Lab-1 (SQL Concepts Revision)

Part – A

1. **Retrieve a unique genre of songs.**
```
SELECT DISTINCT Genre FROM Songs;
```

2. **Find top 2 albums released before 2010.**
```
SELECT TOP 2 Album_title, Release_year
FROM Albums
WHERE Release_year < 2010;
```

3. **Insert Data into the Songs Table. (1245, 'Zaroor', 2.55, 'Feel good', 1005)**
```
INSERT INTO Songs (Song_id, Song_title, Duration, Genre, Album_id)
VALUES (1245, 'Zaroor', 2.55, 'Feel good', 1005);
```

4. **Change the Genre of the song 'Zaroor' to 'Happy'**
```
UPDATE Songs
SET Genre = 'Happy'
WHERE Song_title = 'Zaroor';
```

5. **Delete an Artist 'Ed Sheeran'**
```
DELETE FROM Artists
WHERE Artist_name = 'Ed Sheeran';
```

6. **Add a New Column for Rating in Songs Table. [Ratings decimal(3,2)]**
```
ALTER TABLE Songs
ADD Rating DECIMAL(3, 2);
```

7. **Retrieve songs whose title starts with 'S'.**
```
SELECT * FROM Songs
WHERE Song_title LIKE 'S%';
```

8. **Retrieve all songs whose title contains 'Everybody'.**
```
SELECT * FROM Songs
WHERE Song_title LIKE '%Everybody%';
```

9. **Display Artist Name in Uppercase.**
```
SELECT UPPER(Artist_name) AS Artist_Name_Uppercase
FROM Artists;
```

10. **Find the Square Root of the Duration of a Song 'Good Luck'**
```
SELECT Song_title, SQRT(Duration) AS Duration_SquareRoot
FROM Songs
WHERE Song_title = 'Good Luck';
```

11. **Find Current Date.**
```
SELECT GETDATE() AS CurrentDate;
```

12. **Find the number of albums for each artist.**
```
SELECT Artist_id, COUNT(Album_id) AS Album_Count
FROM Albums
```

```
GROUP BY Artist_id;
```

13. **Retrieve the Album_id which has more than 5 songs in it.**

```
SELECT Album_id
FROM Songs
GROUP BY Album_id
HAVING COUNT(Song_id) > 5;
```

14. **Retrieve all songs from the album 'Album1'. (using Subquery)**

```
SELECT * FROM Songs
WHERE Album_id = (SELECT Album_id FROM Albums WHERE Album_title = 'Album1');
```

15. **Retrieve all albums name from the artist 'Aparshakti Khurana' (using Subquery)**

```
SELECT Album_title FROM Albums
WHERE Artist_id = (SELECT Artist_id FROM Artists WHERE Artist_name = 'Aparshakti Khurana');
```

16. **Retrieve all the song titles with its album title.**

```
SELECT Songs.Song_title, Albums.Album_title
FROM Songs
INNER JOIN Albums
ON Songs.Album_id = Albums.Album_id
```

17. **Find all the songs which are released in 2020.**

```
SELECT S.*
FROM Songs S
JOIN Albums AL ON S.Album_id = AL.Album_id
WHERE AL.Release_year = 2020;
```

18. **Create a view called 'Fav_Songs' from the songs table having songs with song_id 101-105.**

```
CREATE VIEW Fav_Songs AS
SELECT * FROM Songs
WHERE Song_id BETWEEN 101 AND 105;
```

19. **Update a song name to 'Jannat' of song having song_id 101 in Fav_Songs view.**

```
UPDATE Fav_Songs
SET Song_title = 'Jannat'
WHERE Song_id = 101;
```

20. **Find all artists who have released an album in 2020.**

```
SELECT DISTINCT A.Artist_name
FROM Artists A
JOIN Albums AL ON A.Artist_id = AL.Artist_id
WHERE AL.Release_year = 2020;
```

21. **Retrieve all songs by Shreya Ghoshal and order them by duration.**

```
SELECT S.*
FROM Songs S
Left JOIN Albums AL ON S.Album_id = AL.Album_id
Left JOIN Artists A ON AL.Artist_id = A.Artist_id
WHERE A.Artist_name= 'Shreya Ghoshal'
ORDER BY S.Duration;
```

Part – B

22. Retrieve all song titles by artists who have more than one album.

```
SELECT S.Song_title
FROM Songs S
JOIN Albums AL ON S.Album_id = AL.Album_id
JOIN Artists A ON AL.Artist_id = A.Artist_id
WHERE A.Artist_id IN (
  SELECT Artist_id
  FROM Albums
  GROUP BY Artist_id
  HAVING COUNT(Album_id) > 1
);
```

23. Retrieve all albums along with the total number of songs.

```
SELECT AL.Album_title, COUNT(S.Song_id) AS Total_Songs
FROM Albums AL
LEFT JOIN Songs S
ON AL.Album_id = S.Album_id
GROUP BY AL.Album_title;
```

24. Retrieve all songs and release year and sort them by release year.

```
SELECT S.Song_title, AL.Release_year
FROM Songs S
JOIN Albums AL
ON S.Album_id = AL.Album_id
ORDER BY AL.Release_year;
```

25. Retrieve the total number of songs for each genre, showing genres that have more than 2 songs.

```
SELECT Genre, COUNT(Song_id) AS Song_Count
FROM Songs
GROUP BY Genre
HAVING COUNT(Song_id) > 2;
```

26. List all artists who have albums that contain more than 3 songs.

```
SELECT A.Artist_name
FROM Artists A
JOIN Albums AL ON A.Artist_id = AL.Artist_id
JOIN Songs S ON AL.Album_id = S.Album_id
GROUP BY A.Artist_name, AL.Album_id
HAVING COUNT(S.Song_id) > 3;
```

Part – C

27. Retrieve albums that have been released in the same year as 'Album4'

```
SELECT AL2.Album_title
FROM Albums AL1
JOIN Albums AL2
ON AL1.Release_year = AL2.Release_year
WHERE AL1.Album_title = 'Album4' AND AL1.Album_id <> AL2.Album_id;
```

28. Find the longest song in each genre

```
SELECT Genre, MAX(Duration) AS Longest_Duration
FROM Songs
GROUP BY Genre;
```

29. **Retrieve the titles of songs released in albums that contain the word 'Album' in the title.**

```
SELECT S.Song_title
FROM Songs S
JOIN Albums AL ON S.Album_id = AL.Album_id
WHERE AL.Album_title LIKE '%Album%';
```

30. **Retrieve the total duration of songs by each artist where total duration exceeds 15 minutes.**

```
SELECT SUM(Songs.Duration),Artists.Artist_name
FROM Songs
Inner join Albums
ON Songs.Album_id = Albums.Album_id
Inner join Artists
ON Albums.Artist_id = Artists.Artist_id
Group by Artists.Artist_name
HAVING SUM(Songs.Duration) > 15
```

Lab-2 Stored Procedure

Part – A

1.Department, Designation & Person Table's INSERT, UPDATE & DELETE Procedures.

```
-- Department INSERT Procedure
CREATE PROCEDURE PR_Department_Insert
    @DepartmentID INT,
    @DepartmentName VARCHAR(100)
AS
BEGIN
    INSERT INTO Department (DepartmentID, DepartmentName)
    VALUES (@DepartmentID, @DepartmentName);
END;

-- Department UPDATE Procedure
CREATE PROCEDURE PR_Department_Update
    @DepartmentID INT,
    @DepartmentName VARCHAR(100)
AS
BEGIN
    UPDATE Department
    SET DepartmentName = @DepartmentName
    WHERE DepartmentID = @DepartmentID;
END;

-- Department DELETE Procedure
CREATE PROCEDURE PR_Department_Delete
    @DepartmentID INT
AS
BEGIN
    DELETE FROM Department
    WHERE DepartmentID = @DepartmentID;
END;
```

```
-- Designation INSERT Procedure
CREATE PROCEDURE PR_Designation_Insert
    @DesignationID INT,
    @DesignationName VARCHAR(100)
AS
BEGIN
    INSERT INTO Designation (DesignationID, DesignationName)
    VALUES (@DesignationID, @DesignationName);
END;

-- Designation UPDATE Procedure
CREATE PROCEDURE PR_Designation_Update
    @DesignationID INT,
    @DesignationName VARCHAR(100)
AS
BEGIN
    UPDATE Designation
    SET DesignationName = @DesignationName
    WHERE DesignationID = @DesignationID;
END;
```

```
-- Designation DELETE Procedure
CREATE PROCEDURE PR_Designation_Delete
    @DesignationID INT
AS
BEGIN
    DELETE FROM Designation
    WHERE DesignationID = @DesignationID;
END;

-- Person INSERT Procedure
CREATE PROCEDURE PR_Person_Insert
    @FirstName VARCHAR(100),
    @LastName VARCHAR(100),
    @Salary DECIMAL(8, 2),
    @JoiningDate DATETIME,
    @DepartmentID INT = NULL,
    @DesignationID INT = NULL
AS
BEGIN
    INSERT INTO Person (FirstName, LastName, Salary, JoiningDate, DepartmentID, DesignationID)
    VALUES (@FirstName, @LastName, @Salary, @JoiningDate, @DepartmentID, @DesignationID);
END;

-- Person UPDATE Procedure
CREATE PROCEDURE PR_Person_Update
    @PersonID INT,
    @FirstName VARCHAR(100),
    @LastName VARCHAR(100),
    @Salary DECIMAL(8, 2),
    @JoiningDate DATETIME,
    @DepartmentID INT = NULL,
    @DesignationID INT = NULL
AS
BEGIN
    UPDATE Person
    SET FirstName = @FirstName, LastName = @LastName, Salary = @Salary, JoiningDate =
@JoiningDate,
        DepartmentID = @DepartmentID, DesignationID = @DesignationID
    WHERE PersonID = @PersonID;
END;

-- Person DELETE Procedure
CREATE PROCEDURE PR_Person_Delete
    @PersonID INT
AS
BEGIN
    DELETE FROM Person
    WHERE PersonID = @PersonID;
END;
```

2. Department, Designation & Person Table's SELECTBYPRIMARYKEY

```
-- Department SELECTBYPRIMARYKEY Procedure
CREATE PROCEDURE PR_Department_SelectByPrimaryKey
    @DepartmentID INT
AS
BEGIN
    SELECT * FROM Department
    WHERE DepartmentID = @DepartmentID;
END;
```

```
-- Designation SELECTBYPRIMARYKEY Procedure
CREATE PROCEDURE PR_Designation_SelectByPrimaryKey
    @DesignationID INT
AS
BEGIN
    SELECT * FROM Designation
    WHERE DesignationID = @DesignationID;
END;

-- Person SELECTBYPRIMARYKEY Procedure
CREATE PROCEDURE PR_Person_SelectByPrimaryKey
    @PersonID INT
AS
BEGIN
    SELECT P.*, D.DepartmentName, G.DesignationName
    FROM Person P
    LEFT JOIN Department D ON P.DepartmentID = D.DepartmentID
    LEFT JOIN Designation G ON P.DesignationID = G.DesignationID
    WHERE P.PersonID = @PersonID;
END;
```

3. Department, Designation & Person Table's (If foreign key is available then do write join and take columns on select list)

```
-- Department SelectAllWithDetails Procedure
CREATE PROCEDURE PR_Department_SelectAllWithDetails
AS
BEGIN
    SELECT * FROM Department
END;


-- Designation SelectAllWithDetails Procedure
CREATE PROCEDURE PR_Designation_SelectAllWithDetails
AS
BEGIN
    SELECT * FROM Designation
END;



-- Person SelectAllWithDetails Procedure
CREATE PROCEDURE PR_Person_SelectAllWithDetails
AS
BEGIN
    SELECT P.PersonID, P.FirstName, P.LastName, P.Salary, P.JoiningDate, D.DepartmentName,
G.DesignationName
    FROM Person P
    LEFT JOIN Department D ON P.DepartmentID = D.DepartmentID
    LEFT JOIN Designation G ON P.DesignationID = G.DesignationID;
END;
```

4. Create a Procedure that shows details of the first 3 persons.

```
CREATE PROCEDURE PR_Person_ShowFirstThree
AS
BEGIN
    SELECT TOP 3 * FROM Person;
END;
```

