What is MongoDB

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++

Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.

History of MongoDB

The initial development of MongoDB began in 2007 when the company was building a platform as a service similar to window azure.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc.

It was initially developed as a PAAS (Platform as a Service)

Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

MongoDB2.4.9 was the latest and stable version which was released on January 10, 2014.

What is NoSQL Database?

- The acronym **NoSQL** stands for **"**Not Only SQ**L"**.
- **NoSQL** Database is a category of database management systems that does not compliant with the traditional relational DBMS (RDBMS) rules, and does not uses the traditional SQL to query database.
- NoSQL Databases are used to store large volume of unstructured, schema-less non-relational data.

Difference between SQL and NoSQL

SQL	NoSQL
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have a dynamic schema

SQL	NoSQL
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows CAP(consistency, availability, partition tolerance)
Examples: MySQL, Oracle	Examples: MongoDB

Features of MongoDB

1. Support ad hoc queries

In MongoDB, you can search by field, range query and it also supports regular expression searches.

2. Indexing

You can index any field in a document.

3. Replication

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

4. Duplication of data

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

5. Load balancing

It has an automatic load balancing configuration because of data placed in shards.

6. Supports map reduce and aggregation tools.

7. Uses JavaScript instead of Procedures.

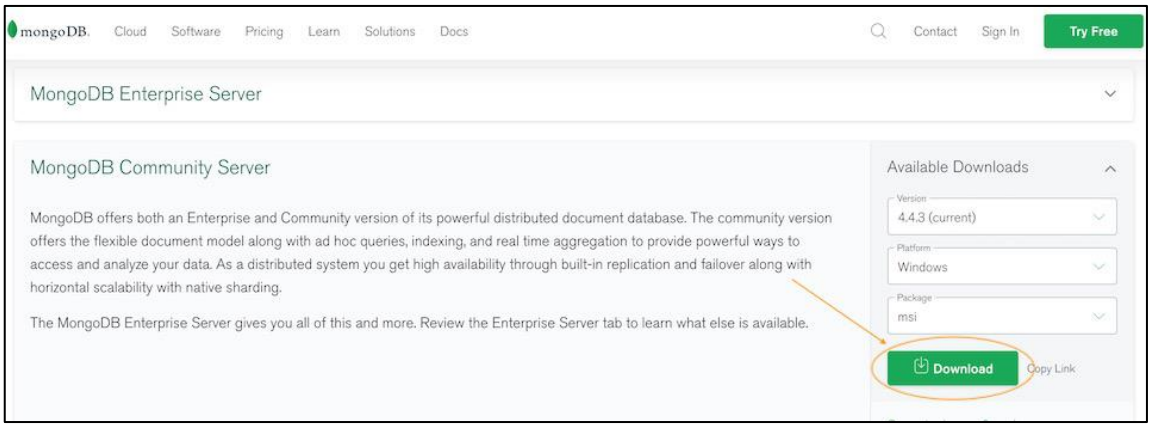
8. It is a schema-less database written in C++.

9. Provides high performance.

10. Stores files of any size easily without complicating your stack.

Installation of MongoDB

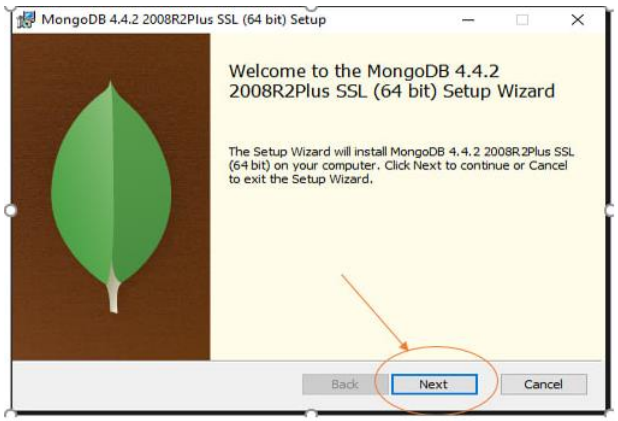
Step 1: Go to MongoDB Download Center to download MongoDB Community Server.



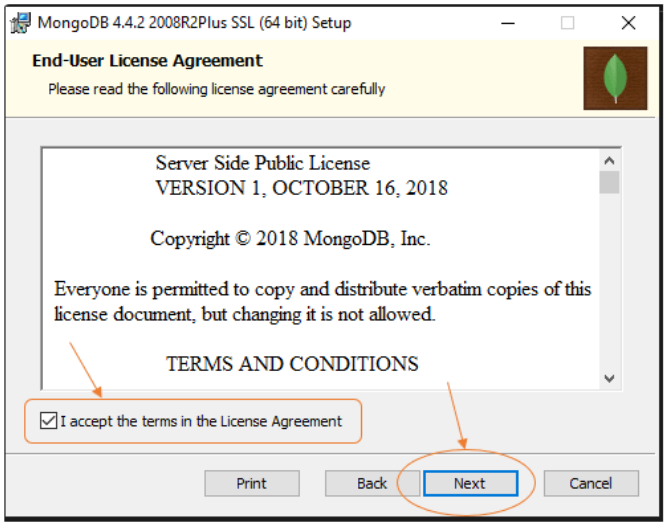
Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

- Version: 4.2.2
- OS: WindowsOS
- Package: msi

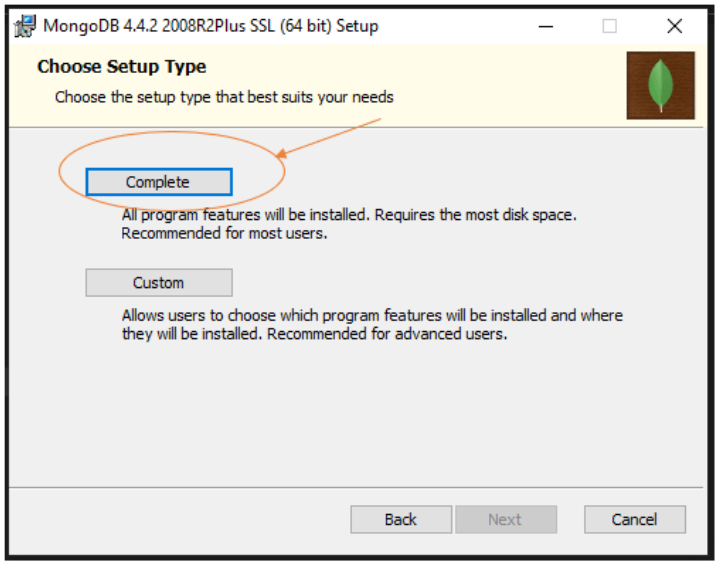
Step 2: When the download is complete open the msi file and click the next button in the startup screen:



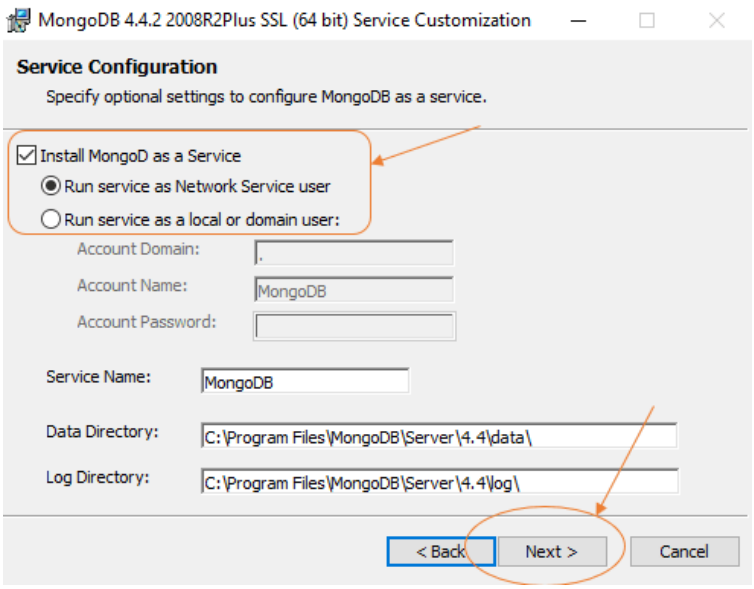
Step 3: Now accept the End-User License Agreement and click the next button:



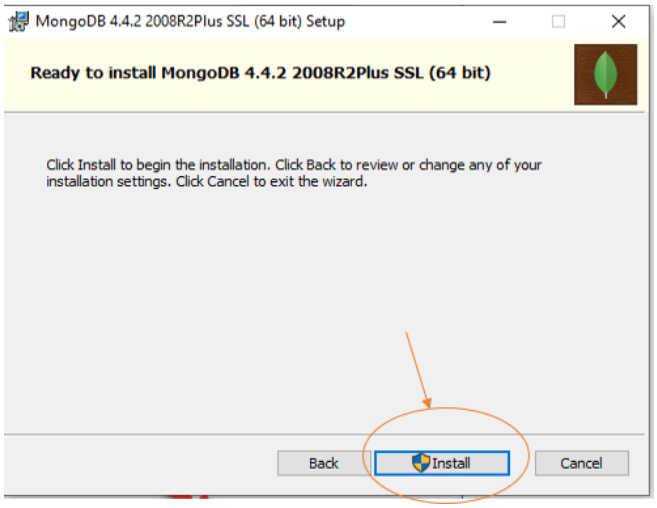
Step 4: Now select the complete option to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the Custom option:



