45000,

Designation: "Programmer",

Age: 32,

Expertise: ["Java", "C++", "Mongodb"],

DOB: new Date("1991-07-20"),

Email: "jane.smith@example.com",

Contact: "0987654321",

Address: [{ PAddr: "789 Oak St", LAddr: "101 Pine St" }]

},

{

Name: { FName: "Bob", LName: "Brown" },

CompanyName: "TCS",

Salary: 28000,

Designation: "Tester",

Age: 25,

Expertise: ["Testing", "Automation", "Selenium"],

DOB: new Date("1998-03-15"),

Email: "bob.brown@example.com",

Contact: "1231231234",

Address: [{ PAddr: "111 Cedar St", LAddr: "222 Maple St" }]

},

{

Name: { FName: "Alice", LName: "White" },

CompanyName: "Infosys",

Salary: 52000,

Designation: "Developer",

Age: 30,

Expertise: ["Python", "Django", "Mongodb"],

DOB: new Date("1994-02-25"),

Email: "alice.white@example.com",

Contact: "3213214321",

Address: [{ PAddr: "333 Birch St", LAddr: "444 Fir St" }]

},

{

Name: { FName: "Eve", LName: "Green" },

CompanyName: "Wipro",

Salary: 47000,

Designation: "Programmer",

Age: 29,

Expertise: ["Mongodb", "Mysql", "Cassandra"],

DOB: new Date("1995-10-12"),

Email: "eve.green@example.com",

Contact: "4321432143",

Address: [{ PAddr: "555 Palm St", LAddr: "666 Willow St" }]

}

]);

Select all documents where Designation is "Programmer" and Salary is greater than 30000:

db.Employee.find({

Designation: "Programmer",

Salary: { $gt: 30000 }

});

Create a new document if no document exists with {Designation: "Tester", CompanyName: "TCS", Age: 25}:

db.Employee.updateOne(

{ Designation: "Tester", CompanyName: "TCS", Age: 25 },

{

$setOnInsert: {

Name: { FName: "New", LName: "Tester" },

CompanyName: "TCS",

Salary: 30000,

Designation: "Tester",

Age: 25,

Expertise: ["Testing"],

DOB: new Date("1998-05-15"),

Email: "new.tester@tcs.com",

Contact: "5675675678",

Address: [{ PAddr: "777 Spruce St", LAddr: "888 Maple St" }]

}

},

{ upsert: true }

);

Increase the salary of each Employee working with "Infosys" by 10000:

db.Employee.updateMany(

{ CompanyName: "Infosys" },

{ $inc: { Salary: 10000 } }

);

Find all employees working with "TCS" and reduce their salary by

db.Employee.updateMany(

{ CompanyName: "TCS" },

{ $inc: { Salary: -5000 } }

);

Return documents where Designation is not "Tester":

db.Employee.find({

Designation: { $ne: "Tester" }

});

Find all employees with an exact match on an array with Expertise: ['Mongodb', 'Mysql', 'Cassandra']:

db.Employee.find({

Expertise: ["Mongodb", "Mysql", "Cassandra"]

});

M2.Design and Develop MongoDB Queries using CRUD operations:

Create Employee collection by considering following Fields:

i. Name: Embedded Doc (FName, LName)

ii. Company Name: String

iii. Salary: Number

iv. Designation: String

v. Age: Number

vi. Expertise: Array

vii. DOB: String or Date

viii. Email id: String

ix. Contact: String

x. Address: Array of Embedded Doc (PAddr, LAddr)

Insert at least 5 documents in collection by considering above

attribute and execute following queries:

1. Final name of Employee where age is less than 30 and salary more

than 50000.

2. Creates a new document if no document in the employee collection

contains

{Designation: "Tester", Company\_name: "TCS", Age: 25}

3. Selects all documents in the collection where the field age has

a value less than 30 or the value of the salary field is greater

than 40000.

