SQL Queries:

a. Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym, different constraints etc.

```
-- Creating the 'Department' table
CREATE TABLE Department (
    dept_id INT PRIMARY KEY,  -- Primary key constraint
    dept_name VARCHAR(50) NOT NULL,  -- Not null constraint
    location VARCHAR(100)
);

-- Creating the 'Employee' table with constraints
CREATE TABLE Employee (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,  -- Primary key with auto-increment
    first_name VARCHAR(50) NOT NULL,  -- Not null constraint
    last_name VARCHAR(50) NOT NULL,
    hire_date DATE NOT NULL,
    salary DECIMAL(10, 2) CHECK (salary > 0),  -- Check constraint on salary
    dept_id INT,  -- Foreign key reference
    email VARCHAR(100) UNIQUE,  -- Unique constraint on email
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)  -- Foreign key constraint
);

-- Create an index to speed up queries filtering by salary
CREATE INDEX idx_salary ON Employee (salary);

CREATE VIEW Emp AS SELECT * FROM Employee;
```

b. Write at least 10 SQL queries on the suitable database application using SQL DML statements.

```
1 .INSERT INTO Department (dept_id, dept_name, location)
VALUES (1, 'Human Resources', 'New York'),(2, 'IT', 'San Francisco');

2.SELECT * FROM Employee;

3.SELECT first_name, last_name, salary FROM Employee WHERE salary > 60000;

4.SELECT dept_id, AVG(salary) AS avg_salary FROM Employee GROUP BY dept_id;

5.UPDATE Employee SET salary = 60000 WHERE emp_id = 1001;

6.DELETE FROM Employee WHERE emp_id = 1002;

7.SELECT first_name, last_name FROM Employee WHERE dept_id = (SELECT dept_id FROM
Employee WHERE emp_id = 1001);

8.SELECT first_name, last_name FROM Employee WHERE dept_id = 1 UNION SELECT
first_name, last_name FROM Employee WHERE dept_id = 2;
```

```
9.SELECT e.first_name, e.last_name, d.dept_name FROM Employee e LEFT JOIN Department d
ON e.dept_id = d.dept_id;

10.SELECT first_name, last_name FROM Employee WHERE dept_id = 1 INTERSECT SELECT
first_name, last_name FROM Employee WHERE dept_id = 2;
```

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.
Note: Instructor will frame the problem statement for writing PL/SQL block in line with above statement

```
DELIMITER $$

CREATE PROCEDURE CalculateCircleArea()
BEGIN
   DECLARE v_radius INT;
   DECLARE v_area DECIMAL(10, 2);

   -- Loop through radii from 5 to 9
   SET v_radius = 5;
   WHILE v_radius <= 9 DO
      -- Calculate area of circle (Area = π * r^2)
      SET v_area = ROUND(PI() * POWER(v_radius, 2), 2);

      -- Insert the radius and area into the 'areas' table
      INSERT INTO areas (radius, area)
      VALUES (v_radius, v_area);

      -- Increment the radius
      SET v_radius = v_radius + 1;
   END WHILE;

END $$

DELIMITER ;


CREATE TABLE areas ( radius INT, area DECIMAL(10, 2) );
CALL CalculateCircleArea();
```

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 899 and 825 category is Higher Second Class.
Write a PL/SQL block to use procedure created with above requirement.
Stud_Marks(name, total_marks) Result(Roll,Name, Class)
Note: Instructor will frame the problem statement for writing stored procedure and Function in
line with above statement

Step 1: Create the Tables

```
-- Create Stud_Marks table to store student marks
CREATE TABLE Stud_Marks (
   name VARCHAR(100),
   total_marks INT
);


-- Create Result table to store the results
CREATE TABLE Result (
   Roll INT PRIMARY KEY AUTO_INCREMENT,
   Name VARCHAR(100),
   Class VARCHAR(50)
);
```

Step 2: Create the Stored Function get_grade

```
DELIMITER $$

CREATE FUNCTION get_grade(total_marks INT)
RETURNS VARCHAR(50)
DETERMINISTIC
BEGIN
   DECLARE grade VARCHAR(50);

   IF total_marks >= 990 AND total_marks <= 1500 THEN
      SET grade = 'Distinction';
   ELSEIF total_marks >= 900 AND total_marks <= 989 THEN
      SET grade = 'First Class';
   ELSEIF total_marks >= 825 AND total_marks <= 899 THEN
      SET grade = 'Higher Second Class';
   ELSE
      SET grade = 'Fail';
   END IF;

   RETURN grade;
END $$

DELIMITER ;
```

Step 3: Create the Stored Procedure proc_Grade

```
DELIMITER $$

CREATE PROCEDURE proc_Grade(
    IN student_name VARCHAR(100),
    IN student_marks INT
)
BEGIN
    DECLARE student_grade VARCHAR(50);

    -- Get the grade using the get_grade function
    SET student_grade = get_grade(student_marks);

    -- Insert the result into the Result table
    INSERT INTO Result (Name, Class)
    VALUES (student_name, student_grade);

END $$

DELIMITER ;

-- Insert sample data into Stud_Marks table
INSERT INTO Stud_Marks (name, total_marks)
VALUES
('Alice', 1200),
('Bob', 950),
('Charlie', 850),
('David', 800);
```