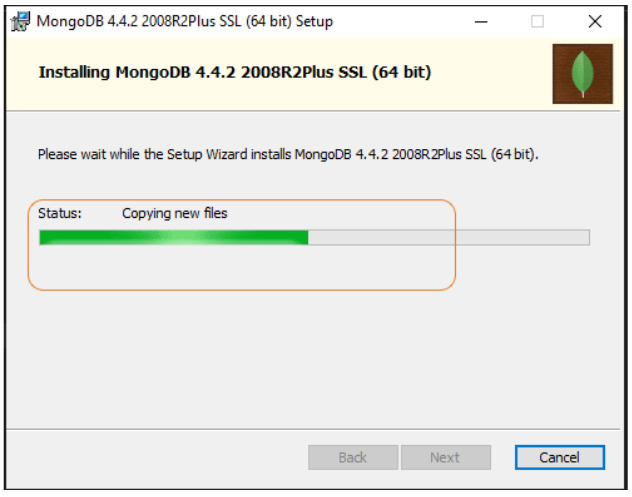
Step 5: Select "Run service as Network Service user" and copy the path of the data directory. Click Next:



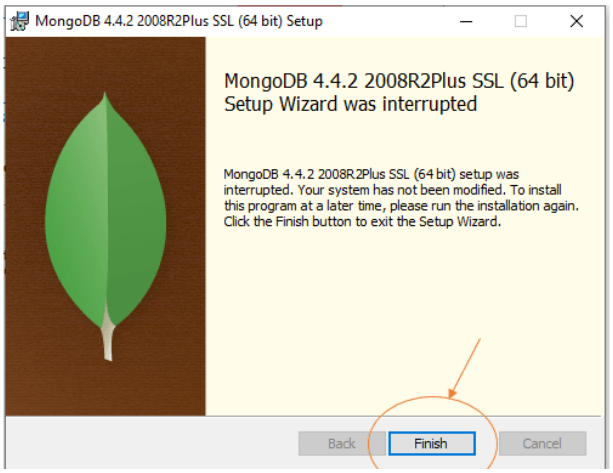
Step 6: Click the Install button to start the installation process:



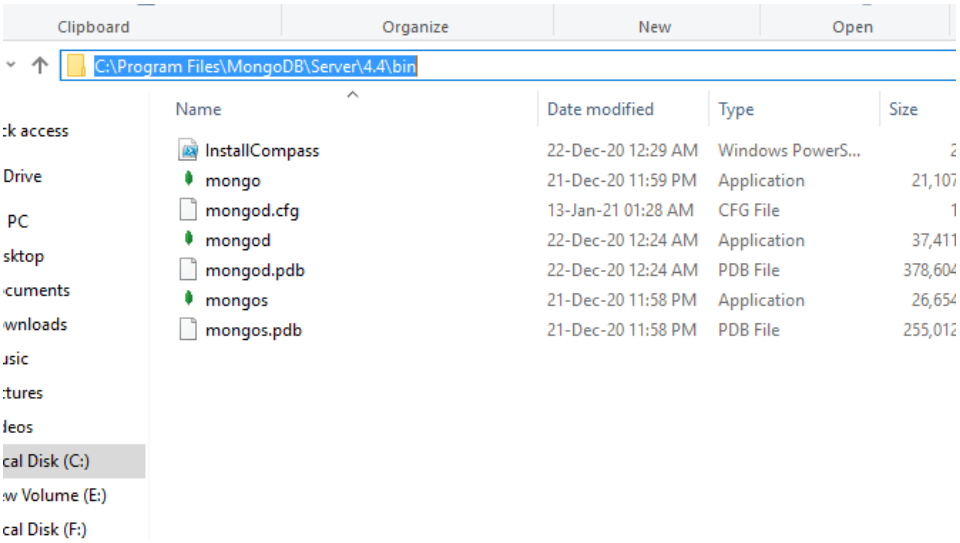
Step 7: After clicking on the install button installation of MongoDB begins:



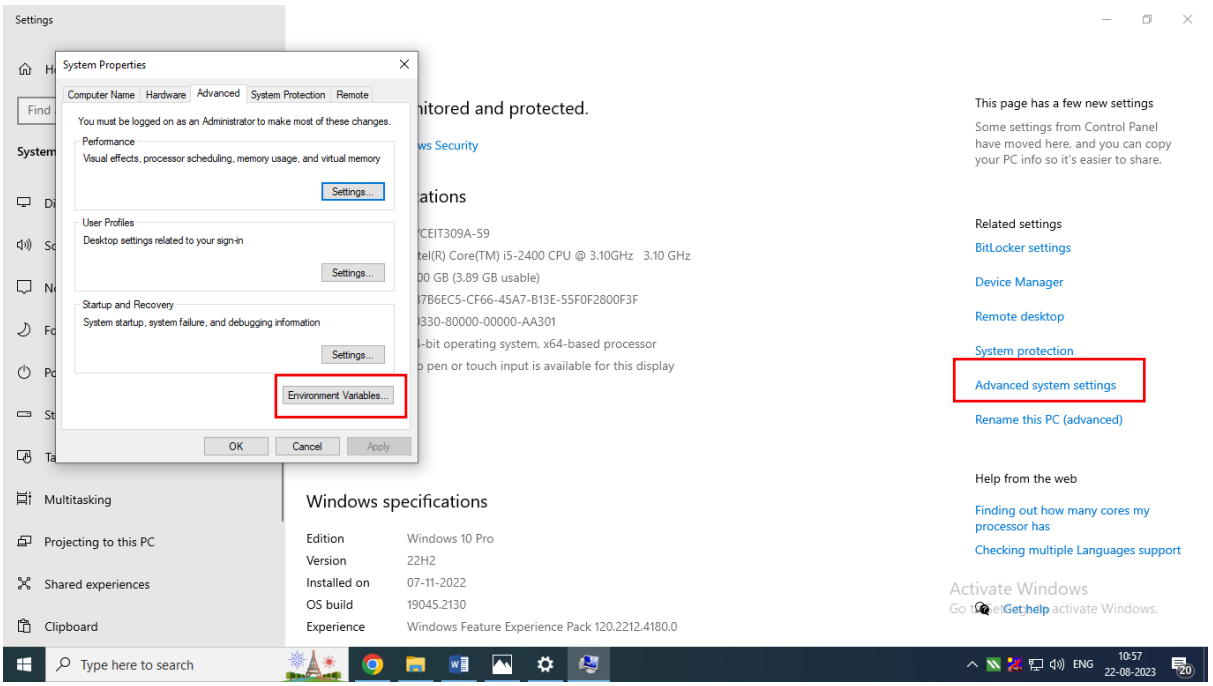
Step 8: Now click the Finish button to complete the installation process:

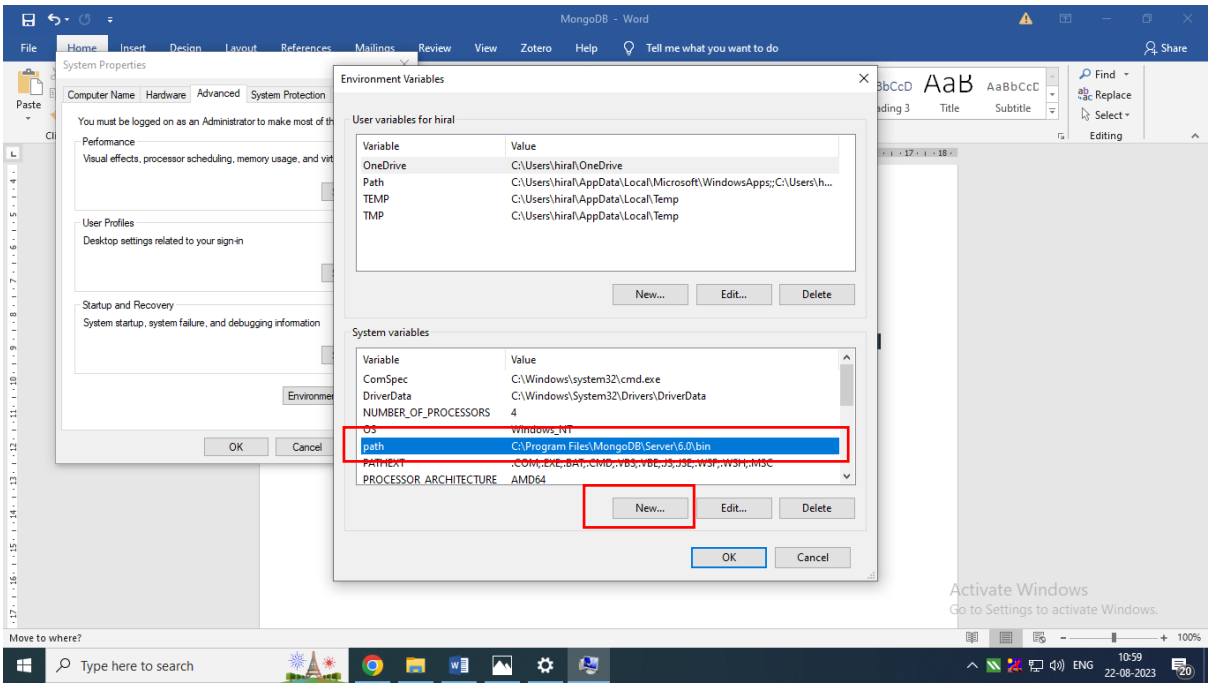


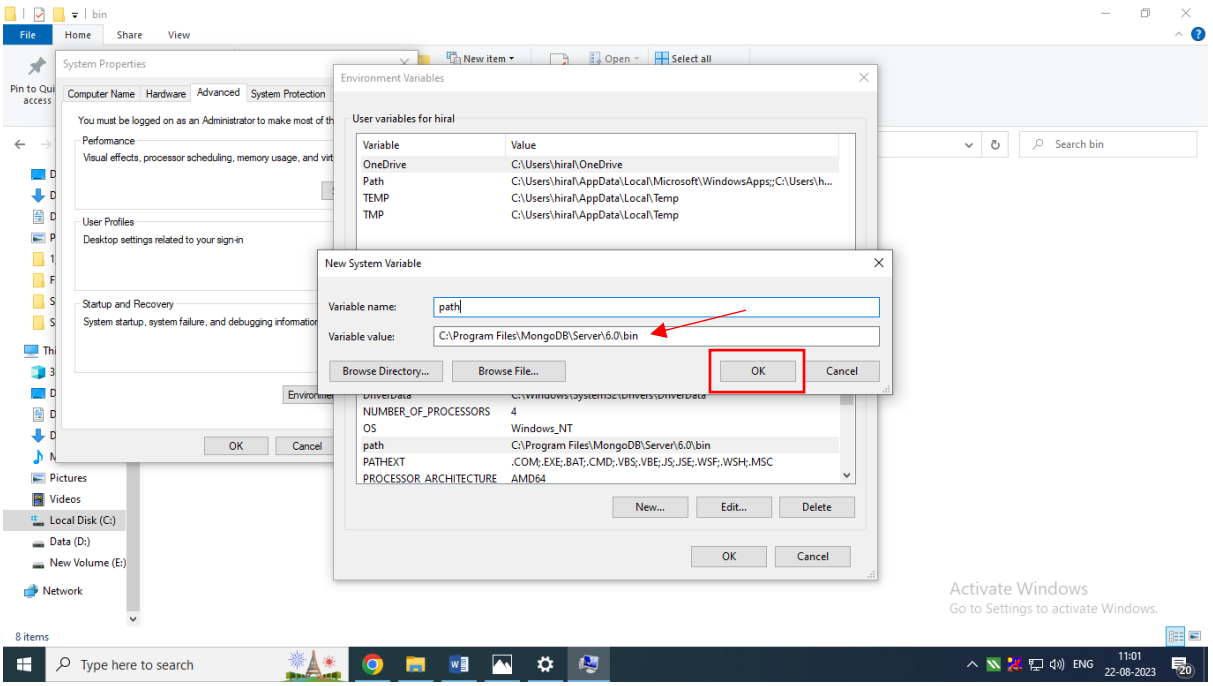
Step 9: Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:



Step 10: Now, to create an environment variable open system properties << Environment Variable << System variable << path << Edit Environment variable and copy to your environment system and click Ok:


Settings
System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your sign-in
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables...
OK
Cancel
Apply
Multitasking
Projecting to this PC
Shared experiences
Clipboard
Windows specifications
Edition Windows 10 Pro
Version 22H2
Installed on 07-11-2022
OS build 19045.2130
Experience Windows Feature Experience Pack 120.2212.4180.0
This page has a few new settings
Some settings from Control Panel have moved here, and you can copy your PC info so it's easier to share.
Related settings
BitLocker settings
Device Manager
Remote desktop
System protection
Advanced system settings
Rename this PC (advanced)
Help from the web
Finding out how many cores my processor has
Checking multiple Languages support
Activate Windows
Get help
Type here to search
ENG
10:57
22-08-2023


MongoDB - Word
System Properties
Environment Variables
User variables for hiral
Variable Value
OneDrive C:\Users\hiral\OneDrive
Path C:\Users\hiral\AppData\Local\Microsoft\WindowsApps;;C:\Users\h...
TEMP C:\Users\hiral\AppData\Local\Temp
TMP C:\Users\hiral\AppData\Local\Temp
New...
Edit...
Delete
System variables
Variable Value
ComSpec C:\Windows\system32\cmd.exe
DriverData C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS 4
OS Windows_NT
path C:\Program Files\MongoDB\Server\6.0\bin
PROCESSOR_ARCHITECTURE AMD64
OK
Cancel
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
10:59
22-08-2023



Step 11: for Mongo Shell download

✓ Go to https://www.mongodb.com/try/download/shell

✓ Download (.Zip file)

✓ Extract files and from bin folder Copy exe and dll files to "C:\Program Files\MongoDB\Server\6.0\bin"

To verify that it has been installed properly, open your terminal and type:

```
mongosh
```

MongoDB – Database, Collection, and Document

Databases, collections, documents are important parts of MongoDB without them you are not able to store data on the MongoDB server. A Database contains a collection, and a collection contains documents and the documents contain data

Database:

To see how many databases are present in your MongoDB server, write the following statement in the mongo shell:

```
>show dbs
```

Creating Database:

In the mongo shell, you can create a database with the help of the following command:

```
>use database_name
```

To **check the currently selected database**, use the command db:

```
>db
```

The dropDatabase command is used to drop a database. It also deletes the associated data files. It operates on the current database.

```
>db.dropDatabase()
```

Collection

Collections are just like tables in relational databases, they also store data, but in the form of documents. A single database is allowed to store multiple collections

Create collection

```
>db.createCollection(name)
```

is used to create collection. And also MongoDB creates collection automatically when you insert some documents

To **check the created collection**, use the command "show collections".

```
>show collections
```

Drop collection

```
>db.collection_name.drop()
```

method is used to drop a collection from a database. It completely removes a collection from the database

Document

Documents are just like rows in relational databases In MongoDB, the data records are stored as BSON documents. Here, BSON stands for binary representation of JSON documents, although BSON contains more data types as compared to JSON. The

document is created using field-value pairs or key-value pairs and the value of the field can be of any BSON type.

Syntax:

```
{
field1: value1
field2: value2
....
fieldN: valueN
}
```

Naming restriction of fields:

- The field names are of strings.
- The _id field name is reserved to use as a primary key. And the value of this field must be unique, immutable, and can be of any type other than an array.
- The field name cannot contain null characters.
- The top-level field names should not start with a dollar sign ($)

Insert Documents

There are 2 methods to insert documents into a MongoDB database.

insertOne()

To insert a single document, use the `insertOne()` method.

This method inserts a single object into the database.

Syntax

```
>db.collectionname.insertOne({fieldname:value})
```

Example

```
db.posts.insertOne({
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1
  })
```

insertMany()

To insert multiple documents at once, use the `insertMany()` method.

This method inserts an array of objects into the database.

Example

```
db.posts.insertMany([
  {
    title: "Post Title 2",
    body: "Body of post.",
    category: "Event",
    likes: 2
     },
  {
    title: "Post Title 3",
    body: "Body of post.",
    category: "Technology",
    likes: 3
     },
  {
    title: "Post Title 4",
    body: "Body of post.",
    category: "Event",
    likes: 4  }
])
```

Find Data

There are 2 methods to find and select data from a MongoDB collection, `find()` and `findOne()`.

find()

To select data from a collection in MongoDB, we can use the `find()` method.

This method accepts a query object. If left empty, all documents will be returned.

Example

```
db.posts.find()
```

findOne()

To select only one document, we can use the `findOne()` method.

This method accepts a query object. If left empty, it will return the first document it finds.

This method only returns the first match it finds.

Example

```
db.posts.findOne()
```

Querying Data

To query, or filter, data we can include a query in our `find()` or `findOne()` methods.

Example

```
db.posts.find( {category: "News"} )
```

Projection

Both find methods accept a second parameter called `projection`.

This parameter is an `object` that describes which fields to include in the results.

This parameter is optional. If omitted, all fields will be included in the results.

Example

This example will only display the `title` and `date` fields in the results.

```
db.posts.find({}, {title: 1, date: 1})
```

Notice that the `_id` field is also included. This field is always included unless specifically excluded.

We use a `1` to include a field and `0` to exclude a field.

Example

let's exclude the _id field.

```
db.posts.find({}, {_id: 0, title: 1, date: 1})
```

You cannot use both 0 and 1 in the same object. The only exception is the `_id` field. You should either specify the fields you would like to include or the fields you would like to exclude.

Let's exclude the date category field. All other fields will be included in the results.

Example

```
db.posts.find({}, {category: 0})
```

We will get an error if we try to specify both 0 and 1 in the same object.

Example

```
db.posts.find({}, {title: 1, date: 0})
```

Update Document

To update an existing document we can use the `updateOne()` or `updateMany()` methods.

The first parameter is a query object to define which document or documents should be updated.

The second parameter is an object defining the updated data.

updateOne()

The `updateOne()` method will update the first document that is found matching the provided query.

Example

```
>db.student.updateOne({'title':'MongoDB Overview'},

                        {$set:{'title':'New MongoDB Tutorial'}})
```

Insert if document not found

If you would like to update the document if it is not found, you can use the `upsert` option.