Part – B

5. Create a Procedure that takes the department name as input and returns a table with all workers working in that department.

```
CREATE PROCEDURE PR_Person_GetWorkersByDepartment
    @DepartmentName VARCHAR(100)
AS
BEGIN
    SELECT P.*
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    WHERE D.DepartmentName = @DepartmentName;
END;
```

6. Create Procedure that takes department name & designation name as input and returns a table with worker's first name, salary, joining date & department name.

```
CREATE PROCEDURE PR_Person_GetWorkersByDeptAndDesig
    @DepartmentName VARCHAR(100),
    @DesignationName VARCHAR(100)
AS
BEGIN
    SELECT P.FirstName, P.Salary, P.JoiningDate, D.DepartmentName
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    JOIN Designation G ON P.DesignationID = G.DesignationID
    WHERE D.DepartmentName = @DepartmentName AND G.DesignationName = @DesignationName;
END;
```

7. Create a Procedure that takes the first name as an input parameter and display all the details of the worker with their department & designation name.

```
CREATE PROCEDURE PR_Person_GetWorkerDetailsByFirstName
    @FirstName VARCHAR(100)
AS
BEGIN
    SELECT P.*, D.DepartmentName, G.DesignationName
    FROM Person P
    LEFT JOIN Department D ON P.DepartmentID = D.DepartmentID
    LEFT JOIN Designation G ON P.DesignationID = G.DesignationID
    WHERE P.FirstName = @FirstName;
END;
```

8. Create Procedure which displays department wise maximum, minimum & total salaries.

```
CREATE PROCEDURE PR_Department_GetSalaryStats
AS
BEGIN
    SELECT D.DepartmentName, MAX(P.Salary) AS MaxSalary, MIN(P.Salary) AS MinSalary,
SUM(P.Salary) AS TotalSalary
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    GROUP BY D.DepartmentName;
END;
```

9. Create Procedure which displays designation wise average & total salaries.

```
CREATE PROCEDURE PR_Designation_GetSalaryStats
AS
BEGIN
    SELECT G.DesignationName, AVG(P.Salary) AS AvgSalary, SUM(P.Salary) AS TotalSalary
    FROM Person P
    JOIN Designation G ON P.DesignationID = G.DesignationID
    GROUP BY G.DesignationName;
```

```
END;
```

Part – C

10. Create Procedure that Accepts Department Name and Returns Person Count.

```
CREATE PROCEDURE PR_Department_GetPersonCount
    @DepartmentName VARCHAR(100)
AS
BEGIN
    SELECT COUNT(*) AS PersonCount
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    WHERE D.DepartmentName = @DepartmentName;
END;
```

11. Create a procedure that takes a salary value as input and returns all workers with a salary greater than 25000 along with their department and designation details.

```
CREATE PROCEDURE PR_Person_GetWorkersWithSalaryAbove
    @Salary DECIMAL(8,2)
AS
BEGIN
    SELECT P.*, D.DepartmentName, G.DesignationName
    FROM Person P
    INNER JOIN Department D ON P.DepartmentID = D.DepartmentID
    INNER JOIN Designation G ON P.DesignationID = G.DesignationID
    WHERE P.Salary = @Salary AND P.Salary > 25000;
END;
```

12 Create a procedure to find the department(s) with the highest total salary among all departments.

```
CREATE PROCEDURE PR_Department_GetHighestTotalSalary
AS
BEGIN
    SELECT TOP 1 D.DepartmentName, SUM(P.Salary) AS TotalSalary
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    GROUP BY D.DepartmentName
    ORDER BY TotalSalary DESC;
END;
```

13. Create a procedure that takes a designation name as input and returns a list of all workers under that designation who joined within the last 10 years, along with their department.

```
CREATE PROCEDURE PR_Designation_GetRecentWorkers
    @DesignationName VARCHAR(100)
AS
BEGIN
    SELECT P.*, D.DepartmentName
    FROM Person P
    JOIN Designation G ON P.DesignationID = G.DesignationID
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    WHERE G.DesignationName = @DesignationName AND datediff(YEAR,p.JoiningDate,GETDATE())>10
END;
```

14. Create a procedure to list the number of workers in each department who do not have a designation assigned.

```
CREATE PROCEDURE PR_Department_GetWorkersWithoutDesignation
AS
BEGIN
    SELECT D.DepartmentName, COUNT(P.PersonID) AS WorkerCount
    FROM Person P
```

```
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    WHERE P.DesignationID IS NULL
    GROUP BY D.DepartmentName;
END;
```

15. Create a procedure to retrieve the details of workers in departments where the average salary is above 12000.

```
CREATE PROCEDURE PR_Department_GetHighAvgSalaryWorkers
AS
BEGIN
    SELECT P.*, D.DepartmentName
    FROM Person P
    JOIN Department D ON P.DepartmentID = D.DepartmentID
    WHERE D.DepartmentID IN (
        SELECT DepartmentID
        FROM Person
        GROUP BY DepartmentID
        HAVING AVG(Salary) > 12000
    );
END;
```

Lab-3 (Advanced Stored Procedure)

Part – A

1. **Create Stored Procedure for Employee table As User enters either First Name or Last Name and based on this you must give EmployeeID, DOB, Gender & Hiredate.**

```
CREATE PROCEDURE PR_GetEmployeeByName (
    @FirstName VARCHAR(100) = NULL,
    @LastName VARCHAR(100) = NULL
)
AS
BEGIN
    SELECT
              EmployeeID,
              DoB,
              Gender,
              HireDate
    FROM Employee
    WHERE FirstName = @FirstName OR LastName = @LastName
END

EXEC PR_GetEmployeeByName @LastName='Doe'
```

2. **Create a Procedure that will accept Department Name and based on that gives employees list who belongs to that department.**

```
CREATE PROCEDURE PR_GetEmployeesByDepartment (
    @DepartmentName VARCHAR(100)
)
AS
BEGIN
    SELECT
              [dbo].[Employee].[EmployeeID],
              [dbo].[Employee].[FirstName],
              [dbo].[Employee].[LastName],
              [dbo].[Employee].[Salary]
    FROM
              [dbo].[Employee]
    INNER JOIN
              [dbo].[Departments]
       ON
              [dbo].[Employee].[DepartmentID] = [dbo].[Departments].[DepartmentID]
    WHERE
              [dbo].[Departments].[DepartmentName] = @DepartmentName
END

EXEC PR_GetEmployeesByDepartment 'iT'
```

3. **Create a Procedure that accepts Project Name & Department Name and based on that you must give all the project related details.**

```
CREATE PROCEDURE PR_GetProjectDetails (
    @ProjectName VARCHAR(100),
    @DepartmentName VARCHAR(100)
)
AS
BEGIN
    SELECT
              [dbo].[Projects].[ProjectID],
              [dbo].[Projects].[ProjectName],
              [dbo].[Projects].[StartDate],
```

```
            [dbo].[Projects].[EndDate],
            [dbo].[Departments].[DepartmentName]
    FROM
            [dbo].[Projects]
    INNER JOIN
            [dbo].[Departments]
       ON
            [dbo].[Projects].[DepartmentID] = [dbo].[Departments].[DepartmentID]
    WHERE
            [dbo].[Projects].ProjectName = @ProjectName AND
[dbo].[Departments].[DepartmentName] = @DepartmentName
END

EXEC PR_GetProjectDetails @ProjectName = 'Project Alpha',@DepartmentName = 'IT'
```

4. **Create a procedure that will accepts any integer and if salary is between provided integer, then those employee list comes in output.**

```
CREATE PROCEDURE PR_GetEmployeesBySalary (
    @MinSalary INT,
       @MaxSalary INT
)
AS
BEGIN
    SELECT
            EmployeeID,
            FirstName,
            LastName,
            Salary
    FROM Employee
    WHERE Salary BETWEEN @MinSalary AND @MaxSalary
END

EXEC PR_GetEmployeesBySalary  60000,100000
```

5. **Create a Procedure that will accepts a date and gives all the employees who all are hired on that date.**

```
CREATE PROCEDURE PR_GetEmployeesByHireDate (
    @HireDate DATETIME
)
AS
BEGIN
    SELECT
            EmployeeID,
            FirstName,
            LastName,
            HireDate
    FROM Employee
    WHERE HireDate = @HireDate;
END;

EXEC PR_GetEmployeesByHireDate '2010-06-15'
```

Part – B

6. **Create a Procedure that accepts Gender's first letter only and based on that employee details will be served.**

```
CREATE PROCEDURE PR_GetEmployeesByGenderInitial (
    @GenderInitial CHAR(1)
)
AS
BEGIN
    SELECT
              EmployeeID,
              FirstName,
              LastName,
              Gender
    FROM Employee
    WHERE LEFT(Gender, 1) = @GenderInitial
END

EXEC PR_GetEmployeesByGenderInitial 'M'
```

7. **Create a Procedure that accepts First Name or Department Name as input and based on that employee data will come.**

```
CREATE PROCEDURE PR_GetEmployeesByFirstNameOrDepartment (
    @FirstName VARCHAR(100) = NULL,
    @DepartmentName VARCHAR(100) = NULL
)
AS
BEGIN
    SELECT
              E.EmployeeID,
              E.FirstName,
              E.LastName,
              D.DepartmentName
    FROM
              Employee as E
    INNER JOIN
              Departments as D
       ON
              E.DepartmentID = D.DepartmentID
    WHERE
              E.FirstName = @FirstName
       OR
              D.DepartmentName = @DepartmentName
END

EXEC PR_GetEmployeesByFirstNameOrDepartment 'John'
```

8. **Create a procedure that will accepts location, if user enters a location any characters, then he/she will get all the departments with all data.**

```
CREATE PROCEDURE PR_GetDepartmentsByLocation (
    @Location VARCHAR(100)
)
AS
BEGIN
    SELECT
              DepartmentID,
              DepartmentName,
              ManagerID,
              Location
    FROM Departments
```

```
    WHERE Location LIKE '%' + @Location + '%'
END

EXEC PR_GetDepartmentsByLocation 'New York'
```

Part – C

9. **Create a procedure that will accepts From Date & To Date and based on that he/she will retrieve Project related data.**

```
CREATE PROCEDURE PR_GetProjectsByDateRange (
    @FromDate DATETIME,
    @ToDate DATETIME
)
AS
BEGIN
    SELECT
              ProjectID,
              ProjectName,
              StartDate,
              EndDate,
              DepartmentID
    FROM Projects
    WHERE StartDate >= @FromDate AND EndDate <= @ToDate;
END

EXEC PR_GetProjectsByDateRange '2022-01-01','2022-12-31'
```

10. **Create a procedure in which user will enter project name & location and based on that you must provide all data with Department Name, Manager Name with Project Name & Starting Ending Dates.**

```
CREATE PROCEDURE PR_GetProjectAndDepartmentDetails (
    @ProjectName VARCHAR(100),
    @Location VARCHAR(100)
)
AS
BEGIN
    SELECT
              P.ProjectID,
              P.ProjectName,
              P.StartDate,
              P.EndDate,
          D.DepartmentName,
              D.ManagerID,
              D.Location
    FROM
              Projects P
    INNER JOIN
              Departments D
       ON
              P.DepartmentID = D.DepartmentID
    WHERE P.ProjectName = @ProjectName AND D.Location LIKE '%' + @Location + '%'
END

EXEC PR_GetProjectAndDepartmentDetails 'Project Alpha','New York'
```

Lab-4 (UDF)

Part – A

1. **Write a function to print "hello world".**

```
CREATE FUNCTION fn_HelloWorld()
RETURNS VARCHAR(25)
AS
BEGIN
	RETURN 'hello world'
END

SELECT dbo.fn_HelloWorld()
```

2. **Write a function which returns addition of two numbers.**

```
CREATE FUNCTION fn_AdditionOfTwoNumbers
(
	@num1 INT,
	@num2 INT
)
RETURNS INT
AS
BEGIN
	return @num1+@num2
END

SELECT dbo.fn_AdditionOfTwoNumbers(3,4)
```

3. **Write a function to check whether the given number is ODD or EVEN.**

```
CREATE FUNCTION fn_NumberIsOddOrEven
(
	@num INT
)
RETURNS  VARCHAR(10)
AS
BEGIN
	DECLARE @ans AS VARCHAR(50)
	IF @num % 2 = 0
		SET @ans='EVEN';
	ELSE
		SET @ans ='ODD';
	return @ans
END

SELECT dbo.fn_NumberIsOddOrEven(5)
```