4. Find documents where Designation is not equal to "Developer".

5. Find \_id, Designation, Address and Name from all documents where

Company\_name is "Infosys".

6. Display only FName and LName of all Employees

1. Final name of Employee where age is less than 30 and salary more than 50000.

db.Employee.find(

{ Age: { $lt: 30 }, Salary: { $gt: 50000 } },

{ "Name.FName": 1, "Name.LName": 1, \_id: 0 }

);

2. Create a new document if no document in the employee collection contains { Designation: "Tester", Company\_name: "TCS", Age: 25 }.

db.Employee.updateOne(

{ Designation: "Tester", CompanyName: "TCS", Age: 25 },

{

$setOnInsert: {

Name: { FName: "New", LName: "Tester" },

CompanyName: "TCS",

Salary: 30000,

Designation: "Tester",

Age: 25,

Expertise: ["Testing"],

DOB: new Date("1998-05-15"),

Email: "new.tester@tcs.com",

Contact: "5675675678",

Address: [{ PAddr: "777 Spruce St", LAddr: "888 Maple St" }]

}

},

{ upsert: true }

);

3. Selects all documents in the collection where the field Age is less than 30 or the field Salary is greater than 40000.

db.Employee.find({

$or: [

{ Age: { $lt: 30 } },

{ Salary: { $gt: 40000 } }

]

});

4. Find documents where Designation is not equal to "Developer".

db.Employee.find({

Designation: { $ne: "Developer" }

});

5. Find \_id, Designation, Address, and Name from all documents where CompanyName is "Infosys".

db.Employee.find(

{ CompanyName: "Infosys" },

{ \_id: 1, Designation: 1, Address: 1, Name: 1 }

);

6. Display only FName and LName of all employees.

db.Employee.find(

{},

{ "Name.FName": 1, "Name.LName": 1, \_id: 0 }

);

M3.Design and Develop MongoDB Queries using CRUD operations:

Create Employee collection by considering following Fields:

i. Emp\_id : Number

ii. Name: Embedded Doc (FName, LName)

iii. Company Name: String

iv. Salary: Number

v. Designation: String

vi. Age: Number

vii. Expertise: Array

viii. DOB: String or Date

ix. Email id: String

x. Contact: String

xi. Address: Array of Embedded Doc (PAddr, LAddr)

Insert at least 5 documents in collection by considering above

attribute and execute following queries:

1. Creates a new document if no document in the employee collection

contains

{Designation: "Tester", Company\_name: "TCS", Age: 25}

2. Finds all employees working with Company\_name: "TCS" and

increase their salary by 2000.

3. Matches all documents where the value of the field Address is an

embedded document that contains only the field city with the

value "Pune" and the field Pin\_code with the value "411001".

4. Find employee details who are working as "Developer" or

"Tester".

5. Drop Single documents where designation="Developer".

6. Count number of documents in employee collection.

1. **Create a new document if no document in the employee collection contains {Designation: "Tester", Company\_name: "TCS", Age: 25}**

db.Employee.updateOne(

{ Designation: "Tester", CompanyName: "TCS", Age: 25 },

{

$setOnInsert: {

Emp\_id: 1001,

Name: { FName: "New", LName: "Tester" },

CompanyName: "TCS",

Salary: 30000,

Designation: "Tester",

Age: 25,

Expertise: ["Testing"],

DOB: new Date("1998-05-15"),

Email: "new.tester@tcs.com",

Contact: "5675675678",

Address: [{ PAddr: "123 Test St", LAddr: "456 Test Ave" }]

}

},

{ upsert: true }

);

2. **Find all employees working with CompanyName: "TCS" and increase their salary by 2000**

db.Employee.updateMany(

{ CompanyName: "TCS" },

{ $inc: { Salary: 2000 } }

);

3. **Match all documents where the value of the field Address is an embedded document containing only the field city with the value "Pune" and the field Pin\_code with the value "411001"**

db.Employee.find(

{ Address: { $elemMatch: { city: "Pune", Pin\_code: "411001" } } }

);