Example

Update the document, but if not found insert it:

```
db.posts.updateOne(
  { title: "Post Title 5" },
  {
    $set:
      {
        title: "Post Title 5",
        body: "Body of post.",
        category: "Event",
        likes: 5
        }
  },
  { upsert: true }
)
```

updateMany()

The `updateMany()` method will update all documents that match the provided query.

Example

```
db.student.updateMany({name: "abc"}, { $set: { age: 20 } })
```

Delete Documents

We can delete documents by using the methods `deleteOne()` or `deleteMany()`.

These methods accept a query object. The matching documents will be deleted.

deleteOne()

The `deleteOne()` method will delete the first document that matches the query provided.

Example

```
db.posts.deleteOne({ title: "Post Title 5" })
```

deleteMany()

The `deleteMany()` method will delete all documents that match the query provided.

Example

```
db.posts.deleteMany({ category: "Technology" })
```

rename Collection()

Call the db.collection.renameCollection() method on a collection object, to rename a collection. Specify the new name of the collection as an argument.

For example:

```
db.student.renameCollection("students")
```

This renames "student" collection to "students".

Limit() Method

To limit the records in MongoDB, you need to use limit() method. method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

Example

display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
```

Skip() Method

Apart from limit() method, there is one more method skip() which also accepts number type argument and is used to skip the number of documents.

syntax

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

Example

query to retrieve only one document and skip 2 documents.

```
>db.mycol.find().limit(1).skip(2)
```

sort() Method

To sort documents in MongoDB, you need to use sort() method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

will display the documents sorted by title in the descending order.

```
>db.mycol.find().sort({"title":-1})
```

Count() Method

This method returns the count of documents that would match a find() query. The db.collection.count() method does not perform the find() operation but instead counts and returns the number of results that match a query.

The db.collection.count() method has the following parameter:

The db.collection.count() method is equivalent to the db.collection.find(query).count() construct.

```
db.student.find({name:"ABC"}).count()
```

MongoDB Operators

Comparison Operators

MongoDB comparison operators can be used to compare values in a document.

Name	Description
$eq	we use this operator to match the values of the fields that are **equal(==)** to a specified value.
$ne	We use the **$ne** operator to match all values of the field that are **not equal (!=)** to a specified value.

$gt	We use the **$gt** operator to match values of the fields that are **greater than (>)** a specified value.
$gte	We use the **$gte** operator to match values of the fields that are **greater than equal to (>=)** the specified value.
$lt	We use the **$lt** operator to match values of the fields that are **lesser than (<)** the specified value.
$lte	We use the **$lte** operator to match values of the fields that are **lesser than equal to (<=)** the specified value.
$in	We use the **$in** operator to match values that exist in the array.
$nin	We use the **$nin** operator to match values that don't exist in the array.

- **$eq:**
  The $eq defines the equality condition (not mandatory). This operator is used to match field values and can make the query more readable and clear to the user.

  Syntax:

```
{ <field> : { $eq: <value> } }
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price of 500.

```
db.books.find({ price: { $eq: 500 } });
```

- **$gt:**
  To match field values that are greater than (>) a given number, we use the $gt operator.

  Syntax:

```
{   field: {$gt: value;}}
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price greater than 500.

```
db.books.find({ price: { $gt: 500 } });
```

- **$gte:**
  The $gte selects documents whose field value is larger than or equal to a given value.

  Syntax:

```
{ field: {$gte: value;}}
```

Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price greater than or equal to 500.

```
db.books.find({ price: { $gte: 500 } });
```

- **$in:**
  The $in operator selects documents where a field's value matches any value in a given array of values.

  Syntax:

```
{ field: { $in: [ <value1>, <value2>, ……] } }
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price in the range of 200 to 500.

```
db.books.find({ price: { $in: [200, 500] } });
```

- **$lt:**
  The $lt operator selects documents whose field value is less than the given value.

  Syntax:

```
{field: { $lt: value;}}
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price less than 500.

```
db.books.find({ price: { $lt: 500 } });
```

- **$lte:**
  The $lt operator selects documents whose field value is less than or equal to the given value.

  Syntax:

```
{ field: {$lte: value;}}
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price less than or equal to 500.

```
db.books.find({ price: { $lte: 500 } });
```

- **$ne:**
  The $ne operator selects documents whose field value does not match the specified value.

  Syntax:

```
{ <field>: { $ne: <value> } }
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price not equal to 500.

```
db.books.find({ price: { $ne: 500 } });
```

- **$nin:**
  The $nin operator selects documents with field values that are not in the given array or do not exist.

  Syntax:

```
{ field : { $nin: [ <value1>, <value2>, .... ] } }
```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that has a price not equal to in the given array.

```
db.books.find({ price: { $nin: [510, 230, 500] } });
```

Logical Query Operators

We use MongoDB Logical Query operators to filter documents from the database based upon the logic between variables or values. There are four types of Logical Operators provided by MongoDB.

Name	Description
$and	We use this operator to join query clauses with a logical **AND** and return the documents matching the given conditions of both clauses.
$or	We use the **$or** operator to join query clauses with a logical **OR** and return the documents matching the given conditions of either clause.
$nor	We use the **$nor** operator to join query clauses with a logical **NOR** and return the documents which fail to match both the clauses.
$not	We use the **$not** operator to invert the effect of the query expressions and return documents that do not match the query expression.

- **$and:**
  On an array, it performs a logical AND action. The array must contain 1, 2, or more phrases, and it selects the ones that meet all of the phrases in the collection(Array).

  Syntax:

```
{ $and: [ { <exp1> }, { <exp2> }, ....]}
```

  Example:

  we are retrieving only those employee's documents whose branch is CSE and joiningYear is 2018.

```
db.employee.find({$and: [{branch: "CSE"}, {joiningYear: 2018}]})
```

- **$not:**
  It acts as a logical NOT on the given phrase, selecting the ones that are unrelated to it.

  Syntax:

```
{ field: { $not: { <operator-expression> } } }
```

Example:

Suppose we have a collection of books, and we need to find the filter for all the books document that does not have a price greater than 500.

```
db.books.find({ price: { $not: { $gt: 500 } } }) )
```

- **$nor:**
  It acts as a logical NOR on a collection of 1, 2, or more query phrases, and selects the one that fails all of the query phrases in the collection of phrases.

  Syntax:

  ```
  { $nor: [ { <expression1> } , { <expresion2> } , ..... ] }
  ```

  Example: retrieving only those employee's documents whose salary is not 3000 and whose branch is not ECE.

  ```
  db.employee.find({$nor: [{salary: 3000}, {branch: "ECE"}]})
  ```

- **$or:**
  It operates as a logical OR action on an ordering of 1, 2, or more phrases, selecting documents that satisfy any one or more of the phrases.

  Syntax:

  ```
  { $or: [ { <exp_1> }, { <exp_2> }, ... , { <exp_n> } ] }
  ```

  Example: Suppose we have a collection of books, and we need to find the filter for all the books document that does have a price equal to 500 or quantity less than 500

  ```
  db.books.find({ $or: [{ quantity: { $lt: 500 } }, { price: 500 }]
  ```

Field Update Operators

MongoDB provides different types of field update operators to update the values of the fields of the documents that matches the specified condition.

Name	Description
$currentDate	**This operator is used to set the value of a field to current date, either as a Date or a Timestamp.**
$inc	**This operator is used to increment the value of the field by the specified amount.**

Name	Description
$mul	**This operator is used to multiply the value of the field by the specified amount.**
$rename	**This operator is used to rename a field.**
$set	**Sets the value of a field in a document.**
$unset	**Removes the specified field from a document.**

- **$currentDate**

This operator is used to set the value of a field to the current date.

- ✓ The default type of $currentDate operator is a Date.
- ✓ You can use this operator in methods like update(), updateOne(), etc., according to your requirements.
- ✓ If the specified field is not found then this operator will add that field in the document.

Syntax:

```
{ $currentDate: { <field1>: <typeSpecification1>, ... } }
```

typeSpecification1 is a boolean true to set the value of a field to the current date as a Date

Example:we are updating the value of joiningDate field of an employee's document whose first name is Om.