4. **Write a function which returns a table with details of a person whose first name starts with B.**

```
CREATE FUNCTION fn_DetailsOfPersonStartsWithB()
RETURNS TABLE
AS
	return (SELECT * FROM Person)

SELECT * FROM dbo.fn_DetailsOfPersonStartsWithB()
```

5. **Write a function which returns a table with unique first names from the person table.**

```
CREATE FUNCTION fn_TableWithUniqueFirstName()
RETURNS TABLE
AS
	RETURN ( SELECT DISTINCT FirstName FROM Person)

SELECT * FROM dbo.fn_TableWithUniqueFirstName()
```

6. **Write a function to print number from 1 to N. (Using while loop)**

```
CREATE FUNCTION fn_1toNnumbers
( @num int)
RETURNS VARCHAR(500)
AS
BEGIN
	DECLARE @ans AS VARCHAR(MAX)
	SET @ans=' '
	DECLARE @i AS INT
	SET @i=1
	WHILE @i<=@num
	BEGIN
		SET @ans=@ans+CAST(@i AS VARCHAR)+' '
		SET @i=@i+1
	END
	RETURN @ans
END

SELECT dbo.fn_1toNnumbers(5)
```

7. **Write a function to find the factorial of a given integer.**

```
CREATE FUNCTION fn_FindFactorial
(	@num INT )
RETURNS INT
AS
BEGIN
	DECLARE @ans as INT
	SET @ans=1;
	DECLARE @i AS INT;
	SET @i=1;
	WHILE @i<=@num
	BEGIN
		SET @ans = @ans*@i;
		SET @i = @i + 1;
	END
	RETURN @ans
END

SELECT dbo.fn_FindFactorial(5)
```

Part – B

8. **Write a function to compare two integers and return the comparison result. (Using Case statement)**

```
CREATE FUNCTION fn_CompareIntegers(@Num1 INT, @Num2 INT)
RETURNS VARCHAR(20)
AS
BEGIN
    RETURN CASE
               WHEN @Num1 > @Num2 THEN 'First is greater'
               WHEN @Num1 < @Num2 THEN 'Second is greater'
               ELSE 'Both are equal'
           END
END

SELECT dbo.fn_CompareIntegers(1,2)
```

9. **Write a function to print the sum of even numbers between 1 to 20.**

```
CREATE FUNCTION fn_SumOfEvens()
RETURNS INT
AS
BEGIN
    DECLARE @ans INT = 0;
```

```
    DECLARE @i INT = 2;

    WHILE @i <= 20
    BEGIN
        SET @ans = @ans + @i;
        SET @i = @i + 2;
    END

    RETURN @ans
END

SELECT dbo.fn_SumOfEvens();
```

10. Write a function that checks if a given string is a palindrome.

```
CREATE FUNCTION fn_IsPalindrome(@Text VARCHAR(100))
RETURNS VARCHAR(20)
AS
BEGIN
    DECLARE @ReversedText VARCHAR(100) = REVERSE(@Text);

    RETURN CASE
               WHEN @Text = @ReversedText THEN 'Palindrome'
               ELSE 'Not a Palindrome'
           END
END

SELECT dbo.fn_IsPalindrome('NayaN')
```

Part – C

11. Write a function to check whether a given number is prime or not.

```
CREATE FUNCTION fn_IsPrime(@num INT)
RETURNS VARCHAR(20)
AS
BEGIN
    DECLARE @i INT = 2;

    IF @num <= 1
        RETURN 'Not Prime';

    WHILE @i <= @num/2
    BEGIN
        IF @num % @i = 0
            RETURN 'Not Prime';

        SET @i = @i + 1;
    END

    RETURN 'Prime'
END
SELECT dbo.fn_IsPrime(19)
```

12. Write a function which accepts two parameters start date & end date, and returns a difference in days.

```
CREATE FUNCTION fn_DateDifference(@StartDate DATE, @EndDate DATE)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(DAY, @StartDate, @EndDate);
END
```

```
SELECT dbo.fn_DateDifference('2023-02-01', '2023-11-30')
```

13. Write a function which accepts two parameters year & month in integer and returns total days each year.

```
CREATE FUNCTION fn_TotalDaysInMonth(@Year INT, @Month INT)
RETURNS INT
AS
BEGIN
    RETURN DAY(EOMONTH(DATEFROMPARTS(@Year, @Month, 1)));
END;
SELECT dbo.fn_TotalDaysInMonth(2024, 11);
```

14. Write a function which accepts departmentID as a parameter & returns a detail of the persons.

```
CREATE FUNCTION fn_GetPersonsByDepartmentID(@DepartmentID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT * FROM Person
    WHERE DepartmentID = @DepartmentID
)
SELECT * FROM fn_GetPersonsByDepartmentID(2);
```

15. Write a function that returns a table with details of all persons who joined after 1-1-1991.

```
CREATE FUNCTION fn_GetPersonsJoinedAfter1991()
RETURNS TABLE
AS
RETURN
(
    SELECT * FROM Person
    WHERE JoiningDate > '1991-01-01'
)
SELECT * FROM fn_GetPersonsJoinedAfter1991();
```

Lab-5 (Trigger)

Part – A

1. **1. Create a trigger that fires on INSERT, UPDATE and DELETE operation on the PersonInfo table to display a message "Record is Affected."**

```
CREATE TRIGGER tr_PersonInfo_RecordAffected
ON PersonInfo
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
     PRINT 'Record is Affected.'
END
```

2. **Create a trigger that fires on INSERT, UPDATE and DELETE operation on the PersonInfo table. For that, log all operations performed on the person table into PersonLog.**

```
-- a. Trigger for Insert Operation
CREATE TRIGGER tr_Person_after_Insert
ON PersonInfo
AFTER INSERT
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(50);
     SELECT @PersonID = PersonID FROM inserted
     SELECT @PersonName = PersonName FROM inserted
     INSERT INTO PersonLog
     VALUES
     (@PersonID,@PersonName,'INSERT',GETDATE())
END

-- b. Trigger for Update Operation
Create TRIGGER tr_Person_after_Update
ON PersonInfo
AFTER UPDATE
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(50);
     SELECT @PersonID = PersonID,
            @PersonName = PersonName
     FROM inserted;
     INSERT INTO PersonLog
     VALUES (@PersonID, @PersonName, 'UPDATE', GETDATE())
END

-- c. Trigger for Delete Operation
Create TRIGGER tr_Person_after_Delete
ON PersonInfo
AFTER DELETE
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(50);
     SELECT @PersonID = PersonID,
            @PersonName = PersonName
     FROM deleted;
     INSERT INTO PersonLog
     VALUES (@PersonID, @PersonName, 'DELETE', GETDATE())
END
```

3. **Create an INSTEAD OF trigger that fires on INSERT, UPDATE and DELETE operation on the PersonInfo table. For that, log all operations performed on the person table into PersonLog.**

```
-- a. Trigger for Insert Operation
Create TRIGGER tr_Person_InsteadOf_Insert
ON PersonInfo
Instead of INSERT
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(100);
     SELECT @PersonID = PersonID From inserted
     SELECT @PersonName = PersonName FROM inserted;
     INSERT INTO PersonLog (PersonID, PersonName, Operation, UpdateDate)
     VALUES (@PersonID, @PersonName, 'INSERT', GETDATE());
END


-- b. Trigger for Update Operation
Create TRIGGER tr_Person_InsteadOf_Update
ON PersonInfo
Instead Of UPDATE
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(50);
     SELECT @PersonID = PersonID,
            @PersonName = PersonName
     FROM inserted
     INSERT INTO PersonLog
     VALUES (@PersonID, @PersonName, 'UPDATE', GETDATE())
END

-- c. Trigger for Delete Operation
Create TRIGGER tr_Person_InsteadOf_Delete
ON PersonInfo
Instead Of DELETE
AS
BEGIN
     DECLARE @PersonID INT;
     DECLARE @PersonName VARCHAR(50);
     SELECT @PersonID = PersonID,
            @PersonName = PersonName
     FROM deleted;
     INSERT INTO PersonLog
     VALUES (@PersonID, @PersonName, 'DELETE', GETDATE())
END
```

4. **Create a trigger that fires on INSERT operation on the PersonInfo table to convert person name into uppercase whenever the record is inserted.**

```
CREATE TRIGGER tr_Person_NameUpper_Inset
ON PersonInfo
AFTER INSERT
AS
BEGIN
       DECLARE @Uname VARCHAR(50)
       DECLARE @PersonID int
       select @Uname=PersonName from inserted
       select @PersonID=PersonID from inserted
       UPDATE PersonInfo
       SET PersonName=Upper(@Uname)
       WHERE PersonID=@PersonID
END
```

5. **Create trigger that prevent duplicate entries of person name on PersonInfo table.**

```
CREATE TRIGGER tr_PersonInfo_PreventDuplicateName
ON PersonInfo
INSTEAD OF INSERT
AS
BEGIN
     INSERT INTO PersonInfo
     (PersonID, PersonName, Salary, JoiningDate, City, Age,   BirthDate)
     SELECT
        PersonID,
        PersonName,
        Salary,
        JoiningDate,
        City,
        Age,
        BirthDate
    FROM inserted
    WHERE PersonName NOT IN (SELECT PersonName FROM PersonInfo)
END
```

6. **Create trigger that prevent Age below 18 years.**

```
CREATE TRIGGER tr_PersonInfo_PreventUnderage
ON PersonInfo
INSTEAD OF INSERT
AS
BEGIN
      INSERT INTO PersonInfo
      (PersonID, PersonName, Salary, JoiningDate, City, Age, BirthDate)
      SELECT
            PersonID,
            PersonName,
            Salary,
            JoiningDate,
            City,
            Age,
            BirthDate
      FROM inserted
      WHERE Age >= 18;
END
```

Part – B

7. **Create a trigger that fires on INSERT operation on person table, which calculates the age and update that age in Person table.**

```
CREATE TRIGGER tr_Person_CalculateAge
ON PersonInfo
AFTER INSERT
AS
BEGIN
     DECLARE @PersonID AS INT,
             @BirthDate AS DATETIME
     SELECT @PersonID = PersonID,
            @BirthDate = BirthDate
     FROM inserted
     UPDATE PersonInfo
     SET Age = DATEDIFF(YEAR, @BirthDate, GETDATE())
     WHERE PersonID = @PersonID
END
```

8. **Create a Trigger to Limit Salary Decrease by a 10%.**

```
CREATE TRIGGER tr_Person_LimitSalaryDecrease
ON PersonInfo
AFTER UPDATE
AS
BEGIN
     DECLARE @OldSalary DECIMAL(8,2),
             @NewSalary DECIMAL(8,2),
             @PersonID INT;
     SELECT  @OldSalary = Salary,
             @PersonID = PersonID
     FROM deleted;

     SELECT  @NewSalary = Salary
     FROM inserted
     WHERE PersonID = @PersonID;

     IF @NewSalary < @OldSalary * 0.9
     BEGIN
          UPDATE PersonInfo
          SET Salary = @OldSalary
          WHERE PersonID = @PersonID;
     END
END;
```

Part – C

9. **Create Trigger to Automatically Update JoiningDate to Current Date on INSERT if JoiningDate is NULL during an INSERT.**

```
CREATE TRIGGER tr_Person_UpdateJoiningDate
ON PersonInfo
AFTER INSERT
AS
BEGIN
     UPDATE PersonInfo
     SET JoiningDate = GETDATE()
     FROM PersonInfo p
     INNER JOIN inserted i
     ON p.PersonID = i.PersonID
     WHERE i.JoiningDate IS NULL
END
```

10. **Create DELETE trigger on PersonLog table, when we delete any record of PersonLog table it prints 'Record deleted successfully from PersonLog'.**

```
CREATE TRIGGER tr_PersonLog_Delete
ON PersonLog
AFTER DELETE
AS
BEGIN
    PRINT 'Record deleted successfully from PersonLog';
END
```

Lab-6 (Cursor)

Part – A

1. **Create a cursor Product_Cursor to fetch all the rows from a products table.**

```
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);

DECLARE Product_Cursor CURSOR
FOR SELECT
    Product_id,
    Product_Name,
    Price
FROM
    Products;

OPEN Product_Cursor;

FETCH NEXT FROM Product_Cursor INTO
    @ProductID,
    @ProductName,
    @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    --SELECT @ProductID AS ProductID, @ProductName AS ProductName, @Price AS Price;
    PRINT CAST(@ProductID AS VARCHAR)+'-'+ @ProductName +'-'+ CAST(@Price AS VARCHAR);
    FETCH NEXT FROM Product_Cursor INTO
        @ProductID,
        @ProductName,
        @Price;
END
CLOSE Product_Cursor
DEALLOCATE Product_Cursor
```

