LAB-1

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. Display 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()
```

LAB-2

Part - A

1. Retrieve/Display every document of Deposit collection.

→ db.Deposit.find()

2. Display only one document of Deposit collection.

→ db.Deposit.findOne()

3. Insert the following document into Deposit collection.

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

4. Insert the following documents into Deposit 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. 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 in ascending order by CNAME.

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

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

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

11. Display all the documents of Deposit collection in 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 the 3rd document of Deposit collection.

→ db.Deposit.find().skip(2).limit(1)

14. Display the 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 documents into “Student” collection.

→ 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 25, 30, or 35 (using $or and $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 25 and 35 and sort 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 by name in ascending order, then by age in descending order.

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

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

→ db.Student.find().limit(5)

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

→ db.Student.find().skip(3).limit(2)

5. Display the name of the 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, skipping the first 2 documents and returning the rest.

→ db.Student.find().skip(2)

LAB-3

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 students 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 set 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 students from 'San Francisco' 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.

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

4. Display first 2 students' names who live 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, 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'.

→ 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 male students in 'Rajkot'.

→ 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 the following 12 log entries into the logs collection.

```
→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.

```
→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-4

PART-A

1. Find employees whose name starts 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.
   --> db.Employee.find({ ENAME: /^[SM]/ })

4. Find employees where city starts with A to M.
   --> 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 (case-insensitive).
   --> db.Employee.find({ ENAME: /n/i })

7. Display employee info whose name starts with E and has 5 characters.
   --> db.Employee.find({ ENAME: /^E.{4}$/ })

8. Display employees whose name starts with S and ends in a.
   --> db.Employee.find({ ENAME: /^S.*a$/ })

9. Display EID, ENAME, CITY, and SALARY where name starts with ‘Phi’.
--> db.Employee.find({ ENAME: /^Phi/ }, { EID: 1, ENAME: 1, CITY: 1, SALARY: 1 })

10. Display ENAME, JOININGDATE, and CITY where city contains ‘dne’.
--> db.Employee.find({ CITY: /dne/ }, { ENAME: 1, JOININGDATE: 1, CITY: 1 })

11. Display ENAME, JOININGDATE, and CITY who do not belong 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.
--> 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'.
--> 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

1. Display documents where sname starts with K.
--> db.Student.find({ SNAME: /^K/ })

2. Display documents where sname starts with Z or D.
-->b.Student.find({ SNAME: /^[ZD]/ })

3. Display documents where city starts with A to R.
--> db.Student.find({ CITY: /^[A-R]/ })

4. Display students' info whose name starts with P and ends with i.
--> db.Student.find({ SNAME: /^P.*i$/ })

5. Display students' info whose department name starts with 'C'.
--> db.Student.find({ DEPARTMENT: /^C/ })

6. Display name, sem, fees, and department where city contains 'med'.
--> db.Student.find({ CITY: /med/ }, { SNAME: 1, SEM: 1, FEES: 1, DEPARTMENT: 1 })

7. Display name, sem, fees, and department who does not belong to Rajkot or Baroda.
--> db.Student.find({ CITY: { $nin: ["Rajkot", "Baroda"] } }, { SNAME: 1, SEM: 1, FEES: 1, DEPARTMENT: 1 })

8. Find students whose names start with 'K' and are followed by any character.
--> db.Student.find({ SNAME: /^K./ })

9. Find students whose names end with 'a'.
--> db.Student.find({ SNAME: /a$/ })

10. Find students whose names contain 'ri'. (case-insensitive)
--> db.Student.find({ SNAME: /ri/i })

PART-C

1. Find students whose names start with a vowel (A, E, I, O, U).
--> db.Student.find({ SNAME: /^[AEIOU]/ })

2. Find students whose CITY ends with 'pur' or 'bad'.

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

3. Find students whose FEES starts with '1'.

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

4. Find students whose SNAME starts with 'K' or 'V'.

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

5. Find students whose CITY contains exactly five characters.

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

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

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

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

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

8. Find students whose names contain exactly one vowel.

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

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

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

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

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

11. Find students whose SNAME has exactly 5 characters.

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

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

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

LAB-5

PART-A

11. Display distinct city.

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

12. Display city wise count of number of students.