```
db.Employee.updateOne({"name.first": "Om"},

                      {$currentDate: {joiningDate: true}})
```

- **$inc**

Syntax:

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

Examples:

- db.table1.updateMany({},{$inc:{age:10}});
  - Increase age field values by 10 for all documents
- db.table1.updateOne({},{$inc:{age:15}});

        - Increase age field values by 15 of 1st document
    - db.table1.updateOne({},{$inc:{age:-15}});
        - increase age field values by (-15) of 1st document

- **$mul**

  Multiply the value of a field by a number. To specify a `$mul` expression, use the following prototype:

Syntax:

```
{ $mul: { <field1>: <number1>, ... } }
```

The field to update must contain a numeric value.

Examples:

- db.table1.updateOne({},{$mul:{age:15}});
    - Multiply age field by 15 of 1st document
- db.table1.updateMany({name:'N1'},{$mul:{age:0.5}});
    - Multiply age field by 0.5 for all documents where name is "N1"
- db.table1.updateOne({},{$mul:{age:2}});
    - Multiply age field by 0.5 for all documents

- **$unset**

The $unset operator deletes a particular field. Consider the following.

syntax:

```
{ $unset: { <field1>: "", ... } }
```

Example:

db.people.updateOne({age:{$eq:21}},{$unset:{branch:"CSE",age: 21}})

- **$rename**

The $rename operator updates the name of a field and has the following form:

Syntax:

```
{$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }
```

Example:

db.people.updateMany({},{$rename:{'name':'uname'}})
db.people.updateMany({},{$rename:{'name':'Uname','branch':'Branch'}})

Task:

Create a collection named student having fields name,age,standard and percentage.

Insert five to ten random records in table. Write MongoDB query for following.

1)find name of all students having age>5

2)update the standard for all students by 1

3)arrange all documents in descending order of age

4)show the name of student who is the most oldest student among all documents

5)delete the document of the student if standard is 12.

Answers

1. db.student.find({age:{$gt:5}})
2. db.student.updateMany({},{$inc:{standard:1}})
3. db.student.find().sort({age:-1})
4. db.student.find().sort({age:-1}).limit(1)
5. db.student.deleteMany({standard:12})

What is Mongoose?

Mongoose is an ODM (Object Data Modeling) library for MongoDB.
Node.js developers choose to work with Mongoose to help with data modeling, schema enforcement, model validation, and general data manipulation and Mongoose makes these tasks effortless.

Why Mongoose?

By default, MongoDB has a flexible data model. This makes MongoDB databases very easy to alter and update in the future. Mongoose forces a semi-rigid schema from the beginning. With Mongoose, developers must define a Schema and Model.

Install Mongoose from the command line using npm:

```
npm install mongoose
```

What is a schema?

A schema defines the structure of your collection documents. A Mongoose schema maps directly to a MongoDB collection.

```
const mySchema=new mg.Schema(
  {
    name:{
```

```
        type:String,
        required:true
    },
    Surname:String,
    age:Number,
    active:Boolean,
    date:{
        type:Date,
        default:new Date()
    }
  }
);
```

With schemas, we define each field and its data type. Permitted types are: String,Number,Date,Buffer,Boolean,Mixed,ObjectId,Array,Decimal128,Map

What is a model?

Models take your schema and apply it to each document in its collection.

Models are responsible for all document interactions like creating, reading, updating, and deleting (CRUD).

An important note: the first argument passed to the model should be the singular form of your collection name.

Mongoose automatically changes this to the plural form, transforms it to lowercase, and uses that for the database collection name.

```
const person=new mg.model("person",mySchema)
```

it will automatically converted to "people" by mongoose

```
mg.pluralize(null) // to add collection as we have mentioned
```

Await and Async

If you are using async functions, you can use the await operator on a Promise to pause further execution until the Promise reaches either the Fulfilled or Rejected state and returns. Since the await operator waits for the resolution of the Promise, you can use it in place of Promise chaining to sequentially execute your logic.

Inserting data

Now that we have our first model and schema set up, we can start inserting data into our database.

Back in the file, let's insert a new data.

```
const createDoc=async()=>
   {
    try{
      const personData=new person(
         {
            name:"def",
            Surname:"test",
            age:31,
            active:true,
            email:"abc@gmail.com"
         }
      )
   const result=await personData.save();
}
}
createDoc()
```

we create a new object and then use the save() method to insert it into our MongoDB database.

Example:

create database connection and insert one document(record) in collection(table)

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
   {
      name:{
         type:String,
         required:true
      },
      Surname:String,
      age:Number,
      active:Boolean,
      date:{
         type:Date,
```

```
        default:new Date()
    }
  }
);
const person=new mg.model("person",mySchema)
const personData=new person({
          name:"abc",
          Surname:"xyz",
          age:30,
          active:true
})

personData.save()
```

Example: use promise return for record insert using **async** and **await**

```
const createDoc=async()=>
{
  try{
    const personData=new person({
      name:"test",
      Surname:"XYZ",
      age:3,
      active:true
    })
    const result=await personData.save(); //for single data record
    console.log(result);
  }
  catch(err)
  {
    console.log("problem");
  }
}
createDoc();
```

Example: create database connection and insert multiple document(record) in collection(table)

```
const mg=require("mongoose")
```

```
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
   {
      name:{
         type:String,
         required:true
      },
      Surname:String,
      age:Number,
      active:Boolean,
      date:{
         type:Date,
         default:new Date()
      }
   }
);
const person=new mg.model("person",mySchema)
const createDoc=async()=>
   {
      try{
      const personData=new person(
         {
            name:"abc",
            Surname:"test1",
            age:33,
            active:true
         }
      )
      const personData1=new person(
         {
            name:"hi",
            Surname:"hi1",
            age:30,
            active:true
         }
```

```
    )
      const personData2=new person(
        {
          name:"hello",
          Surname:"hello1",
          age:37,
          active:true
        }
      )
      const personData3=new person(
        {
          name:"pqr",
          Surname:"hello11",
          age:37,
          active:true
        }
      )

      const result= await person.insertMany    ([personData,personData1,personData2,
personData3])

    }
      catch(err)
      {
        console.log("problem");
      }
  }
createDoc();
```

Example: create database connection and insert multiple document(record) in collection(table) and **find some Documents(records)**

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
  {
```

```
    name:{
      type:String,
      required:true
    },
    Surname:String,
    age:Number,
    active:Boolean,
    date:{
      type:Date,
      default:new Date()
    }
  }
);
const person=new mg.model("person",mySchema)
const createDoc=async()=>
  {
    try{
    const personData=new person(
      {
        name:"abc",
        Surname:"test1",
        age:33,
        active:true
      }
    )
    const personData1=new person(
      {
        name:"hi",
        Surname:"hi1",
        age:30,
        active:true
      }
    )
      const personData2=new person(
        {
          name:"hello",
          Surname:"hello1",
          age:37,
```

```
            active:true
        }
    )
    const personData3=new person(
        {
            name:"pqr",
            Surname:"hello11",
            age:37,
            active:true
        }
    )

     const result= await person.insertMany   ([personData,personData1,personData2,
personData3])
    const result= await person.find({name:"abc"})
    const result1= await person.find({name:"abc"},{date:1,_id:0})
   const result2= await person.find({name:"hi"}).limit(1)
  const result3= await person.find({name:"hello1"}).select({name:1}).limit(1)
  const result4= await person.findOne({name:"abc"}).select({Surname:1})
  const result5= await person.find({age:{$gt:28}},{name:1,_id:0})
  const result6= await person.find({name:{$in:["abc","pqr"]}})
 const result7= await person.find({age:{$gt:0,$lt:28}})
  const result8=await person.deleteMany({Surname:"hello1"})

 const result9=await person.find({age:{$lte:28}}).sort({name:-1}).count()


    console.log(result)
    console.log(result1)
    console.log(result2)
    console.log(result3)
    console.log(result4)
    console.log(result5)
    console.log(result6)
    console.log(result7)
    console.log(result8)
    console.log(result9)
```

```
    }
        catch(err)
        {
           console.log(err);
        }
  }
createDoc();
```

Example: create database connection and insert multiple document(record) in collection(table) and **update Document(record) by ID.**

Using Update function

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
  {
     name:{
        type:String,
        required:true
     },
     Surname:String,
     age:Number,
     active:Boolean,
     date:{
        type:Date,
        default:new Date()
     }
  }
);
const person=new mg.model("person",mySchema)
 const updateDoc=async(i) =>
{
    const result=await person.updateOne({_id:i},
    {
```

```
            $set:{age:37}
      })
    console.log(result)
}
updateDoc("64a7cd53376fc04b3c2637bd");
```

Using findByIdAndUpdate function

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
   {
      name:{
         type:String,
         required:true
      },
      Surname:String,
      age:Number,
      active:Boolean,
      date:{
         type:Date,
         default:new Date()
      }
   }
);
const person=new mg.model("person",mySchema)
 const updateDoc=async(_id) =>
{
     const result=await person.findByIdAndUpdate({_id},
         {
            $set:{age:27}
          },
           {new:true}
      )
    console.log(result)
```

```
}
updateDoc("64a7cd53376fc04b3c2637bd");
```

Example: create database connection and insert multiple document(record) in collection(table) and **delete Document(record) by ID.**

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
 {
      name:{
          type:String,
         required:true
    },
      Surname:String,
       age:Number,
      active:Boolean,
      date:{
          type:Date,
          default:new Date()
       }
   }
 );
 const person=new mg.model("person",mySchema)
const deleteDoc=async(_id) =>
{
      const result=await person.deleteOne({_id})
                              OR
      const result=await person.findByIdAndDelete({_id})

   console.log(result)
}
deleteDoc("64a7cd53376fc04b3c2637bd");
```

Inbuilt Validation using Mongoose

While creating document,if there are attributes and we want to specify some constraints,then it's possible by adding respective members in schema creation.

If we insert duplicate elements in name field,then it may throw error,if specify name as **unique:true.** After insertion,if we have specified **lowercase:true** then it may show all newly inserted names in lowercase.

If we have set a list of allowed values in enum, then it requires exact match.

We can provide custom validation using **validate(value)** inbuild function.