2. **Create a cursor Product_Cursor_Fetch to fetch the records in form of ProductID_ProductName. (Example: 1_Smartphone)**

```
DECLARE Product_Cursor_Fetch CURSOR
FOR SELECT
    CAST(Product_id AS VARCHAR) + '_' + Product_Name AS ProductInfo
FROM
    Products;

OPEN Product_Cursor_Fetch;

DECLARE @ProductInfo VARCHAR(300);

FETCH NEXT FROM Product_Cursor_Fetch INTO @ProductInfo;

WHILE @@FETCH_STATUS = 0
BEGIN
    Print @ProductInfo;
    FETCH NEXT FROM Product_Cursor_Fetch INTO @ProductInfo;
END;

CLOSE Product_Cursor_Fetch;
DEALLOCATE Product_Cursor_Fetch;
```

3. **Create a Cursor to Find and Display Products Above Price 30,000.**

```
DECLARE @ProductID INT
DECLARE Product_CursorDelete CURSOR
FOR SELECT
    Product_id
FROM
    Products;

OPEN Product_CursorDelete;

FETCH NEXT FROM Product_CursorDelete INTO @ProductID;

WHILE @@FETCH_STATUS = 0
BEGIN
    DELETE FROM Products WHERE Product_id = @ProductID;
    FETCH NEXT FROM Product_CursorDelete INTO @ProductID;
END;

CLOSE Product_CursorDelete;
DEALLOCATE Product_CursorDelete;
```

4. **Create a cursor Product_CursorDelete that deletes all the data from the Products table.**

```
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);
DECLARE Product_Cursor_Above_30000 CURSOR
FOR SELECT
    Product_id,
    Product_Name,
    Price
FROM
    Products
WHERE
    Price > 30000;

OPEN Product_Cursor_Above_30000;

FETCH NEXT FROM Product_Cursor_Above_30000 INTO @ProductID, @ProductName, @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    SELECT @ProductID AS ProductID, @ProductName AS ProductName, @Price AS Price;
    FETCH NEXT FROM Product_Cursor_Above_30000 INTO @ProductID, @ProductName, @Price;
END;

CLOSE Product_Cursor_Above_30000;
DEALLOCATE Product_Cursor_Above_30000;
```

Part – B

5. **Create a cursor Product_CursorUpdate that retrieves all the data from the products table and increases the price by 10%.**

```
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);
DECLARE Product_CursorUpdate CURSOR
FOR SELECT
    Product_id,
    Price
FROM
    Products;

OPEN Product_CursorUpdate;

FETCH NEXT FROM Product_CursorUpdate INTO @ProductID, @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE Products
    SET Price = Price * 1.10
    WHERE Product_id = @ProductID;

    FETCH NEXT FROM Product_CursorUpdate INTO @ProductID, @Price;
END;

CLOSE Product_CursorUpdate;
DEALLOCATE Product_CursorUpdate;
```

6. **Create a Cursor to Rounds the price of each product to the nearest whole number.**

```
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);
DECLARE Product_Cursor_Round CURSOR
FOR SELECT
    Product_id,
    Price
FROM
    Products;

OPEN Product_Cursor_Round;

FETCH NEXT FROM Product_Cursor_Round INTO @ProductID, @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE Products
    SET Price = ROUND(Price, 0)
    WHERE Product_id = @ProductID;

    FETCH NEXT FROM Product_Cursor_Round INTO @ProductID, @Price;
END;

CLOSE Product_Cursor_Round;
DEALLOCATE Product_Cursor_Round;
```

Part – C

7. **Create a cursor to insert details of Products into the NewProducts table if the product is "Laptop" (Note: Create NewProducts table first with same fields as Products table)**

```
CREATE TABLE NewProducts (
    Product_id INT ,
    Product_Name VARCHAR(250) NOT NULL,
    Price DECIMAL(10, 2) NOT NULL
);
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);
DECLARE Product_Cursor_Insert_Laptop CURSOR
FOR SELECT
    Product_id,
    Product_Name,
    Price
FROM
    Products
WHERE
    Product_Name = 'Laptop';

OPEN Product_Cursor_Insert_Laptop;

FETCH NEXT FROM Product_Cursor_Insert_Laptop INTO @ProductID, @ProductName, @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    INSERT INTO NewProducts (Product_id, Product_Name, Price)
    VALUES (@ProductID, @ProductName, @Price);

    FETCH NEXT FROM Product_Cursor_Insert_Laptop INTO @ProductID, @ProductName, @Price;
END;

CLOSE Product_Cursor_Insert_Laptop;
DEALLOCATE Product_Cursor_Insert_Laptop;
```

8. **Create a Cursor to Archive High-Price Products in a New Table (ArchivedProducts), Moves products with a price above 50000 to an archive table, removing them from the original Products table.**

```
CREATE TABLE ArchivedProducts (
    Product_id INT PRIMARY KEY,
    Product_Name VARCHAR(250) NOT NULL,
    Price DECIMAL(10, 2) NOT NULL
);
DECLARE
    @ProductID INT,
    @ProductName VARCHAR(250),
    @Price DECIMAL(10, 2);

DECLARE Product_Cursor_Archive CURSOR
FOR SELECT
    Product_id,
    Product_Name,
    Price
FROM
    Products
WHERE
    Price > 50000;

OPEN Product_Cursor_Archive;
```

```
FETCH NEXT FROM Product_Cursor_Archive INTO @ProductID, @ProductName, @Price;

WHILE @@FETCH_STATUS = 0
BEGIN
    INSERT INTO ArchivedProducts (Product_id, Product_Name, Price)
    VALUES (@ProductID, @ProductName, @Price);

    DELETE FROM Products WHERE Product_id = @ProductID;

    FETCH NEXT FROM Product_Cursor_Archive INTO @ProductID, @ProductName, @Price;
END;

CLOSE Product_Cursor_Archive;
DEALLOCATE Product_Cursor_Archive;
```

Lab-7 (Exception Handling)

Part – A

1. **Handle Divide by Zero Error and Print message like: Error occurs that is - Divide by zero error.**

```
BEGIN TRY
      DECLARE @VAL1 INT;
      DECLARE @VAL2 INT;
      SET @VAL1 = 8;
      SET @VAL2 = 0;
      PRINT @VAL1/@VAL2;
END TRY
BEGIN CATCH
      PRINT 'ERROR OCCURS THAT IS DIVIDED BY ZERO'
END CATCH
```

2. **Try to convert string to integer and handle the error using try…catch block.**

```
SELECTBEGIN TRY
     SELECT CAST('abc' AS INT)
END TRY
BEGIN CATCH
     SELECT ERROR_NUMBER() AS ErrorNumber,
            ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

3. **Create a procedure that prints the sum of two numbers: take both numbers as integer & handle exception with all error functions if any one enters string value in numbers otherwise print result.**

```
CREATE PROCEDURE CalculateSum
     @NUM1 INT
     @NUM2 INT
AS
BEGIN
     IF ISNUMERIC(@NUM1)=1 AND ISNUMERIC(@NUM2)=1
        PRINT @NUM1 + @NUM2
END
BEGIN TRY
     EXEC CalculateSum 10,'ABC'
END TRY
BEGIN CATCH
     SELECT ERROR_NUMBER() AS 'Error Number',
            ERROR_MESSAGE() AS 'Error Message',
            ERROR_LINE() AS 'Error Line',
            ERROR_SEVERITY() AS 'Error Severity',
            ERROR_PROCEDURE() AS 'Error Proc',
            ERROR_STATE() AS 'Error State',
END CATCH
```

4. **Handle a Primary Key Violation while inserting data into customers table and print the error details such as the error message, error number, severity, and state.**

```
BEGIN TRY
     INSERT INTO CUSTOMERS VALUES (1, 'JOHN DOE' , 'JD@GMAIL.COM')
     INSERT INTO CUSTOMERS VALUES (1, 'JANE SMITH' , 'JS@GMAIL.COM')
END TRY
BEGIN CATCH
     SELECT ERROR_NUMBER() AS 'Error Number',
            ERROR_MESSAGE() AS 'Error Message',
            ERROR_LINE() AS 'Error Line',
            ERROR_SEVERITY() AS 'Error Severity',
            ERROR_STATE() AS 'Error State';
```

```
END CATCH
```

5. **Throw custom exception using stored procedure which accepts Customer_id as input & that throws Error like no Customer_id is available in database.**

```
CREATE PROCEDURE SP_CUSTID
     @CUSTOMER_ID INT
AS
BEGIN TRY
     IF EXISTS (SELECT * FROM CUSTOMERS WHERE CUSTOMER_ID = @CUSTOMER_ID)
          PRINT 'CUSTOMER ID IS AVAILABLE'
     ELSE
          THROW 500002 , 'No Customer Id is avalilable' , 1;
END TRY
BEGIN CATCH
     PRINT 'ERROR MESSAGE : ' + ERROR_MESSAGE()
END CATCH
```

Part – B

6. **Handle a Foreign Key Violation while inserting data into Orders table and print appropriate error message.**

```
BEGIN TRY
     INSERT INTO ORDERS VALUES ( 1 , 6 , '2025-06-11' )
END TRY
BEGIN CATCH
     PRINT 'ERROR MESSAGE : ' + ERROR_MESSAGE()
END CATCH
```

7. **Throw custom exception that throws error if the data is invalid.**

```
CREATE PROCEDURE SP_INVALID_DATA
      @CID INT ,
      @CNAME VARCHAR(50) ,
      @EMAIL VARCHAR(50)
AS
BEGIN TRY
      IF @CID<0
           THROW 500003 , 'Invalid Data' , 2;
      ELSE
           INSERT INTO CUSTOMERS VALUES ( @CID , @CNAME , @EMAIL )
END TRY
BEGIN CATCH
      PRINT 'ERROR MESSAGE : ' + ERROR_MESSAGE()
END CATCH
```

8. **Create a Procedure to Update Customer's Email with Error Handling.**

```
CREATE PROCEDURE PR_CUSTOMER_UPDATE_EMAIL
     @CID INT,
     @NEWEMAIL VARCHAR(50)
AS
BEGIN
     BEGIN TRY
          UPDATE CUSTOMERS
          SET EMAIL = @NEWEMAIL
          WHERE CUSTOMER_ID = @CID
     END TRY
     BEGIN CATCH
         SELECT ERROR_NUMBER() AS 'Error Number',
            ERROR_MESSAGE() AS 'Error Message',
            ERROR_LINE() AS 'Error Line',
            ERROR_SEVERITY() AS 'Error Severity',
```

```
            ERROR_STATE() AS 'Error State';
    END CATCH
END;
```

Part – C

9. **Create a procedure which prints the error message that "The Customer_id is already taken. Try another one"**

```
CREATE PROCEUDRE PR_CUSTOMER_INSERT
      @CUSTOMER_ID INT ,
      @CUSTOMER_NAME VARCHAR(50) ,
      @EMAIL VARCHAR(50)
AS
BEGIN TRY
      INSERT INTO CUSTOMERS VALUES ( @CUSTOMER_ID , @CUSTOMER_NAME , @EMAIL )
END TRY
BEGIN CATCH
      PRINT 'The Customer_id is already taken. Try another one'
END CATCH
```

10. **Handle Duplicate Email Insertion in Customers Table.**