Step 5: Execute the Procedure and Use the Function

```
CALL proc_Grade('Alice', 1200);    -- Should be Distinction
CALL proc_Grade('Bob', 950);       -- Should be First Class
CALL proc_Grade('Charlie', 850);   -- Should be Higher Second Class
CALL proc_Grade('David', 800);     -- Should be Fail (or custom category)

-- Check the contents of the Result table
SELECT * FROM Result;
```

Cursors: (All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor)
Write a PL/SQL block of code using parameterized Cursor that will merge the data available in the newly created table N_RollCall 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.
Note: Instructor will frame the problem statement for writing PL/SQL block using all types of Cursors in line with above statement.

Step 1: Create the Tables (Assumed Structure)

```
-- Create N_RollCall table
CREATE TABLE N_RollCall (
    RollNo INT,
    Name VARCHAR(100),
    Status VARCHAR(20)
);

-- Create O_RollCall table
CREATE TABLE O_RollCall (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(100),
    Status VARCHAR(20)
);
```

Step 2: PL/SQL Block Using a Parameterized Cursor

```
DECLARE
    -- Declare a parameterized cursor
    CURSOR c_rollcall(p_status VARCHAR) IS
        SELECT RollNo, Name, Status
        FROM N_RollCall
        WHERE Status = p_status;

    -- Variables to store fetched data
    v_rollno INT;
    v_name VARCHAR(100);
    v_status VARCHAR(20);

BEGIN
    -- Loop through different statuses
    FOR status IN ('Present', 'Absent') LOOP
        -- Open and fetch data using the cursor
        OPEN c_rollcall(status);

        -- Loop through all the rows returned by the cursor
        LOOP
            FETCH c_rollcall INTO v_rollno, v_name, v_status;
            EXIT WHEN c_rollcall%NOTFOUND;

            -- Check if the RollNo already exists in O_RollCall
            BEGIN
                -- If the RollNo does not exist, insert the data
                INSERT INTO O_RollCall (RollNo, Name, Status)
                SELECT v_rollno, v_name, v_status
                WHERE NOT EXISTS (
                    SELECT 1 FROM O_RollCall WHERE RollNo = v_rollno
                );
            EXCEPTION
                WHEN DUP_VAL_ON_INDEX THEN
                    -- If duplicate key error (for primary key constraint), skip the insertion
                    NULL; -- Do nothing, just continue
            END;
        END LOOP;
        -- Close the cursor
        CLOSE c_rollcall;
    END LOOP;

    -- Commit the changes (optional, depending on your environment)
    COMMIT;
```

```
-- Insert sample data into N_RollCall
INSERT INTO N_RollCall (RollNo, Name, Status)
VALUES
(1, 'Alice', 'Present'),
(2, 'Bob', 'Absent'),
(3, 'Charlie', 'Present'),
(4, 'David', 'Absent'),
(5, 'Eve', 'Present');
```

Step 3: Check the `O_RollCall` Table After Running the PL/SQL Block

```
-- Check data in O_RollCall
SELECT * FROM O_RollCall;
```

Database Trigger (All Types: Row level and Statement level triggers, Before and After Triggers).
Write a database 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.
Note: Instructor will Frame the problem statement for writing PL/SQL block for all types of Triggers in line with above statement

```
-- Create Products table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(255),
    Price DECIMAL(10, 2),
    Quantity INT
);

-- Create Product_Audit table to store audit records
CREATE TABLE Product_Audit (
    AuditID INT AUTO_INCREMENT PRIMARY KEY,
    ProductID INT,
    ProductName VARCHAR(255),
    Price DECIMAL(10, 2),
    Quantity INT,
    Action VARCHAR(50),  -- 'UPDATE' or 'DELETE'
    Timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


DELIMITER $$

CREATE TRIGGER before_update_product
BEFORE UPDATE ON Products
FOR EACH ROW
BEGIN
    -- Insert the old record into the Product_Audit table before updating
    INSERT INTO Product_Audit (ProductID, ProductName, Price, Quantity, Action)
    VALUES (OLD.ProductID, OLD.ProductName, OLD.Price, OLD.Quantity, 'UPDATE');
END $$

DELIMITER ;




DELIMITER $$

CREATE TRIGGER before_delete_product
BEFORE DELETE ON Products
FOR EACH ROW
BEGIN
    -- Insert the old record into the Product_Audit table before deleting
    INSERT INTO Product_Audit (ProductID, ProductName, Price, Quantity, Action)
    VALUES (OLD.ProductID, OLD.ProductName, OLD.Price, OLD.Quantity, 'DELETE');
END $$

DELIMITER ;




-- Insert sample data into the Products table
INSERT INTO Products (ProductID, ProductName, Price, Quantity)
VALUES
(101, 'Laptop', 799.99, 50),
(102, 'Smartphone', 499.99, 100),
(103, 'Headphones', 59.99, 200);


-- Update the price and quantity of the Laptop
UPDATE Products
SET Price = 749.99, Quantity = 45
WHERE ProductID = 101;
```

```
-- Delete the Smartphone product
DELETE FROM Products
WHERE ProductID = 102;


-- Check the Product_Audit table to view audit logs
SELECT * FROM Product_Audit;
```