Example:

```
const mg=require("mongoose")
const v=require("validator")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
  {
    name:{
      type:String,
      required:true,
      unique:true,
      lowercase:true,
      trim:true,
      minlength:[3,"Min length must be 3"],
      maxlength:[7,"max length must be 7"],
      enum:["abc","xyz","def"]
          },
    age:{
      type:Number,
      validate(val){
        if(val<=0)
        {
          let k=new Error("Age must be positive");
          throw k;
```

```
            }
        }
    },
    email:{
        type:String,
        validate(val)
        {
            if(!v.isEmail(val))
            {
                throw new Error("Enter valid email_id")
            }
        }
    }

  }
);
const person=new mg.model("person",mySchema)
const createDoc=async()=>
  {
    try{
    const personData=new person(
        {
            name:"def",
            age:31,
            email:"abc@gmail.com"
        }
    )
    const result=await personData.save();
    console.log(result)
    }
     catch(err)
      {
          console.log("error has occured");
      }
  }
createDoc();
```

Express and MongoDB connectivity

Program: Create a form having Username and password. On clicking submit button username and password should be store inside database using node,express and mongoDB.

Index.html

```
<html>
  <body>
    <h1>Html file</h1>
    <form action="/login" method="get">
    <input type="text" name="username" />
    <input type="password" name="password" />
    <input type="submit"/>
    </form>
  </body>
</html>
```

Login.js (express file)

```
const expr=require("express")
const app=expr()
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.log(err)})
const mgschema=new mg.Schema(
  {
    username:String,
    password :String,
  },
  {
     versionKey:false
  }
)
mg.pluralize(null)
const login=new mg.model("login",mgschema)
app.get("/",(req,res)=>{
  res.sendFile(__dirname+"/index.html")
})
```

```
app.get("/login",(req,res)=>{
   const createDoc =async ()=>{
      try{
         var personData= new login(
            {
               username:req.query.username,
               password:req.query.password,
            }

         )
         const result= await login.insertOne(personData)
      }
      catch(err){
         console.log(err)
      }
   }
   createDoc()
   res.write("record inserted")
   res.send()
      }

).listen(3000)
```

push() method

The *push*() *method* adds new items to the end of an array. The *push*() *method* changes the length of the array. The *push*() *method* returns the new length. It will display all results in array.

Mongoose Example using push method:

Create Collection “employees” with following data

[{_id: 1,name: "Eric",age: 30,position: "Full Stack Developer",salary: 60000},

{_id: 2,name: "Erica",age: 35,position: "Intern",salary: 8000},

{_id: 3,name: "Erical",age: 40,position: "UX/UI Designer",salary: 56000},

{_id: 4,name: "treric7",age: 37,position: "Team Leader",salary: 85000},

{_id: 5,name: "Eliza",age: 25,position: "Software Developer",salary: 45000},

{_id: 6,name: "Trian",age: 29,position: "Data Scientist",salary: 75000},

{_id: 7,name: "Elizan",age: 25,position: "Full Stack Developer",salary: 49000}]

1) Find All Documents:
2) Find Documents by Position “Full Stack Developer”:
3) Retrieve name of employees whose age is greater than or equal to 25 and less than or equal to 40.
4) Retrieve name of the employee with the highest salary.
5) Retrieve employees with a salary greater than 50000.
6) Retrieve employees' names and positions, excluding the "_id" field.
7) Count the number of employees who have salary greater than 50000
8) Retrieve employees who are either " **Software Developer**" or "**Full Stack Developer**" and are below 30 years.
9) Increase the salary of an employee who has salary less than 50000 by 10%.
10) Delete all employees who are older than 50.
11) Give a 5% salary raise to all "**Data Scientist**"
12) Find documents where name like “%an”
13) Find documents where name like “Eri--” (Case Insensitive)
14) Find documents where name like “%ric%”
15) Find documents where name contains only 4 or 5 letters.
16) Find documents where name must end with digit

```
const mg=require("mongoose")
const v=require("validator")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)

  const mySchema=new mg.Schema(
    {
      _id:Number,
      name:String,
      age:Number,
      position:String,
      salary:Number
    }
);
const emp=new mg.model("employees ",mySchema)
const createDoc=async()=>
  {
    try{
      const personData1=[{_id: 1,name: "Eric",age: 30,position: "Full Stack Developer",salary: 60000},
      {_id: 2,name: "Erica",age: 35,position: "Intern",salary: 8000},
      {_id: 3,name: "Erical",age: 40,position: "UX/UI Designer",salary: 56000},
```

```
        {_id: 4,name: "treric7",age: 37,position: "Team Leader",salary: 85000},
        {_id: 5,name: "Eliza",age: 25,position: "Software Developer",salary: 45000},
        {_id: 6,name: "Trian",age: 29,position: "Data Scientist",salary: 75000},
        {_id: 7,name: "Elizan",age: 25,position: "Full Stack Developer",salary: 49000}]
      const result=[]
      result.push(await emp.insertMany(personData1))
      result.push(await emp.find())
      result.push(await emp.find({position:"Full Stack Developer"}))
      result.push(await emp.find({ age: { $gte: 30, $lte: 40 } },{name:1,_id:0}))
      result.push(await emp.find().sort({ salary: -1 }).limit(1))
      result.push(await emp.find({ salary: { $gt: 50000 } }))
      result.push(await emp.find({salary:{$gt:50000}}).count())
      result.push(await emp.find({ $and: [{ $or: [{ position: "Software Developer" }, { position: "Full
Stack Developer" }] }, { age: { $lt: 30 } }] }))
      result.push(await emp.updateOne({salary:{$lt:50000}},{ $mul: { salary: 1.1 } }))
      result.push(await emp.deleteMany({ age: { $gt: 50 } }))
      result.push(await emp.updateMany({ position: "Data Scientist" }, { $mul: { salary: 1.05 } }))
      result.push(await emp.find({name:{$regex:/an$/}}))
      result.push(await emp.find({name:{$regex:/^eri[A-z]{2}$/i}}))

      console.log('Query Results:', result);

    }
    catch(err)
    {
       console.log(err);
    }
    }
createDoc()
```

React and MongoDB connectivity(MERN stack)

In MERN stack , Node and Express serve backend functionality ,React serve frontend functionality and MongoDB serve database.

Folder structure:

```
React_connection  (Main folder)
--Backend (folder for backend)  )  ( install packages   express,cors,mongoose,body-parser,bcrypt)
  ---Server.js  (backend file)
--Frontend  (folder for frontend)  (create react app in this folder)  (install axios)
  ---myapp
     public
     src
      ---Signup.js  (frontend file)
```

---Login.js **(frontend file)**

Signup.js

```
import React, { useState } from 'react';
import axios from 'axios';

function Signup() {
  const [username, setUsername] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSignup = async (e) => {
    e.preventDefault();

    try {
      await axios.post('http://localhost:5000/api/signup', {
        username,
        email,
        password
      });
      alert('User signed up successfully.');
      setUsername('');
      setEmail('');
      setPassword('');
    }
   catch (error) {
      console.error('Error signing up:', error);
      alert('An error occurred.');
    }
  };

  return (
    <div>
      <h1>Sign Up</h1>
      <form onSubmit={handleSignup}>
        <input
          type="text"
          placeholder="Username"
          value={username}
          onChange={(e) => setUsername(e.target.value)}
        />
        <input
          type="email"
          placeholder="Email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
```

```
        <input
          type="password"
          placeholder="Password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        />
        <button type="submit">Sign Up</button>
      </form>
    </div>
  );
}

export default Signup;
```

Login.js

```
import React, { useState } from 'react';
import axios from 'axios';

function Login() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = async (e) => {
    e.preventDefault();

    try {
      await axios.post('http://localhost:5000/api/login', {
        username,
        password,
      });
      alert('Login successful.');
      setUsername('');
      setPassword('');
    } catch (error) {
      console.error('Error logging in:', error);
      alert('Login failed.');
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <form onSubmit={handleLogin}>
        <input
          type="text"
          placeholder="Username"
          value={username}
          onChange={(e) => setUsername(e.target.value)}
```

```
        />
        <input
          type="password"
          placeholder="Password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        />
        <button type="submit">Login</button>
      </form>
    </div>
  );
}

export default Login;
```

Server.js

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const bcrypt = require('bcrypt');
const cors=require('cors')
const app = express();

app.use(cors());
app.use(bodyParser.json());

mongoose.connect('mongodb://127.0.0.1:27017/db1');

const UserSchema = new mongoose.Schema({
  username: String,
  email: String,
  password: String
});

const User = mongoose.model('User', UserSchema);

app.post('/api/signup', async(req, res) => {
  try {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();
    res.status(201).json({ message: 'User signed up successfully.' });
  } catch (error) {
    res.status(500).json({ error: 'An error occurred.' });
  }
});

app.post('/api/login', async (req, res) => {
```

```
  try {
    const { username, password } = req.body;
    const user = await User.find({ username });

    if (!user) {
      return res.status(401).json({ error: 'User not found.' });
    }

    const passwordMatch = await bcrypt.compare(password, user.password);

    if (!passwordMatch) {
      return res.status(401).json({ error: 'Invalid password.' });
    }

    res.json({ message: 'Login successful.' });
  } catch (error) {
    res.status(500).json({ error: 'An error occurred.' });
  }
});

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Indexing in MongoDB

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

When not to create indexing:

- When collection is small
- When collection is updated frequently
- When queries are complex
- When collection is large and multiple indexes are applied.