```
CREATE PROCEDURE PR_CUSTOMER_HANDLEDUPLICATEEMAIL
      @CUSTOMER_ID INT ,
      @CUSTOMER_NAME VARCHAR(50) ,
      @EMAIL VARCHAR(50)
AS
BEGIN
      IF EXISTS ( SELECT 1 FROM CUSTOMERS WHERE EMAIL = @EMAIL )
            THROW 50003 ,  'The Email is already taken' , 12;
      ELSE
      BEGIN
            INSERT INTO CUSTOMERS VALUES ( @CUSTOMER_ID , @CUSTOMER_NAME , @EMAIL )
            PRINT 'Customer record inserted successfully'
      END
END
```

LAB – 8 (createcollection, dropcollection and insert method)

Part – A

1. **Create a new database named "Darshan".**
   use Darshan
2. **Create another new database named "DIET".**
   use DIET
3. **List all databases**
   show databases
4. **Check the current database.**
   db
5. **Drop "DIET" database.**
   use DIET
   db.dropDatabase()
6. **Create a collection named "Student" in the "Darshan" database**
   use Darshan
   db.createCollection("Student")
7. **Create a collection named "Department" in the "Darshan" database.**
   db.createCollection("Department")
8. **List all collections in the "Darshan" database.**
   show collections
9. **Insert a single document using insertOne into "Department" collection. (Dname:'CE', HOD:'Patel').**
   db.Department.insertOne({ Dname: 'CE', HOD: 'Patel' })
10. **Insert two document using insertMany into "Department" collection. (Dname:'IT' and Dname:'ICT')**
    db.Department.insertMany([{ Dname: 'IT' }, { Dname: 'ICT' }])
11. **Drop a collection named "Department" from the "Darshan" database.**
    db.Department.drop()
12. **Insert a single document using insertOne into "Student" collection. (Fields are Name, City, Branch, Semester, Age) Insert your own data.**
    ```
    db.Student.insertOne({
      Name: 'Mann',
      City: 'Junagadh',
      Branch: 'CSE',
      Semester: '6',
      Age: 20
    })
    ```
13. **Insert three documents using insertMany into "Student" collection. (Fields are Name, City, Branch, Semester, Age) Insert your three friend's data.**
    ```
    db.Student.insertMany([
      { Name: 'Friend1', City: 'City1', Branch: 'Branch1', Semester: 'Sem1', Age: 18 },
      { Name: 'Friend2', City: 'City2', Branch: 'Branch2', Semester: 'Sem2', Age: 19 },
      { Name: 'Friend3', City: 'City3', Branch: 'Branch3', Semester: 'Sem3', Age: 20 }
    ])
    ```
14. **Check whether "Student" collection exists or not.**
    db.getCollectionNames().includes("Student")
15. **Check the stats of "Student" collection.**
    db.Student.stats()
16. **Drop the "Student" collection.**
    db.Student.drop()
17. **Create a collection named "Deposit"**
    db.createCollection("Deposit")
18. **Insert following data in to "Deposit" collection.**
    ```
    db.Deposit.insertMany([
      { ACTNO: 101, CNAME: 'ANIL', BNAME: 'VRCE', AMOUNT: 1000.00, CITY: 'RAJKOT' },
    ```

```
    { ACTNO: 102, CNAME: 'SUNIL', BNAME: 'AJNI', AMOUNT: 5000.00, CITY: 'SURAT' },
    { ACTNO: 103, CNAME: 'MEHUL', BNAME: 'KAROLBAGH', AMOUNT: 3500.00, CITY: 'BARODA' },
    { ACTNO: 104, CNAME: 'MADHURI', BNAME: 'CHANDI', AMOUNT: 1200.00, CITY: 'AHMEDABAD' },
    { ACTNO: 105, CNAME: 'PRMOD', BNAME: 'M.G. ROAD', AMOUNT: 3000.00, CITY: 'SURAT' },
    { ACTNO: 106, CNAME: 'SANDIP', BNAME: 'ANDHERI', AMOUNT: 2000.00, CITY: 'RAJKOT' },
    { ACTNO: 107, CNAME: 'SHIVANI', BNAME: 'VIRAR', AMOUNT: 1000.00, CITY: 'SURAT' },
    { ACTNO: 108, CNAME: 'KRANTI', BNAME: 'NEHRU PLACE', AMOUNT: 5000.00, CITY: 'RAJKOT' }
  ])
```

19. DI all the documents of "Deposit" collection.

```
db.Deposit.find()
```

20. Drop the "Deposit" collection.

```
db.Deposit.drop()
```

Part – B

1. Create a new database named "Computer".

```
use Computer
```

2. Create a collection named "Faculty" in the "Computer" database.

```
db.createCollection("Faculty")
```

3. Insert a below document using insertOne into "Faculty" collection.

```
db.Faculty.insertOne({
  FID: 1, FNAME: 'ANIL', BNAME: 'CE', SALARY: 10000, JDATE: '1-3-95'
 })
```

4. Insert below documents using insertMany into "Faculty" collection.

```
db.Faculty.insertMany([
    { FID: 2, FNAME: 'SUNIL', BNAME: 'CE', SALARY: 50000, JDATE: '4-1-96' },
    { FID: 3, FNAME: 'MEHUL', BNAME: 'IT', SALARY: 35000, JDATE: '17-11-95' },
    { FID: 4, FNAME: 'MADHURI', BNAME: 'IT', SALARY: 12000, JDATE: '17-12-95' },
    { FID: 5, FNAME: 'PRMOD', BNAME: 'CE', SALARY: 30000, JDATE: '27-3-96' },
    { FID: 6, FNAME: 'SANDIP', BNAME: 'CE', SALARY: 20000, JDATE: '31-3-96' },
    { FID: 7, FNAME: 'SHIVANI', BNAME: 'CE', SALARY: 10000, JDATE: '5-9-95' },
    { FID: 8, FNAME: 'KRANTI', BNAME: 'IT', SALARY: 50000, JDATE: '2-7-95' }
  ])
```

5. Display all the documents of "Faculty" collection.

```
db.Faculty.find()
```

6. Drop the "Faculty" collection.

```
db.Faculty.drop()
```

7. Drop the "Computer" database.

```
use Computer
db.dropDatabase()
```

Part – C

1. Create a new database named "Computer".

```
db.Deposit.find()
```

2. Create a collection named "Faculty" in the "Computer" database.

```
db.Deposit.findOne()
```

3. Insert a below documents into "Faculty" collection.

```
db.Deposit.insertOne({ ACTNO: 109, CNAME: 'KIRTI', BNAME: 'VIRAR', AMOUNT: 3000, ADATE: '3-5-97' })
```

4. Display all the documents of "Faculty" collection.

```
db.Deposit.insertMany([{ ACTNO: 110, CNAME: 'MITALI', BNAME: 'ANDHERI', AMOUNT: 4500, ADATE: '4-9-95'
   }, { ACTNO: 111, CNAME: 'RAJIV', BNAME: 'NEHRU PLACE', AMOUNT: 7000, ADATE: '2-10-98' }])
```

5. Drop the "Faculty" collection.

```
db.Deposit.find({ BNAME: 'VIRAR' })
```

6. Drop the “Computer” database.

```
db.Deposit.find({ AMOUNT: { $gte: 3000, $lte: 5000 } })
```

Lab-9 (Find, limit, skip and sort method)

Part – A

1. **Retrieve/Display every document of Deposit collection.**

   db.Deposit.find()

2. **Display only one document of Deposit collection. (Use: findOne()).**

   db.Deposit.findOne()

3. **Insert following document into Deposit collection. (Use: insertOne()).**

   db.Deposit.insertOne({ ACTNO: 109, CNAME: 'KIRTI', BNAME: 'VIRAR', AMOUNT: 3000, ADATE: '3-5-97' })

4. **Insert following documents into Deposit collection. (Use: insertMany()).**

   db.Deposit.insertMany([{ ACTNO: 110, CNAME: 'MITALI', BNAME: 'ANDHERI', AMOUNT: 4500, ADATE: '4-9-95' }, { ACTNO: 111, CNAME: 'RAJIV', BNAME: 'NEHRU PLACE', AMOUNT: 7000, ADATE: '2-10-98' }])

5. **Display all the documents of 'VIRAR' branch from Deposit collection.**

   db.Deposit.find({ BNAME: 'VIRAR' })

6. **Display all the documents of Deposit collection whose amount is between 3000 and 5000.**

   db.Deposit.find({ AMOUNT: { $gte: 3000, $lte: 5000 } })

7. **Display all the documents of Deposit collection whose amount is greater than 2000 and branch is VIRAR.**

   db.Deposit.find({ AMOUNT: { $gt: 2000 }, BNAME: 'VIRAR' })

8. **Display all the documents with CNAME, BNAME and AMOUNT fields from Deposit collection.**

   db.Deposit.find({}, { CNAME: 1, BNAME: 1, AMOUNT: 1, _id: 0 })

9. **Display all the documents of Deposit collection on ascending order by CNAME.**

   db.Deposit.find().sort({ CNAME: 1 })

10. **Display all the documents of Deposit collection on descending order by BNAME.**

    db.Deposit.find().sort({ BNAME: -1 })

11. **Display all the documents of Deposit collection on ascending order by ACTNO and descending order by AMOUNT.**

    db.Deposit.find().sort({ ACTNO: 1, AMOUNT: -1 })

12. **Display only two documents of Deposit collection.**

    db.Deposit.find().limit(2)

13. **Display 3rd document of Deposit collection.**

    db.Deposit.find().skip(2).limit(1)

14. **Display 6th and 7th documents of Deposit collection.**

    db.Deposit.find().skip(5).limit(2)

15. **Display the count of documents in Deposit collection.**

db.Deposit.countDocuments()

Part – B

1. **Insert following documents into "Student" collection. (Use: insertMany())**

   { "_id": 1, "name": "John", "age": 30, "city": "New York", "isActive": true }

   { "_id": 2, "name": "Jane", "age": 25, "city": "Los Angeles", "isActive": false }

   { "_id": 3, "name": "Tom", "age": 35, "city": "Chicago", "isActive": true }

   { "_id": 4, "name": "Lucy", "age": 28, "city": "San Francisco", "isActive": true }

   { "_id": 5, "name": "David", "age": 40, "city": "Miami", "isActive": false }

   { "_id": 6, "name": "Eva", "age": 23, "city": "Boston", "isActive": true }

   { "_id": 7, "name": "Nick", "age": 38, "city": "Seattle", "isActive": false }

   { "_id": 8, "name": "Sophia", "age": 27, "city": "New York", "isActive": true }

   { "_id": 9, "name": "Liam", "age": 32, "city": "Los Angeles", "isActive": false }

   { "_id": 10, "name": "Olivia", "age": 29, "city": "San Diego", "isActive": true }

   db.Student.insertMany([{ _id: 1, name: "John", age: 30, city: "New York", isActive: true }, { _id: 2, name: "Jane", age: 25, city: "Los Angeles", isActive: false }, { _id: 3, name: "Tom", age: 35, city: "Chicago", isActive: true }, { _id: 4, name: "Lucy", age: 28, city: "San Francisco", isActive: true }, { _id: 5, name: "David", age: 40, city: "Miami", isActive: false }, { _id: 6, name: "Eva", age: 23, city: "Boston", isActive: true }, { _id: 7, name: "Nick", age: 38, city: "Seattle", isActive: false }, { _id: 8, name: "Sophia", age: 27, city: "New York", isActive: true }, { _id: 9, name: "Liam", age: 32, city: "Los Angeles", isActive: false }, { _id: 10, name: "Olivia", age: 29, city: "San Diego", isActive: true }])

2. **Display all documents of "Student" collection.**

   db.Student.find()

3. **Display all documents of "Student" collection whose age is 30.**

   db.Student.find({ age: 30 })

4. **Display all documents of "Student" collection whose age is greater than 25.**

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

5. **Display all documents of "Student" collection whose name is "John" and age is 30.**

   db.Student.find({ name: "John", age: 30 })

6. **Display all documents of "Student" collection whose age is not equal to 25.**

   db.Student.find({ age: { $ne: 25 } })

7. **Display all documents of "Student" collection whose age is equal to 25 or 30 or 35. (using $or as well as using $in).**

   db.Student.find({ $or: [{ age: 25 }, { age: 30 }, { age: 35 }] })

   db.Student.find({ age: { $in: [25, 30, 35] } })

8. **Display all documents of "Student" collection whose name is "John" or age is 30.**

   db.Student.find({ $or: [{ name: "John" }, { age: 30 }] })

9. **Display all documents of "Student" collection whose name is "John" and city is New York.**

   db.Student.find({ name: "John", city: "New York" })

10. **Display name and age of students from "Student" collection whose name is "John" and city is New York.**

    db.Student.find({ name: "John", city: "New York" }, { name: 1, age: 1, _id: 0 })

Part – C

1. **Display name of students from "Student" collection whose age is between to 25 and 35 and sort output by age in ascending order.**

   db.Student.find({ age: { $gte: 25, $lte: 35 } }, { name: 1, _id: 0 }).sort({ age: 1 })

2. **Display all documents of "Student" collection and sort all the documents by name in ascending order and then by age in descending.**

   db.Student.find().sort({ name: 1, age: -1 })

3. **Display first five documents of "Student" collection.**

   db.Student.find().limit(5)

4. **Display fourth and fifth documents of "Student" collection.**

   db.Student.find().skip(3).limit(2)

5. **Display the name of oldest student from "Student" collection.**

   db.Student.find().sort({ age: -1 }).limit(1).project({ name: 1, _id: 0 })

6. **Display all documents of "Student" collection in such a way that skip the first 2 documents and return the rest documents.**

   db.Student.find().skip(2)

Lab-10 (update, delete and rename method)

Part – A

1. **Update the age of John's to 31.**

   db.Student.updateOne({ name: "John" }, { $set: { age: 31 } })

2. **Update the city of all students from 'New York' to 'New Jersey'.**

   db.Student.updateMany({ city: "New York" }, { $set: { city: "New Jersey" } })

3. **Set isActive to false for every student older than 35.**

   db.Student.updateMany({ age: { $gt: 35 } }, { $set: { isActive: false } })

4. **Increment the age of all students by 1 year.**

   db.Student.updateMany({}, { $inc: { age: 1 } })

5. **Set the city of 'Eva' to 'Cambridge'.**

   db.Student.updateOne({ name: "Eva" }, { $set: { city: "Cambridge" } })

6. **Update 'Sophia's isActive status to false.**

   db.Student.updateOne({ name: "Sophia" }, { $set: { isActive: false } })

7. **Update the city field of student aged below 30 to 'San Diego'.**

   db.Student.updateMany({ age: { $lt: 30 } }, { $set: { city: "San Diego" } })

8. **Rename the age field to years for all documents.**

   db.Student.updateMany({}, { $rename: { "age": "years" } })

9. **Update 'Nick' to make him active (isActive = true).**

   db.Student.updateOne({ name: "Nick" }, { $set: { isActive: true } })

10. **Update all documents to add a new field country with the value 'USA'.**

    db.Student.updateMany({}, { $set: { country: "USA" } })

11. **Update 'David's city to 'Orlando'.**

    db.Student.updateOne({ name: "David" }, { $set: { city: "Orlando" } })

12. **Multiply the age of all students by 2.**

    db.Student.updateMany({}, { $mul: { years: 2 } })

13. **Unset (remove) the city field for 'Tom'.**

    db.Student.updateOne({ name: "Tom" }, { $unset: { city: "" } })

14. **Add a new field premiumUser and to true for users older than 30.**

    db.Student.updateMany({ years: { $gt: 30 } }, { $set: { premiumUser: true } })

15. **Set isActive to true for 'Jane'.**

    db.Student.updateOne({ name: "Jane" }, { $set: { isActive: true } })

16. **Update isActive field of 'Lucy' to false.**

    db.Student.updateOne({ name: "Lucy" }, { $set: { isActive: false } })

17. **Delete a document of 'Nick' from the collection.**

    db.Student.deleteOne({ name: "Nick" })

18. **Delete all students who are inactive (isActive = false).**