4. **Find employee details who are working as "Developer" or "Tester"**

db.Employee.find(

{ Designation: { $in: ["Developer", "Tester"] } }

);

5. **Drop single documents where Designation: "Developer"**

db.Employee.deleteOne(

{ Designation: "Developer" }

);

6. **Count the number of documents in the employee collection**

db.Employee.countDocuments();

M4. Design and Develop MongoDB Queries using Aggregation operations:

Create Employee collection by considering following Fields:

i. Emp\_id : Number

ii. Name: Embedded Doc (FName, LName)

iii. Company Name: String

iv. Salary: Number

v. Designation: String

vi. Age: Number

vii. Expertise: Array

viii. DOB: String or Date

ix. Email id: String

x. Contact: String

xi. Address: Array of Embedded Doc (PAddr, LAddr)

Insert at least 5 documents in collection by considering above

attribute and execute following:

1. Using aggregation Return Designation with Total Salary is Above

200000.

2. Using Aggregate method returns names and \_id in upper case and

in alphabetical order.

3. Using aggregation method find Employee with Total Salary for

Each City with Designation="DBA".

4. Create Single Field Indexes on Designation field of employee

collection

5. To Create Multikey Indexes on Expertise field of employee

collection.

6. Create an Index on Emp\_id field, compare the time require to

search Emp\_id before and after creating an index. (Hint Add at

least 10000 Documents)

7. Return a List of Indexes on created on employee Collection.

1. Return Designation with Total Salary Above 200000

db.Employee.aggregate([

{

$group: {

\_id: "$Designation",

totalSalary: { $sum: "$Salary" }

}

},

{

$match: {

totalSalary: { $gt: 200000 }

}

}

]);

2. Return Names and \_id in Upper Case and Alphabetical Order

db.Employee.aggregate([

{

$project: {

\_id: 1,

"Name.FName": { $toUpper: "$Name.FName" },

"Name.LName": { $toUpper: "$Name.LName" }

}

},

{

$sort: { "Name.FName": 1 }

}

]);

3. Find Total Salary for Each City for Employees with Designation="DBA"

db.Employee.aggregate([

{

$match: { Designation: "DBA" }

},

{

$unwind: "$Address"

},

{

$group: {

\_id: "$Address.PAddr",

totalSalary: { $sum: "$Salary" }

}

}

]);

4. Create Single Field Index on Designation

db.Employee.createIndex({ Designation: 1 });

5. Create Multikey Index on Expertise

db.Employee.createIndex({ Expertise: 1 });

6. Create an Index on Emp\_id and Compare Query Time

// Define the document structure you want to insert

const employeeDoc = {

Emp\_id: 1,

Name: { FName: "John", LName: "Doe" },

CompanyName: "TechCorp",

Salary: 60000,

Designation: "Engineer",

Age: 30,

Expertise: ["MongoDB", "JavaScript"],

DOB: new Date("1993-04-15"),

Email: "john.doe@techcorp.com",

Contact: "1234567890",

Address: [{ PAddr: "123 Main St", LAddr: "456 Secondary St" }]

};

// Loop to insert the same document 10,000 times

for (let i = 0; i < 10000; i++) {

db.Employee.insertOne(employeeDoc);

}

**6. Create an Index on Emp\_id and Compare Query Time**

First, perform a search without the index:

var startTime = new Date();

db.Employee.find({ Emp\_id: 5000 }); // Sample query

var endTime = new Date();

print("Time taken without index: ", endTime - startTime);

Create an index on the Emp\_id field:

db.Employee.createIndex({ Emp\_id: 1 });

Now, search again and check the time taken:

var startTime = new Date();

db.Employee.find({ Emp\_id: 5000 });

var endTime = new Date();

print("Time taken with index: ", endTime - startTime);

7. Return a List of Indexes on the Employee Collection

db.Employee.getIndexes();

M5. Design and Develop MongoDB Queries using Aggregation operations:

Create Employee collection by considering following Fields:

i. Emp\_id : Number

ii. Name: Embedded Doc (FName, LName)

iii. Company Name: String

iv. Salary: Number

v. Designation: String

vi. Age: Number

vii. Expertise: Array

viii. DOB: String or Date

ix. Email id: String

x. Contact: String

xi. Address: Array of Embedded Doc (PAddr, LAddr)

Insert at least 5 documents in collection by considering above

attribute and execute following:

1. Using aggregation Return separates value in the Expertise array

and return sum of each element of array.