If your application is repeatedly running queries on the same fields, you can create an index on those fields to improve performance. For example, consider the following scenarios:

Scenario	Index Type
A human resources department often needs to look up employees by employee ID. You can create an index on the employee ID field to improve query performance.	**Single Field Index**
A store manager often needs to look up inventory items by name and quantity to determine which items are low stock. You can create a single index on both the item and quantity fields to improve query performance.	**Compound Index**

- Indexes are special data structures that store a small portion of the collection's data set in an easy-to-traverse form.
- MongoDB indexes use a B-tree data structure.
- The index stores the value of a specific field or set of fields, ordered by the value of the field.

Default Index

MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index.

Index Names

The default name for an index is the combination of the **index keys** and each key's **direction** in the index **(1 or -1)** with **underscores** as a separator.

For example, an index created on { item : 1, quantity: -1 } has the name item_1_quantity_-1.

xplain() method

Provides information on the query plan for the db.collection.find() method.
The explain() method has the following form:
db.collection.find().explain()
The following example runs cursor.explain() in "executionStats" verbosity mode to return the query planning and execution information for the specified db.collection.find() operation:

```
db.students.find({age:{$gt:15}}).explain("executionStats")
```

Explain(executionStats) is a method that you can apply to simple queries or to cursors to investigate the query execution plan. The execution plan is how MongoDB resolves a query. Looking at all the information returned by explain():

- **totalDocsExamined**: how many documents were examined
- **nReturned**:how many documents were returned
- **stage:** Inside the winningPlan -> queryPlan -> stage is defined. It is used to defined the stage of input scanning.
    - COLLSCAN for a collection scan
    - IXSCAN for scanning index keys

Creating an Index

MongoDB provides a method called createIndex() that allows user to create an index.

Syntax: db.**collection_name**.createIndex({KEY:1})

MongoDB only creates the index if an index of the same specification does not exist.
The key determines the field on the basis of which you want to create an index and 1 (or -1) determines the order in which these indexes will be arranged

1 for ascending order and -1 for descending order

1. Create an Index on a Single Field

We can create an index on any one of the fields of collection.

Example:

db.table1.createIndex({age:1})

This will create an index named "age_1".

***Read documents using Index:**

Syntax: db.collection_name.find().explain("executionStats")

2. Compound indexing

Compound indexes are indexes that contain references to multiple fields. Compound indexes improve performance for queries on exactly the fields in the index or fields in the index prefix.
To create a compound index, use the db.collection.createIndex() method:
db.<collection>.createIndex({<field1>:<sortOrder>,<field2>:<sortOrder>,...,<fieldN>: <sortOrder>})

Example:

db.table1.createIndex({age:1,name:-1})

An index created on { age : 1, name: -1 } has the name age_1_name_-1.

In this example:

- The index on age is ascending (1).
- The index on name is descending (-1).

The created index supports queries that applies on:

- Both age and name fields.
- Only the age field, because age is a prefix of the compound index.

For example, the index supports these queries: (Perform IXSCAN)

db.students.find({ name: "Abc", age: 36 }) db.students.find({ age: 26 })

The index does not support queries on only the name field, because name is not part of the index prefix. For example, the index does not support this query:

db.students.find({ name: "Abc" }) →perform CoLLSCAN

- getIndexes()

To confirm that the index was created, use below command:

db.collection.getIndexes()

Output:

[{ v: 2, key: { _id: 1 }, name: '_id_' },{ v: 2, key: { age: 1 }, name: 'age_1' }]

- dropIndex()/dropIndexes()

dropIndex() : Drops or removes the specified index from a collection.

db.collection.dropIndex(index)

dropIndexes() : It is used to drop all indexes except the _id index from a collection.

db.collection.dropIndexes()

Drop a specified index from a collection. To specify the index, you can pass the method either:

- **The index specification document**
  db.collection.dropIndexes({ a: 1, b: 1 })
- The index name:
  db.collection.dropIndexes("a_1_b_1")
- Drop specified indexes from a collection. To specify multiple indexes to drop, pass the method an array of index names:
  db.collection.dropIndexes(["a_1_b_1", "a_1", "a_1_id_-1"])

If the array of index names includes a non-existent index, the method errors without dropping any of the specified indexes

To get the names of the indexes, use the db.collection.getIndexes() method.

Let's understand the concept with following example.

Suppose, we have a collection named student and we want to apply query to find students who **have age greater than 12**.

Student Collection

_id	Name	Age
1	ABC	10
2	XYZ	12
3	ABC	15
4	PQR	13
5	XYZ	15
6	ABC	8

- Find students without creating an index.

db.student.find({age:{$gt:12}}).explain("executionStats")

It will examine 6 documents and return 3 documents by performing COLLSCAN.
totalDocsExamined: 6
nReturned:3

- **Create an index on the age field to improve performance for those queries:**

db.students.createIndex({ age: 1 }) index
name: age_1

- **Now, find the students with age greater than 12**

db.student.find({age:{$gt:12}}).explain("executionStats")

It will examine 3 documents and return 3 documents by performing IXSCAN.
totalDocsExamined: 3

nReturned: 3

- **Create an index on the age and name fields to improve performance.(Compound Indexing)**

Note: Before creating any other indexing on same fields. Drop the created indexing

db.students.createIndex({ age: 1,name:-1 })
index name: age_1_name_-1

_id	name	age
1	ABC	8
2	ABC	10
3	XYZ	12
4	PQR	13
5	XYZ	15
6	ABC	15

- So here on name field indexing is applied based on age field.
- In this example we have two students with age 15. For age field values 15 the name field values are arranged in descending order.
- name field indexing is depending on the age field. So if we apply query on name field then it will perform the collscan.

Now, consider below three scenarios

1. Display student with name “ABC” and age 15

db.student.find({age:15,name:”ABC”}).explain("executionStats")

It will examine 1 documents and return 1 documents by performing IXSCAN.
totalDocsExamined: 1

nReturned: 1

db.student.find({age:{$gt:12}).explain("executionStats")

It will examine 3 documents and return 3 documents by performing IXSCAN.
totalDocsExamined: 3

nReturned: 3

2. Display students whose age is greater than 12

3. Display students whose name is “ABC”

db.student.find({name:”ABC”}).explain("executionStats")

It will examine 6 documents and return 3 documents by performing COLLSCAN.
totalDocsExamined: 6

nReturned: 3 stage:

3. Partial Indexing

The partial index functionality allows users to create indexes that match a certain filter condition. Partial indexes use the partialFilterExpression option to specify the filter condition. The partialFilterExpression option accepts a document that specifies the filter condition using:

- equality expressions (i.e. field: value or using the $eq operator),
- $exists: true expression,
- $gt, $gte, $lt, $lte expressions,
- $type expressions,
- $and operator,
- $or operator,
- $in operator

Example:

Consider student collection with 6 documents

- **Create index on age where age is greater than 15**

db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:15}}})

- **Display student/s whose age is 16.**

db.student.find({age:16}).explain("executionStats")

Output: stage: 'IXSCAN',

nReturned: 1,

docsExamined: 1

Suppose we have only one document available with age value 16.

- **Display student/s whose age is 13.**

db.student.find({age:13}).explain("executionStats")

Output: stage: 'COLLSCAN',

nReturned: 1,

docsExamined: 6

Suppose we have only one document available with age value 13.

We have applied index on age field where age is greater than 15. we are looking for age 13 which is less than 15. Then here it will perform the COLLSCAN.

- **Display student/s whose age is 17.**

db.student.find({age:17}).explain("executionStats")

Output: stage: 'IXSCAN',

nReturned: 2,

docsExamined: 2

Suppose we have only 2 documents available with age value 17.

We have applied index on age field where age is greater than 15. we are looking for age 17 which is greater than 15. Then here it will perform the IXSCAN.

Winning plan:

If we have applied same field indexing on multiple times, then searching can be done using one type of indexing only. In this scenario, it races with one indexing strategy and only one indexing strategy wins, the other strategy is stored in loosing plan (rejected plan).
The winning plan is stored in cache till 1000 write operations and next time it does not race with matching index. So, this index is stored in winning plan and the other index is stored in rejected plan.
Let's say we have already one index on age and one more index apply on age like:

Code: db.DATA.createIndex({age:1,name:1})
db.DATA.find({age:19}).explain("executionStats")

so, two indexes have been created on age field, then it selects only one index for searching called inside winning plan, rest of the index is called as rejected plan.

Consider a collection student having documents like this:
[
{_id:123433,name: "DDD",age:32},
{_id:123434,name: "BBB",age:20},
{_id:123435,name: "AAA",age:10},
]
Do as directed:
(1) Create an index & fire a command to retrieve a document having age>15 and name is "BBB". Stats must return values nReturned=1, docExamined=1, stage="IXSCAN". Perform required indexing.
(2) Create an index on subset of a collection having age>30. Also write a command to get a stats "IXSCAN" for age>30.

Example:

Solution:

1) db.student.createIndex({age:1,name:1})
   db.student.find({age:{$gt:15},name:'DDD'}).explain('executionStats')

2) db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:30}}})
   db.student.find({age:{$gt:30}}).explain('executionStats')

MongoDb Replication

Replication is the process of synchronizing data across multiple servers. It provides redundancy and increase data availability with multiple copies of data on diff. DB servers.

Replication protect a Database from loss of a single server. It also allows you to recover from Hardware failure and service interruption.