```
db.Student.deleteMany({ isActive: false })
```

19. **Delete all students who live in 'New York'.**

```
db.Student.deleteMany({ city: "New York" })
```

20. **Delete all the students aged above 35.**

```
db.Student.deleteMany({ years: { $gt: 35 } })
```

21. **Delete a student named "Olivia" from the collection.**

```
db.Student.deleteOne({ name: "Olivia" })
```

22. **Delete all the students whose age is below 25.**

```
db.Student.deleteMany({ years: { $lt: 25 } })
```

23. **Delete the first student whose isActive field is true.**

```
db.Student.deleteOne({ isActive: true })
```

24. **Delete all students from 'Los Angeles'.**

```
db.Student.deleteMany({ city: "Los Angeles" })
```

25. **Delete all students who have city field missing.**

```
db.Student.deleteMany({ city: { $exists: false } })
```

26. **Rename 'city' field to 'location' for all documents.**

```
db.Student.updateMany({}, { $rename: { "city": "location" } })
```

27. **Rename the name field to FullName for 'John'.**

```
db.Student.updateOne({ name: "John" }, { $rename: { "name": "FullName" } })
```

28. **Rename the isActive field to status for all documents.**

```
db.Student.updateMany({}, { $rename: { "isActive": "status" } })
```

29. **Rename age to yearsOld for everyone from 'San Francisco' student only.**

```
db.Student.updateMany({ location: "San Francisco" }, { $rename: { "years": "yearsOld" } })
```

30. **Create a Capped Collection named “Employee” as per follows:**

    a. Ecode and Ename are compulsory fields

    b. Datatype of EID is int, Ename is string, Age is int and City is string

    Insert following documents into above “Employee” collection.

    {"Ecode": 1, "Ename": "John"}

    {"Ecode ": 2, "Ename": "Jane", "age": 25, "city": "Los Angeles"}

    {"Ecode ": 3, "Ename": "Tom", "age": 35}

    {"Ecode ": 4, "Ename": "Lucy", "age": 28, "city": "San Francisco", "isActive": true}

    {"Ename": "Dino"}

```
db.createCollection("Employee", {
        capped: true,
        size: 5120,
```

```
            max: 100,
            validator: {
            $jsonSchema: {
                  bsonType: "object",
                  required: ["Ecode", "Ename"],
                        properties: {
                              Ecode: { bsonType: "int" },
                              Ename: { bsonType: "string" },
                              Age: { bsonType: "int" },
                              City: { bsonType: "string" }
                        }
                  }
            }
      })
      db.Employee.insertMany([
      { Ecode: 1, Ename: "John" },
      { Ecode: 2, Ename: "Jane", age: 25, city: "Los Angeles" },
      { Ecode: 3, Ename: "Tom", age: 35 },
      { Ecode: 4, Ename: "Lucy", age: 28, city: "San Francisco", isActive: true },
      { Ename: "Dino" }
      ])
```

Part – B

1. **Display Female students and belong to Rajkot city.**

   ```
   db.Student_data.find({ GENDER: "Female", CITY: "Rajkot" })
   ```

2. **Display students not studying in 3rd sem.**

   ```
   db.Student_data.find({ SEM: { $ne: 3 } })
   ```

3. **Display students whose city is Jamnagar or Baroda. (use: IN)**

   ```
   db.Student_data.find({ CITY: { $in: ["Jamnagar", "Baroda"] } })
   ```

4. **Display first 2 students names who lives in Baroda.**

   ```
   db.Student_data.find({ CITY: "Baroda" }).limit(2).project({ SNAME: 1, _id: 0 })
   ```

5. **Display Male students who studying in 3rd sem.**

   ```
   db.Student_data.find({ GENDER: "Male", SEM: 3 })
   ```

6. **Display sname and city and fees of those students whose roll no is less than 105.**

   ```
   db.Student_data.find({ ROLLNO: { $lt: 105 } }, { SNAME: 1, CITY: 1, FEES: 1, _id: 0 })
   ```

7. **Update City of all students from 'Jamnagar' City and Department as 'CE' to 'Surat'.**

   ```
   db.Student_data.updateMany({ CITY: "Jamnagar", DEPARTMENT: "CE" }, { $set: { CITY: "Surat" } })
   ```

8. **Increase Fees by 500 where the Gender is not 'Female'. (Use: Not)**

   db.Student_data.updateMany({ GENDER: { $ne: "Female" } }, { $inc: { FEES: 500 } })

9. **Set the Department of all students from 'EE' and in Sem 3 to 'Electrical'.**

   db.Student_data.updateMany({ DEPARTMENT: "EE", SEM: 3 }, { $set: { DEPARTMENT: "Electrical" } })

10. **Update the Fees of students in 'Rajkot' who are male.**

    db.Student_data.updateMany({ CITY: "Rajkot", GENDER: "Male" }, { $set: { FEES: 11000 } })

11. **Change City to 'Vadodara' for students in Sem 5 and with fees less than 10000.**

    db.Student_data.updateMany({ SEM: 5, FEES: { $lt: 10000 } }, { $set: { CITY: "Vadodara" } })

12. **Delete all students where the City is 'Ahmedabad' or GENDER is 'Male'.**

    db.Student_data.deleteMany({ $or: [{ CITY: "Ahmedabad" }, { GENDER: "Male" }] })

13. **Delete students whose Rollno is not in the list [101, 105, 110].**

    db.Student_data.deleteMany({ ROLLNO: { $nin: [101, 105, 110] } })

14. **Delete students from the 'Civil' department who are in Sem 5 or Sem 7.**

    db.Student_data.deleteMany({ DEPARTMENT: "Civil", SEM: { $in: [5, 7] } })

15. **Delete all students who are not in the cities 'Rajkot', 'Baroda', or 'Jamnagar'.**

    db.Student_data.deleteMany({ CITY: { $nin: ["Rajkot", "Baroda", "Jamnagar"] } })

16. **Delete students whose Rollno is between 105 and 108.**

    db.Student_data.deleteMany({ ROLLNO: { $gte: 105, $lte: 108 } })

17. **Rename the City field to LOCATION for all students.**

    db.Student_data.updateMany({}, { $rename: { "CITY": "LOCATION" } })

18. **Rename the Department field to Branch where the Fees is less than 10000.**

    db.Student_data.updateMany({ FEES: { $lt: 10000 } }, { $rename: { "DEPARTMENT": "Branch" } })

19. **Rename Sname to Fullname for students with Rollno in [106, 107, 108].**

    db.Student_data.updateMany({ ROLLNO: { $in: [106, 107, 108] } }, { $rename: { "SNAME": "Fullname" } })

20. **Rename Fees to Tuition_Fees for all students with Fees greater than 9000.**

    db.Student_data.updateMany({ FEES: { $gt: 9000 } }, { $rename: { "FEES": "Tuition_Fees" } })

21. **Rename Department to Major where the Fees is less than 15000 and Gender is 'Female'.**

    db.Student_data.updateMany({ FEES: { $lt: 15000 }, GENDER: "Female" }, { $rename: { "DEPARTMENT": "Major" } })

22. **Rename City to Hometown for all students whose SEM is 3 and Department is not 'Mechanical'.**

    db.Student_data.updateMany({ SEM: 3, DEPARTMENT: { $ne: "Mechanical" } }, { $rename: { "CITY": "Hometown" } })

Part – C

1. **Create a capped collection named" logs" with a maximum size of 100 KB and a maximum of 10 documents.**

```
db.createCollection("logs", { capped: true, size: 102400, max: 10 })
```

2. **Insert below 12 log entries into the "logs" collection. Each entry should contain a message, level (e.g., "info", "warning", "error"), and a timestamp field. Use the insertMany() method.**

   { message: "System started", level: "info", timestamp: new Date() }

   { message: "Disk space low", level: "warning", timestamp: new Date() }

   { message: "User login", level: "info", timestamp: new Date() }

   { message: "System reboot", level: "info", timestamp: new Date() }

   { message: "Error in module", level: "error", timestamp: new Date() }

   { message: "Memory usage high", level: "warning", timestamp: new Date() }

   { message: "User logout", level: "info", timestamp: new Date() }

   { message: "File uploaded", level: "info", timestamp: new Date() }

   { message: "Network error", level: "error", timestamp: new Date() }

   { message: "Backup completed", level: "info", timestamp: new Date() }

   { message: "Database error", level: "error", timestamp: new Date() }

   { message: "Service started", level: "info", timestamp: new Date() }

```
db.logs.insertMany([
{ message: "System started", level: "info", timestamp: new Date() },
{ message: "Disk space low", level: "warning", timestamp: new Date() },
{ message: "User login", level: "info", timestamp: new Date() },
{ message: "System reboot", level: "info", timestamp: new Date() },
{ message: "Error in module", level: "error", timestamp: new Date() },
{ message: "Memory usage high", level: "warning", timestamp: new Date() },
{ message: "User logout", level: "info", timestamp: new Date() },
{ message: "File uploaded", level: "info", timestamp: new Date() },
{ message: "Network error", level: "error", timestamp: new Date() },
{ message: "Backup completed", level: "info", timestamp: new Date() },
{ message: "Database error", level: "error", timestamp: new Date() },
{ message: "Service started", level: "info", timestamp: new Date() }
])
```

3. **Perform find method on "logs" collection to ensure only the last 10 documents are retained (even though you inserted 12).**