```
--> db.Student.aggregate([{ $group: { _id: "$CITY", count: { $sum: 1 } } }])
```

13. Display sum of salary in your collection.

```
--> db.Student.aggregate([{ $group: { _id: null, totalSalary: { $sum: "$SALARY" } } }])
```

14. Display average of salary in your document.

```
--> db.Student.aggregate([{ $group: { _id: null, avgSalary: { $avg: "$SALARY" } } }])
```

15. Display maximum and minimum salary of your document.

--> db.Student.aggregate([{ $group: { _id: null, maxSalary: { $max: "$SALARY" }, minSalary: { $min: "$SALARY" } } }])

16. Display city wise total salary in your collection.
--> db.Student.aggregate([{ $group: { _id: "$CITY", totalSalary: { $sum: "$SALARY" } } }])

17. Display gender wise maximum salary in your collection.
--> db.Student.aggregate([{ $group: { _id: "$GENDER", maxSalary: { $max: "$SALARY" } } }])

18. Display city wise maximum and minimum salary.
--> db.Student.aggregate([{ $group: { _id: "$CITY", maxSalary: { $max: "$SALARY" }, minSalary: { $min: "$SALARY" } } }])

19. Display count of persons lives in Sydney city in your collection.
--> db.Student.aggregate([{ $match: { CITY: "Sydney" } }, { $count: "count" }])

20. Display average salary of New York city.
--> db.Student.aggregate([{ $match: { CITY: "New York" } }, { $group: { _id: "$CITY", avgSalary: { $avg: "$SALARY" } } }])

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

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

23. 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" } } }])

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

25. 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 } } }])

26. 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" } } }])

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

28. 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" } } }])

29. 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 } }])

30. 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
11. Create a collection named" Stock."
--> db. createCollection("Stock")

12. Insert below 9 documents into the "Stock" collection.
--> 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 }])

13. Calculate the total sales of all companies.
--> db.Stock.aggregate([{ $group: { _id: null, totalSales: { $sum: "$sales" } } }])

14. Find the average profit for companies in each sector.
--> db.Stock.aggregate([{ $group: { _id: "$sector", avgProfit: { $avg: "$profit" } } }])

15. Get the count of companies in each sector.
--> db.Stock.aggregate([{ $group: { _id: "$sector", count: { $sum: 1 } } }])

16. Find the company with the highest PE ratio.
--> db.Stock.aggregate([{ $sort: { pe: -1 } }, { $limit: 1 }])

17. Filter companies with PE ratio greater than 20.(Use: Aggregate)
--> db.Stock.aggregate([{ $match: { pe: { $gt: 20 } } }])

18. 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" } } }])

19. Project only the company name and profit fields.(Use: Aggregate)
--> db.Stock.aggregate([{ $project: { company: 1, profit: 1 } }])

20. Find companies where EPS is greater than the average EPS.

```
--> 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" } }
])
```

21. Group companies by sector and get the maximum sales in each sector.

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

22. Calculate the total sales and total profit of companies in each sector.

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

23. Sort companies by profit in descending order.(Use: Aggregate)

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

24. Find the average ROE across all companies.

```
--> db.Stock.aggregate([{ $group: { _id: null, avgROE: { $avg: "$roe" } } }])
```

25. 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

11. Count the number of companies with profit greater than 30,000.

```
--> db.Stock.aggregate([
   { $match: { profit: { $gt: 30000 } } },
   { $count: "companyCount" }
])
```

12. Get the total profit by sector and sort by descending total profit.

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

13. Find the top 3 companies with the highest sales.

```
--> db.Stock.aggregate([
   { $sort: { sales: -1 } },
   { $limit: 3 }
])
```

14. Calculate the average PE ratio of companies grouped by sector.

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

15. Get the sum of sales and profit for each company.

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

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

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

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

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

18. 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" } } }
])
```

19. Find the maximum profit in each sector.

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

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

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

21. 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 } }
])
```

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

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

LAB-6

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"
                }
            }
        }
    }
});
```