A replica set in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments

Redundancy and Data Availability

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centres can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Replication Key Features :

- Replica sets are the clusters of N different nodes that maintain the same copy of the data set.
- The primary server receives all write operations and record all the changes to the data/
- The secondary members then copy and apply these changes in an asynchronous process.
- All the secondary nodes are connected with the primary nodes. there is one heartbeat signal from the primary nodes. If the primary server goes down an eligible secondary will hold the new primary.

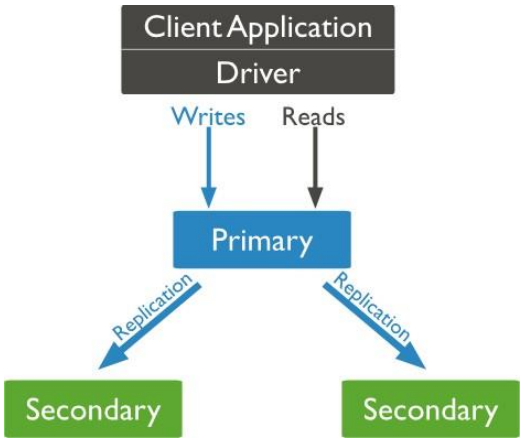
Why Replication?

- High Availability of data disasters recovery
- No downtime for maintenance (like backups index rebuilds and compaction)
- Read Scaling (Extra copies to read from)

How replication works

Replica set is a group of two or more nodes.

In a replica set, one node is Primary node and remaining nodes are secondary.



- Data of primary server is copied to secondary. This duplication is done asynchronously.

Sharding

Sharding is a method for allocating data across multiple machines. MongoDB used sharding to help deployment with very big data sets and large throughput the operation. By sharding, you combine more devices to carry data extension and the needs of read and write operations

Sharding solve the the problem of horizontal scaling.

The vertical scaling approach

The vertical scaling approach, sometimes referred to as "scaling up," focuses on adding more resources or more processing power to a single machine. These additions may include CPU and RAM resources upgrades which will increase the processing speed of a single server or increase the storage capacity of a single machine to address increasing data requirements.

The horizontal scaling approach

The horizontal scaling approach, sometimes referred to as "scaling out," entails adding more machines to further distribute the load of the database and increase overall storage and/or processing power. There are two common ways to perform horizontal scaling — they include sharding, which increases the overall capacity of the system, and replication, which increases the availability and reliability of the system.

Automatic Failover

When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period (10 seconds by default), an eligible secondary calls for an election to nominate itself as the new primary. The cluster attempts to complete the election of a new primary and resume normal operations.

In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary.

Steps for Replication

Primary server on port 27018

Secondary server on port 27019

Steps	Command
1	Prepare new folder data inside that another folder named db. Inside that make two sub folder db1 and db2.
2	Open **command prompt(cmd 1)** and run below command to create mongodb server on port number 27020 and store data at mentioned database path. mongod --port 27018 --dbpath "D:\data\db\db1" --replSet rs1
3	Open **command prompt(cmd 2)** and run below command to create mongodb server on port number 27021 and store data at mentioned database path. mongod --port 27019 --dbpath "D:\data\db\db2" --replSet rs1
4	Open **command prompt (cmd no.3)** to check replica created or not by using below command mongosh --port 27018 (if it opens "test>" then created successfully)
5	To initiate replica set **cmd no.3** The following example initiates a new replica set with two members. rs.initiate({ _id:"rs1", members:[{_id:0, host:"127.0.0.1:27018"}, {_id:1, host:"127.0.0.1:27019"}] })
6	**rs.status():** on Cmd no.3 **rs.status() –** Returns the replica set status from the point of view of the member where the method is run. Shows that 127.0.0.1:27018 is the primary server and 127.0.0.1:27019 is the Secondary serve
7	To lookout present database on server port 27018 hit below command in cmd no.3 rs1 [direct: primary] test> show dbs 3 Dbs named admin,config and local. On New server port :27018

8	Switch database by using below command in Cmd no 3: rs1 [direct: primary] test> use pspdata Insert record on primary server rs1 [direct: primary] pspdata> db.people.insertMany([{ name:"ABC",age:29,dept:"IT"},{ name:"PQR",age:29,dept:"IT"}]) This will also create a replica of the mydb database on 27019 port. Open mondoDB compass. Connect with port 27019 and check the database and collection.
9	Data automatically inserted in secondary server. Permission required to perform read operation. **Open cmd 4 and hit below to command to reach specific database** • mongosh --port 27019 • test>use pspdata To specify Read Preference Mode The following operation sets the read preference mode to target the read to a primary member. This implicitly allows reads from primary. **Command to hit: db.getMongo().setReadPref("primaryPreferred")** **Read Preference Mode**s 1. **primary:** Default mode. All operations read from the current replica set primary. 2. **primaryPreferred:** In most situations, operations read from the primary but if it is unavailable, operations read from secondary members. 3. **secondary:** All operations read from the secondary members of the replica set. 4. **secondaryPreferred:** Operations typically read data from secondary members of the replica set. If the replica set has only one single primary member and no other members, operations read data from the primary member. 5. **nearest:** Operations read from a random eligible replica set member, irrespective of whether that member is a primary or secondary, based on a specified latency threshold.
10	Run following command to read documents of people collection from secondary server 27019. **db.people.find()**
11	Try to enter data in secondary server db.people.insertOne({name:"a1"}). It will not allow and gives message not primary.

Example 1: create collections named demo having fields name,age,dep,salary,gender

1.insert 7 records into demo collection
2. find name and age of all person who are female
db.demo.find({gender:'female'},{name:true,age:true,_id:false})
3.update the sal of all by 5000
db.demo.updateMany({},{$inc:{sal:5000}})
4.arrange all records in asscending order of name
db.demo.find().sort({name:1})
5.update all person who are male into female
db.demo.updateMany({gender:'male'},{$set:{gender:"female"}})
6.delete all records whose salary are gt 70000
db.deleteMany({sal:{$gt:70000}})

Example 2: create a collection faculty having fields fname, lname, sub, exp, age, gender, sal

1. enter 10 records into fac table
2. find all fnames where abc occurs as a substring or as a seperate word
db.fac.find({fname:/abc/},{fname:1,lname:1,_id:0})
3. update document if lname beginning with P update their sal by half
db.fac.updateMany({lname:/^P/i},{$mul:{sal:0.5}})
4. find all records whose age are end with 0/5
db.fac.find({age:/0|5$/})
5. update first record by gender to others
db.fac.updateOne({gender:'others'})
6. count records who are teaching fsd-2 sub
db.fac.find({sub:'fsd-2'}).count()
7. update all record and insert current date to that record
db.fac.updateMany({},{$set:{date:new Date()}})
8. update sal of person whose sal less than 20000 to double
db.fac.updateMany({sal:{$lt:20000}},{$mul:{sal:2}})
9. update subject java to fsd-2 if java is not found insert using updateOne
db.fac.updateOne({sub:'java'},{$set:{sub:'fsd-2'}},{upsert:true})
10. delete all records and table
db.fac.deleteMany({}) or db.fac.drop()

Example 3

Insert 10 documents with random data with fields _id,brand,price,cat as shown below.

```
db.product.insertMany([
{_id:1,brand:"samsung",price:29000,cat:"mobile"},
{_id:2,brand:"nokia",price:5000,cat:"mobile"},
{_id:3,brand:"vivo",price:16000,cat:"mobile"},
{_id:4,brand:"samsung",price:60000,cat:"tv"},
{_id:5,brand:"samsung",price:40000,cat:"washing machine"},
{_id:6,brand:"ifb",price:45000,cat:"wasing machine"},
```

{_id:7,brand:"apple",price:120000,cat:"mobile"},
{_id:8,brand:"oppo",price:20000,cat:"mobile"},
{_id:9,brand:"sony",price:80000,cat:"tv"},
{_id:10,brand:"vivo",price:31000,cat:"mobile"},
])

1) Display price and brand of product which are of mobile cat.
2) Increase price of each Samsung products by 1000.
3) Update all vivo product by adding field quantity and add random value
4) Display price of products which are of vivo or oppo brand.
5) Display brand and cat of products which are less than 80000 and greater than or equal to 30000.

Answers:

1) db.product.find({cat:”mobile”},{cat:0,_id:0})
2) db.product.updateMany({brand:"samsung"},{$inc:{price:1000}})
3)
```
db.product.updateMany({brand:"vivo"},{$set:{quantity:5}})
```
4) db.product.find({$or:[{brand:"vivo"},{brand:"oppo"}]},{price:1,_id:0})
5) db.product.find({price:{$lt:80000,$gte:30000}},{price:0,_id:0})

Example 4

Consider following student collection:
[
{_id:123433,name: "SSS",age:22},
{_id:123434,name: "YYY",age:2},
{_id:123435,name: "PPP",age:32},
]
Do as directed:
(1) Update name=”JJJ” and age=40, where age=20 occurs. Insert new document, if
record is not found.
(2) To retrieve age and name fields of documents having names “YYY” & “SSS”. Don’t
project _id field.

Answers:

1) db.info.updateMany({age:20},{$set:{name:'JJJ',age:40}},{upsert:true})
2) db.info.find({$or:[{name:'YYY'},{name:'SSS'}]},{_id:0}