```
db.logs.find()
```

4. **Insert below 5 more documents and check if the oldest ones are automatically removed.**

   { message: "New log entry 1", level: "info", timestamp: new Date() }

   { message: "New log entry 2", level: "info", timestamp: new Date() }

   { message: "New log entry 3", level: "info", timestamp: new Date() }

{ message: "New log entry 4", level: "warning", timestamp: new Date() }

{ message: "New log entry 5", level: "error", timestamp: new Date() }

```
db.logs.insertMany([
{ message: "New log entry 1", level: "info", timestamp: new Date() },
{ message: "New log entry 2", level: "info", timestamp: new Date() },
{ message: "New log entry 3", level: "info", timestamp: new Date() },
{ message: "New log entry 4", level: "warning", timestamp: new Date() },
{ message: "New log entry 5", level: "error", timestamp: new Date() }
])
```

Lab-11 (Regex)

Part – A

1. **Find employees whose name start with E.**

   --> db.Employee.find({ ENAME: /^E/ })

2. **Find employees whose name ends with n.**

   --> db.Employee.find({ ENAME: /n$/ })

3. **Find employees whose name starts with S or M in your collection.**

   --> db.Employee.find({ ENAME: /^[SM]/ })

4. **Find employees where city starts with A to M in your collection.**

   --> db.Employee.find({ CITY: /^[A-M]/ })

5. **Find employees where city name ends in 'ney'.**

   --> db.Employee.find({ CITY: /ney$/ })

6. **Display employee info whose name contains n. (Both uppercase(N) and lowercase(n))**

   --> db.Employee.find({ ENAME: /n/i })

7. **Display employee info whose name starts with E and having 5 characters.**

   --> db.Employee.find({ ENAME: /^E.{4}$/ })

8. **Display employee whose name start with S and ends in a.**

   --> db.Employee.find({ ENAME: /^S.*a$/ })

9. **Display EID, ENAME, CITY and SALARY whose name starts with 'Phi'.**

   --> db.Employee.find({ ENAME: /^Phi/ }, { EID: 1, ENAME: 1, CITY: 1, SALARY: 1 })

10. **Display ENAME, JOININGDATE and CITY whose city contains 'dne' as three letters in city name.**

    --> db.Employee.find({ CITY: /dne/ }, { ENAME: 1, JOININGDATE: 1, CITY: 1 })

11. **Display ENAME, JOININGDATE and CITY who does not belongs to city London or Sydney.**

    --> db.Employee.find({ CITY: { $nin: ["London", "Sydney"] } }, { ENAME: 1, JOININGDATE: 1, CITY: 1 })

12. **Find employees whose names start with 'J'.**

    --> db.Employee.find({ ENAME: /^J/ })

13. **Find employees whose names end with 'y'.**

    --> db.Employee.find({ ENAME: /y$/ })

14. **Find employees whose names contain the letter 'a'.**

    --> db.Employee.find({ ENAME: /a/ })

15. **Find employees whose names contain either 'a' or 'e'.**

    --> db.Employee.find({ ENAME: /[ae]/ })

16. **Find employees whose names start with 'J' and end with 'n'.**

    --> db.Employee.find({ ENAME: /^J.*n$/ })

17. **Find employees whose CITY starts with 'New'.**

--> db.Employee.find({ CITY: /^New/ })

18. **Find employees whose CITY does not start with 'L'**

    --> db.Employee.find({ CITY: { $not: /^L/ } })

19. **Find employees whose CITY contains the word 'York'.**

    --> db.Employee.find({ CITY: /York/ })

20. **Find employees whose names have two consecutive vowels (a, e, i, o, u).**

    --> db.Employee.find({ ENAME: /[aeiou]{2}/ })

21. **Find employees whose names have three or more letters.**

    --> db.Employee.find({ ENAME: /^.{3,}$/ })

22. **Find employees whose names have exactly 4 letters.**

    --> db.Employee.find({ ENAME: /^.{4}$/ })

23. **Find employees whose names start with either 'S' or 'M'.**

    --> db.Employee.find({ ENAME: /^[SM]/ })

24. **Find employees whose names contain 'il' anywhere.**

    --> db.Employee.find({ ENAME: /il/ })

25. **Find employees whose names do not contain 'a'.**

    --> db.Employee.find({ ENAME: { $not: /a/ } })

26. **Find employees whose names contain any digit.**

    --> db.Employee.find({ ENAME: /\d/ })

27. **Find employees whose names contain exactly one vowel.**

    --> db.Employee.find({ ENAME: /^[^aeiou]*[aeiou][^aeiou]*$/i })

28. **Find employees whose names start with any uppercase letter followed by any lowercase letter.**

    --> db.Employee.find({ ENAME: /^[A-Z][a-z]/ })

Part – B

29. **Display documents where sname start with K.**

    --> db.Student.find({ SNAME: /^K/ })

30. **Display documents where sname starts with Z or D.**

    --> db.Student.find({ SNAME: /^[ZD]/ })

31. **Display documents where city starts with A to R.**

    --> db.Student.find({ CITY: /^[A-R]/ })

32. **Display students' info whose name start with P and ends with i.**

    --> db.Student.find({ SNAME: /^P.*i$/ })

33. **Display students' info whose department name starts with 'C'.**

    --> db.Student.find({ DEPARTMENT: /^C/ })

34. **Display name, sem, fees, and department whose city contains ‘med’ as three letters somewhere in city name.**

    --> db.Student.find({ CITY: /med/ }, { SNAME: 1, SEM: 1, FEES: 1, DEPARTMENT: 1 })

35. **Display name, sem, fees, and department who does not belongs to city Rajkot or Baroda.**

    --> db.Student.find({ CITY: { $nin: ["Rajkot", "Baroda"] } }, { SNAME: 1, SEM: 1, FEES: 1, DEPARTMENT: 1 })

36. **Find students whose names start with 'K' and are followed by any character.**

    --> db.Student.find({ SNAME: /^K./ })

37. **Find students whose names end with 'a'.**

    --> db.Student.find({ SNAME: /a$/ })

38. **Find students whose names contain 'ri'. (case-insensitive)**

    --> db.Student.find({ SNAME: /ri/i })

Part – C

39. **Find students whose names start with a vowel (A, E, I, O, U).**

    --> db.Student.find({ SNAME: /^[AEIOU]/ })

40. **Find students whose CITY ends with 'pur' or 'bad'.**

    --> db.Student.find({ CITY: /(pur|bad)$/ })

41. **Find students whose FEES starts with '1'.**

    --> db.Student.find({ FEES: /^1/ })

42. **Find students whose SNAME starts with 'K' or 'V'.**

    --> db.Student.find({ SNAME: /^[KV]/ })

43. **Find students whose CITY contains exactly five characters.**

    --> db.Student.find({ CITY: /^.{5}$/ })

44. **Find students whose names do not contain the letter 'e'.**

    --> db.Student.find({ SNAME: { $not: /e/ } })

45. **Find students whose CITY starts with 'Ra' and ends with 'ot'.**

    --> db.Student.find({ CITY: /^Ra.*ot$/ })

46. **Find students whose names contain exactly one vowel.**

    --> db.Student.find({ SNAME: /^[^aeiou]*[aeiou][^aeiou]*$/i })

47. **Find students whose names start and end with the same letter.**

    --> db.Student.find({ SNAME: /^(.).*\1$/ })

48. **Find students whose DEPARTMENT starts with either 'C' or 'E'.**

    --> db.Student.find({ DEPARTMENT: /^[CE]/ })

49. **Find students whose SNAME has exactly 5 characters.**

    --> db.Student.find({ SNAME: /^.{5}$/ })

50. **Find students whose GENDER is Female and CITY starts with 'A'.**

    --> db.Student.find({ GENDER: "Female", CITY: /^A/ })

Lab-12 (Aggregate)

Part – A

1. **Display distinct city.**
   --> db.Student.aggregate([{ $group: { _id: "$CITY" } }, { $project: { _id: 0, city: "$_id" } }])

2. **Display city wise count of number of students.**
   --> db.Student.aggregate([{ $group: { _id: "$CITY", count: { $sum: 1 } } }])

3. **Display sum of salary in your collection.**
   --> db.Student.aggregate([{ $group: { _id: null, totalSalary: { $sum: "$SALARY" } } }])
4. **Display average of salary in your document.**
   --> db.Student.aggregate([{ $group: { _id: null, avgSalary: { $avg: "$SALARY" } } }])
5. **Display maximum and minimum salary of your document.**
   --> db.Student.aggregate([{ $group: { _id: null, maxSalary: { $max: "$SALARY" }, minSalary: { $min: "$SALARY" } } }])
6. **Display city wise total salary in your collection.**
   --> db.Student.aggregate([{ $group: { _id: "$CITY", totalSalary: { $sum: "$SALARY" } } }]
7. **Display gender wise maximum salary in your collection.**
   --> db.Student.aggregate([{ $group: { _id: "$GENDER", maxSalary: { $max: "$SALARY" } } }])
8. **Display city wise maximum and minimum salary.**
   --> db.Student.aggregate([{ $group: { _id: "$CITY", maxSalary: { $max: "$SALARY" }, minSalary: { $min: "$SALARY" } } }])
9. **Display count of persons lives in Sydney city in your collection.**
   --> db.Student.aggregate([{ $match: { CITY: "Sydney" } }, { $count: "count" }])

10. **Display average salary of New York city.**
    --> db.Student.aggregate([{ $match: { CITY: "New York" } }, { $group: { _id: "$CITY", avgSalary: { $avg: "$SALARY" } } }])
11. **Count the number of male and female students in each Department**
    --> db.Student.aggregate([{ $group: { _id: { Department: "$DEPARTMENT", Gender: "$GENDER" }, count: { $sum: 1 } } }])

12. **Find the total Fees collected from each Department.**
    --> db.Student.aggregate([{ $group: { _id: "$DEPARTMENT", totalFees: { $sum: "$FEES" } } }])

13. **Find the minimum Fees paid by male and female students in each City.**
    --> db.Student.aggregate([{ $group: { _id: { City: "$CITY", Gender: "$GENDER" }, minFees: { $min: "$FEES" } } }])

14. **Sort students by Fees in descending order and return the top 5.**
    --> db.Student.find().sort({ FEES: -1 }).limit(5)

15. **Group students by City and calculate the average Fees for each city, only including cities with more than 1 student.**
    --> db.Student.aggregate([{ $group: { _id: "$CITY", avgFees: { $avg: "$FEES" }, count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } }])

16. **Filter students from CE or Mechanical department, then calculate the total Fees**
    --> db.Student.aggregate([{ $match: { DEPARTMENT: { $in: ["CE", "Mechanical"] } } }, { $group: { _id: null, totalFees: { $sum: "$FEES" } } }])

17. **Count the number of male and female students in each Department.**
    --> db.Student.aggregate([{ $group: { _id: { Department: "$DEPARTMENT", Gender: "$GENDER" }, count: { $sum: 1 } } }])

18. **Filter students from Rajkot, then group by Department and find the average Fees for each department.**
    --> db.Student.aggregate([{ $match: { CITY: "Rajkot" } }, { $group: { _id: "$DEPARTMENT", avgFees: { $avg: "$FEES" } } }])
19. **Group by Sem and calculate both the total and average Fees, then sort by total fees in descending order.**
    --> db.Student.aggregate([{ $group: { _id: "$SEM", totalFees: { $sum: "$FEES" }, avgFees: { $avg: "$FEES" } } }, { $sort: { totalFees: -1 } }])

20. **Find the top 3 cities with the highest total Fees collected by summing up all students' fees in those cities.**
    --> db.Student.aggregate([{ $group: { _id: "$CITY", totalFees: { $sum: "$FEES" } } }, { $sort: { totalFees: -1 } }, { $limit: 3 }])

Part – B

1. **Create a collection named" Stock."**
   --> db. `createCollection("Stock")`

2. **Insert below 9 documents into the "Stock" collection.**