MongoDB Queries:
Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators etc.).

```
// Create the 'library' collection
db.createCollection("library");

// Insert documents using insert() method (creating new entries)
db.library.insert({"bid": 1, "name": "dbms"});
db.library.insert({"bid": 2, "name": "Toc", "author": "XYZ"});
db.library.insert({"bid": 3, "name": "CN", "author": "ABCD", "cost": 700});
db.library.insert({"bid": 4, "name": "OOP", "author": "Addison-Wesley", "cost": 400});
db.library.insert({"bid": 5, "name": "SPOS", "author": "PQR", "cost": 500});
db.library.insert({"bid": 6, "name": "AI", "author": "SSC Education", "cost": 800});
db.library.insert({"bid": 7, "name": "C++", "author": "MD Publications", "cost": 400});

// Display all documents in the 'library' collection
db.library.find().pretty();

// Use update() to modify a field (changing cost from 400 to 600)
db.library.update({'cost': 400}, {$set: {'cost': 600}});

// Use updateOne() to update a document where cost > 600, set the cost to 900
db.library.updateOne({'cost': {$gt: 600}}, {$set: {'cost': 900}});

// Display all documents again to reflect changes
db.library.find().pretty();

// Use find with $not and $gt to find documents with cost not greater than 800
db.library.find({"cost": {$not: {$gt: 800}}}).pretty();

// Sort documents by 'bid' in ascending order
db.library.find().sort({"bid": 1}).pretty();

// Use $or operator to find documents where cost is 500 or 800
db.library.find({$or: [{"cost": 500}, {"cost": 800}]}).pretty();

// Get the total number of documents in the 'library' collection
db.library.count();

// Remove the document with bid = 1
db.library.remove({"bid": 1});
```

```
// Using the save() method to insert a new document or update an existing document
// Example 1: Inserting a new document
db.library.save({
  "bid": 1,
  "name": "dbms",
  "author": "Author1", // Adding author to the previously missing field
  "cost": 350
});

// Example 2: Updating an existing document by saving with an existing bid (bid = 3)
db.library.save({
  "bid": 3,
  "name": "CN",
  "author": "ABCD",
  "cost": 750 // Updating the cost of the book
});

// Display the updated collection
db.library.find().pretty();
```

MongoDB aggregate and aggregation

```
db.orders.insertMany([
  {
    "_id": 1,
    "customer": "John Doe",
    "items": [
      { "product": "Laptop", "quantity": 1, "price": 1000 },
      { "product": "Mouse", "quantity": 2, "price": 50 }
    ],
    "total": 1100
  },
  {
    "_id": 2,
    "customer": "Jane Smith",
    "items": [
      { "product": "Laptop", "quantity": 1, "price": 1000 },
      { "product": "Keyboard", "quantity": 1, "price": 150 }
    ],
    "total": 1150
  }
]);

db.orders.aggregate([
  { $unwind: "$items" },  // Unwind the items array
  { $group: {
      _id: "$items.product",  // Group by product name
      totalQuantity: { $sum: "$items.quantity" },  // Sum of quantity sold
```

```
     totalRevenue: { $sum: { $multiply: [ "$items.quantity", "$items.price" ] } }  // Calculate total revenue
  }},
  { $sort: { totalRevenue: -1 } }  // Sort by total revenue in descending order
]);

db.customers.insertMany([
  { "_id": 1, "name": "John Doe", "age": 25, "location": "New York" },
  { "_id": 2, "name": "Jane Smith", "age": 30, "location": "London" },
  { "_id": 3, "name": "Bob Johnson", "age": 22, "location": "Sydney" }
]);

db.customers.createIndex({ age: 1 });

db.customers.find({ age: { $gt: 25 } });

db.customers.createIndex({ name: 1, age: -1 });// Compound index on `name` (ascending) and `age` (descending)

db.customers.find({ name: "John Doe", age: { $gt: 20 } });
```

Mongodb map reduce

```
db.sales.insertMany([
  { "_id": 1, "product": "Laptop", "quantity": 3, "price": 1000 },
  { "_id": 2, "product": "Mouse", "quantity": 5, "price": 50 },
  { "_id": 3, "product": "Laptop", "quantity": 2, "price": 1000 },
  { "_id": 4, "product": "Keyboard", "quantity": 10, "price": 100 },
  { "_id": 5, "product": "Mouse", "quantity": 3, "price": 50 }
]);

var mapFunction = function() {
  emit(this.product, this.quantity * this.price);  // Key: product, Value: total revenue (quantity * price)
};
var reduceFunction = function(key, values) {
  return Array.sum(values);  // Sum the values for the product
};

db.sales.mapReduce(
  mapFunction,        // The map function
  reduceFunction,     // The reduce function
  {
    out: "total_revenue_per_product"  // Output collection name
  }
);

db.total_revenue_per_product.find();
```