2. Using Aggregate method return Max and Min Salary for each

company.

3. Using Aggregate method find Employee with Total Salary for Each

City with Designation="DBA".

4. Using aggregation method Return separates value in the Expertise

array for employee name where Swapnil Jadhav

5. To Create Compound Indexes on Name: 1, Age: -1

6. Create an Index on Emp\_id field, compare the time require to

search Emp\_id before and after creating an index. (Hint Add at

least 10000 Documents)

7. Return a List of Indexes on created on employee Collection.

1. Using Aggregation to Return Separate Values in the Expertise Array and Sum Each Element’s Occurrences

db.Employee.aggregate([

{ $unwind: "$Expertise" },

{ $group: { \_id: "$Expertise", count: { $sum: 1 } } }

]);

2. Using Aggregation to Return Max and Min Salary for Each Company

db.Employee.aggregate([

{

$group: {

\_id: "$CompanyName",

maxSalary: { $max: "$Salary" },

minSalary: { $min: "$Salary" }

}

}

]);

**3. Using Aggregation to Find Total Salary for Each City with Designation "DBA"**

db.Employee.aggregate([

{ $match: { Designation: "DBA" } },

{ $unwind: "$Address" },

{

$group: {

\_id: "$Address.PAddr",

totalSalary: { $sum: "$Salary" }

}

}

]);

4. Using Aggregation to Return Separate Values in the Expertise Array for Employee Named "Swapnil Jadhav"

db.Employee.aggregate([

{ $match: { "Name.FName": "Swapnil", "Name.LName": "Jadhav" } },

{ $unwind: "$Expertise" },

{

$group: {

\_id: "$Name",

expertiseList: { $addToSet: "$Expertise" }

}

}

]);

5. Creating a Compound Index on Name and Age

db.Employee.createIndex({ "Name.FName": 1, Age: -1 });

6. Creating an Index on Emp\_id Field and Comparing Search Times Before and After Indexing

**Check Search Time Without Index**:

db.Employee.find({ Emp\_id: 1 }).explain("executionStats");

**Create Index on Emp\_id**:

db.Employee.createIndex({ Emp\_id: 1 });

**Check Search Time With Index**:

db.Employee.find({ Emp\_id: 1 }).explain("executionStats");

7. Returning a List of Indexes on the Employee Collection

db.Employee.getIndexes();

M6. Design MongoDB database and perform following Map reduce operation:

Create Employee collection by considering following Fields:

i. Name: Embedded Doc (FName, LName)

ii. Company Name: String

iii. Salary: Number

iv. Designation: String

v. Age: Number

vi. Expertise: Array

vii. DOB: String or Date

viii. Email id: String

ix. Contact: String

x. Address: Array of Embedded Doc (PAddr, LAddr)

Execute the following query:

1. Display the total salary of per company

2. Display the total salary of company Name:"TCS"

3. Return the average salary of company whose address is “Pune".

4. Display total count for “City=Pune”

5. Return count for city pune and age greater than 40.

1. **Create Employee Collection with Sample Data**

db.Employee.insertMany([

{

Name: { FName: "John", LName: "Doe" },

CompanyName: "TCS",

Salary: 60000,

Designation: "Manager",

Age: 45,

Expertise: ["Java", "MongoDB"],

DOB: "1978-10-15",

Email: "john.doe@tcs.com",

Contact: "1234567890",

Address: [{ PAddr: "Pune", LAddr: "Sector 12" }]