```
{   "_id": 1,
    "company": "Company-A",
    "sector": "Technology",
    "eps": 5.2,
    "pe": 15.3,
    "roe": 12.8,
    "sales": 300000,
    "profit": 25000
}
{   "_id": 2,
    "company": "Company-B",
    "sector": "Finance",
    "eps": 7.1,
    "pe": 12.4,
    "roe": 10.9,
    "sales": 500000,
    "profit": 55000
}
{   "_id": 3,
    "company": "Company-C",
    "sector": "Retail",
```

"eps": 3.8,
"pe": 22.1,
"roe": 9.5,
"sales": 200000,
"profit": 15000
}
{ "_id": 4,
"company": "Company-D",
"sector": "Technology",
"eps": 5.2,
"pe": 15.3,
"roe": 12.8,
"sales": 300000,
"profit": 25000
}
{ "_id": 5,
"company": "Company-E",
"sector": "Finance",
"eps": 7.1,
"pe": 12.4,
"roe": 10.9,
"sales": 450000,
"profit": 40000
}
{ "_id": 6,
"company": "Company-F",
"sector": "Healthcare",
"eps": 3.8,
"pe": 18.9,
"roe": 9.5,
"sales": 500000,
"profit": 35000
}
{ "_id": 7,
"company": "Company-G",
"sector": "Retail",
"eps": 4.3,
"pe": 22.1,
"roe": 14.2,
"sales": 600000,
"profit": 45000
}
{
"_id": 8,
"company": "Company-H",
"sector": "Energy",
"eps": 6.5,

"pe": 10.5,
"roe": 16.4,
"sales": 550000,
"profit": 50000
}
{
"_id": 9,
"company": "Company-I",
"sector": "Consumer Goods",
"eps": 2.9,
"pe": 25.3,
"roe": 7.8,
"sales": 350000,
"profit": 20000
}

--> db.Stock.insertMany([{ _id: 1, company: "Company-A", sector: "Technology", eps: 5.2, pe: 15.3, roe: 12.8, sales: 300000, profit: 25000 }, { _id: 2, company: "Company-B", sector: "Finance", eps: 7.1, pe: 12.4, roe: 10.9, sales: 500000, profit: 55000 }, { _id: 3, company: "Company-C", sector: "Retail", eps: 3.8, pe: 22.1, roe: 9.5, sales: 200000, profit: 15000 }, { _id: 4, company: "Company-D", sector: "Technology", eps: 5.2, pe: 15.3, roe: 12.8, sales: 300000, profit: 25000 }, { _id: 5, company: "Company-E", sector: "Finance", eps: 7.1, pe: 12.4, roe: 10.9, sales: 450000, profit: 40000 }, { _id: 6, company: "Company-F", sector: "Healthcare", eps: 3.8, pe: 18.9, roe: 9.5, sales: 500000, profit: 35000 }, { _id: 7, company: "Company-G", sector: "Retail", eps: 4.3, pe: 22.1, roe: 14.2, sales: 600000, profit: 45000 }, { _id: 8, company: "Company-H", sector: "Energy", eps: 6.5, pe: 10.5, roe: 16.4, sales: 550000, profit: 50000 }, { _id: 9, company: "Company-I", sector: "Consumer Goods", eps: 2.9, pe: 25.3, roe: 7.8, sales: 350000, profit: 20000 }])

3. **Calculate the total sales of all companies.**
   --> db.Stock.aggregate([{ $group: { _id: null, totalSales: { $sum: "$sales" } } }])
4. **Find the average profit for companies in each sector.**
   --> db.Stock.aggregate([{ $group: { _id: "$sector", avgProfit: { $avg: "$profit" } } }])
5. **Get the count of companies in each sector**
   --> db.Stock.aggregate([{ $group: { _id: "$sector", count: { $sum: 1 } } }])
6. **Find the company with the highest PE ratio.**
   --> db.Stock.aggregate([{ $sort: { pe: -1 } }, { $limit: 1 }])
7. **Filter companies with PE ratio greater than 20.(Use: Aggregate)**
   --> db.Stock.aggregate([{ $match: { pe: { $gt: 20 } } }])
8. **Calculate the total profit of companies with sales greater than 250,000.**
   --> db.Stock.aggregate([{ $match: { sales: { $gt: 250000 } } }, { $group: { _id: null, totalProfit: { $sum: "$profit" } } }])
9. **Project only the company name and profit fields.(Use: Aggregate)**
   --> db.Stock.aggregate([{ $project: { company: 1, profit: 1 } }])

10. **Find companies where EPS is greater than the average EPS.**
11. --> db.Stock.aggregate([
    { $group: { _id: null, avgEPS: { $avg: "$eps" } } },
    {
    $lookup: {
    from: "Stock",
    let: { avgEPS: "$avgEPS" },
    pipeline: [
    { $match: { $expr: { $gt: ["$eps", "$$avgEPS"] } } }
    ],
    as: "companiesWithHigherEPS"
    }
    },
    { $unwind: "$companiesWithHigherEPS" },
    { $replaceRoot: { newRoot: "$companiesWithHigherEPS" } }
    ])

12. **Group companies by sector and get the maximum sales in each sector.**
    --> db.Stock.aggregate([{ $group: { _id: "$sector", maxSales: { $max: "$sales" } } }])
13. **Calculate the total sales and total profit of companies in each sector.**
    --> db.Stock.aggregate([{ $group: { _id: "$sector", totalSales: { $sum: "$sales" }, totalProfit: { $sum: "$profit" } } }])

14. **Sort companies by profit in descending order.(Use: Aggregate)**
    --> db.Stock.aggregate([{ $sort: { profit: -1 } }])

15. **Find the average ROE across all companies.**
    --> db.Stock.aggregate([{ $group: { _id: null, avgROE: { $avg: "$roe" } } }])

16. **Group companies by sector and calculate both the minimum and maximum EPS.**
    --> db.Stock.aggregate([{ $group: { _id: "$sector", minEPS: { $min: "$eps" }, maxEPS: { $max: "$eps" } } }])

Part – C

1. **Count the number of companies with profit greater than 30,000.**
   --> db.Stock.aggregate([
   { $match: { profit: { $gt: 30000 } } },
   { $count: "companyCount" }
   ])
2. **Get the total profit by sector and sort by descending total profit.**
   --> db.Stock.aggregate([
   { $group: { _id: "$sector", totalProfit: { $sum: "$profit" } } },
   { $sort: { totalProfit: -1 } }
   ])
3. **Find the top 3 companies with the highest sales.**
   --> db.Stock.aggregate([
   { $sort: { sales: -1 } },
   { $limit: 3 }

```
])
```

4. **Calculate the average PE ratio of companies grouped by sector.**

```
--> db.Stock.aggregate([
  { $group: { _id: "$sector", averagePE: { $avg: "$pe" } } }
])
```

5. **Get the sum of sales and profit for each company.**

```
--> db.Stock.aggregate([
  { $project: { company: "$company", totalSalesAndProfit: { $add: ["$sales", "$profit"] } } }
])
```

6. **Find companies with sales less than 400,000 and sort them by sales.**

```
--> db.Stock.aggregate([
  { $match: { sales: { $lt: 400000 } } },
  { $sort: { sales: 1 } }
])
```

7. **Group companies by sector and find the total number of companies in each sector.**

```
--> db.Stock.aggregate([
  { $group: { _id: "$sector", totalCompanies: { $sum: 1 } } }
])
```

8. **Get the average ROE for companies with sales greater than 200,000.**

```
--> db.Stock.aggregate([
  { $match: { sales: { $gt: 200000 } } },
  { $group: { _id: null, averageROE: { $avg: "$roe" } } }
])
```

9. **Find the maximum profit in each sector.**

```
--> db.Stock.aggregate([
  { $group: { _id: "$sector", maxProfit: { $max: "$profit" } } }
])
```

10. **Get the total sales and count of companies in each sector.**

```
--> db.Stock.aggregate([
  { $group: { _id: "$sector", totalSales: { $sum: "$sales" }, companyCount: { $count: {} } } }
])
```

11. **Project fields where profit is more than 20,000 and only show company and profit.**

```
--> db.Stock.aggregate([
  { $match: { profit: { $gt: 20000 } } },
  { $project: { company: 1, profit: 1 } }
])
```

12. **Find companies with the lowest ROE and sort them in ascending order.(Use: Aggregate)**

```
--> db.Stock.aggregate([
  { $sort: { roe: 1 } }
])
```

Lab-13 (Index, Cursor, Schema Validation)

Part – A

1. **Create an index on the company field in the stocks collection.**
   --> db.Stock.createIndex({ company: 1 })
2. **Create a compound index on the sector and sales fields in the stocks collection.**
   --> db.Stock.createIndex({ sector: 1, sales: -1 })
3. **List all the indexes created on the stocks collection.**
   --> db.Stock.getIndexes()
4. **Drop an existing index on the company field from the stocks collection.**
   --> db.Stock.dropIndex("company_1")
5. **Use a cursor to retrieve and iterate over documents in the stocks collection, displaying each document.**
   --> const cursor = db.Stock.find();
   cursor.forEach(doc => printjson(doc));
6. **Limit the number of documents returned by a cursor to the first 3 documents in the stocks collection.**
   --> const cursor = db.Stock.find().limit(3);
   cursor.forEach(doc => printjson(doc));
7. **Sort the documents returned by a cursor in descending order based on the sales field.**
   --> const cursor = db.Stock.find().sort({ sales: -1 });
   cursor.forEach(doc => printjson(doc));
8. **Skip the first 2 documents in the result set and return the next documents using the cursor.**
   --> const cursor = db.Stock.find().skip(2);
   cursor.forEach(doc => printjson(doc));
9. **Convert the cursor to an array and return all documents from the stocks collection.**
   --> const allDocsArray = db.Stock.find().toArray();
   printjson(allDocsArray);
10. **Create a collection named "Companies" with schema validation to ensure that each document must contains a company field (string) and a sector field (string).**

```
--> db.createCollection("Companies", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["company", "sector"],
      properties: {
        company: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        sector: {
          bsonType: "string",
          description: "must be a string and is required"
        }
      }
    }
  }
});
```

Part – B

1. **Create a collection named "Scripts" with validation for fields like eps, pe, and roe to ensure that they are numbers and required/compulsory fields.**

```
--> db.createCollection("Scripts", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["eps", "pe", "roe"],
         properties: {
            eps: {
               bsonType: "number",
               description: "must be a number and is required"
            },
            pe: {
               bsonType: "number",
               description: "must be a number and is required"
            },
            roe: {
               bsonType: "number",
               description: "must be a number and is required"
            }
         }
      }
   }
});
```

2. **Create a collection named "Products" where each product has an embedded document for manufacturer details and a multivalued field for categories that stores an array of category names the product belongs to.**
   - **manufacturer details: The manufacturer will be an embedded document with fields like name, country, and establishedYear.**
   - **categories: The categories will be an array field that holds multiple values. (i.e. Electronics, Mobile, Smart Devices).**

```
--> db.createCollection("Products", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["manufacturer", "categories"],
         properties: {
            manufacturer: {
               bsonType: "object",
               required: ["name", "country", "establishedYear"],
               properties: {
                  name: {
                     bsonType: "string",
                     description: "must be a string and is required"
                  },
                  country: {
                     bsonType: "string",
                     description: "must be a string and is required"
```

```
            },
            establishedYear: {
              bsonType: "int",
              description: "must be an integer and is required"
            }
          }
        },
        categories: {
          bsonType: "array",
          items: {
            bsonType: "string",
            description: "must be a string"
          },
          description: "must be an array of strings and is required"
        }
      }
    }
  }
});
```

Part – C

1. **Create a collection named "financial_Reports" that requires revenue (a positive number) but allows optional fields like expenses and netIncome (if provided, they should also be numbers).**

```
--> db.createCollection("financial_Reports", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["revenue"],
      properties: {
        revenue: {
          bsonType: "double",
          description: "must be a positive number and is required"
        },
        expenses: {
          bsonType: "double",
          description: "must be a number if provided"
        },
        netIncome: {
          bsonType: "double",
          description: "must be a number if provided"
        }
      }
    }
  }
});
```

2. **Create a collection named "Student" where each student has name and address are embedded document and mobilenumber and emailaddress are multivalued field that stores an array of values**

```
--> db.createCollection("Student", {
```

```
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "address", "mobileNumber", "emailAddress"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        address: {
          bsonType: "object",
          properties: {
            street: { bsonType: "string" },
            city: { bsonType: "string" },
            state: { bsonType: "string" },
            zip: { bsonType: "string" }
          }
        },
        mobileNumber: {
          bsonType: "array",
          items: { bsonType: "string" },
          description: "must be an array of strings and is required"
        },
        emailAddress: {
          bsonType: "array",
          items: { bsonType: "string" },
          description: "must be an array of strings and is required"
        }
      }
    }
  }
});
```