},

{

Name: { FName: "Jane", LName: "Smith" },

CompanyName: "TCS",

Salary: 70000,

Designation: "Developer",

Age: 35,

Expertise: ["JavaScript", "MongoDB"],

DOB: "1988-03-25",

Email: "jane.smith@tcs.com",

Contact: "1234567891",

Address: [{ PAddr: "Pune", LAddr: "Sector 10" }]

},

{

Name: { FName: "Mark", LName: "Taylor" },

CompanyName: "Infosys",

Salary: 50000,

Designation: "Tester",

Age: 38,

Expertise: ["QA", "Automation"],

DOB: "1985-06-12",

Email: "mark.taylor@infosys.com",

Contact: "1234567892",

Address: [{ PAddr: "Pune", LAddr: "Sector 8" }]

},

{

Name: { FName: "Sara", LName: "Connor" },

CompanyName: "Wipro",

Salary: 55000,

Designation: "Developer",

Age: 42,

Expertise: ["Python", "Django"],

DOB: "1981-05-01",

Email: "sara.connor@wipro.com",

Contact: "1234567893",

Address: [{ PAddr: "Mumbai", LAddr: "Sector 5" }]

},

{

Name: { FName: "Sam", LName: "Johnson" },

CompanyName: "TCS",

Salary: 65000,

Designation: "Developer",

Age: 29,

Expertise: ["Node.js", "MongoDB"],

DOB: "1994-07-21",

Email: "sam.johnson@tcs.com",

Contact: "1234567894",

Address: [{ PAddr: "Pune", LAddr: "Sector 6" }]

}

]);

2. **MapReduce Queries**

**1. Display the total salary per company**

var mapFunction = function() {

emit(this.CompanyName, this.Salary);

};

var reduceFunction = function(key, values) {

return Array.sum(values);

};

db.Employee.mapReduce(mapFunction, reduceFunction, { out: { inline: 1 } });

2. **Display the total salary of company "TCS"**

var mapFunction = function() {

if (this.CompanyName == "TCS") {

emit(this.CompanyName, this.Salary);

}

};

var reduceFunction = function(key, values) {

return Array.sum(values);

};

db.Employee.mapReduce(mapFunction, reduceFunction, { out: { inline: 1 } });

3. **Return the average salary of company whose address is "Pune"**

var mapFunction = function() {

for (var i = 0; i < this.Address.length; i++) {

if (this.Address[i].PAddr == "Pune") {

emit(this.CompanyName, { salary: this.Salary, count: 1 });

}

}

};

var reduceFunction = function(key, values) {

var result = { salary: 0, count: 0 };

values.forEach(function(value) {

result.salary += value.salary;

result.count += value.count;

});

return result;

};

var finalizeFunction = function(key, reducedValue) {

reducedValue.averageSalary = reducedValue.salary / reducedValue.count;

return reducedValue;

};

db.Employee.mapReduce(mapFunction, reduceFunction, { out: { inline: 1 }, finalize: finalizeFunction });

4. **Display total count for "City=Pune"**

var mapFunction = function() {

for (var i = 0; i < this.Address.length; i++) {

if (this.Address[i].PAddr == "Pune") {

emit("Pune", 1);

}

}

};

var reduceFunction = function(key, values) {

return Array.sum(values);

};

db.Employee.mapReduce(mapFunction, reduceFunction, { out: { inline: 1 } });

5. **Return count for city "Pune" and age greater than 40**

var mapFunction = function() {

for (var i = 0; i < this.Address.length; i++) {

if (this.Address[i].PAddr == "Pune" && this.Age > 40) {

emit("Pune\_Age\_Above\_40", 1);

}

}

};

var reduceFunction = function(key, values) {

return Array.sum(values);

};

db.Employee.mapReduce(mapFunction, reduceFunction, { out: { inline: 1 } });