**What is Mongodb ?**

* MongoDB is a document-oriented NoSQL database which is used to store huge data as documents. It has collection just like tables in relational databases. It has no schema. We can use JSON object to store data here but behind the scenes mongodb server stores this json into binary format.

**What is mongod?**

* It is a ececutable file, used to start the mongodb server locally

**What is mongo/mongosh?**

* It is a mongodb shell, used to connect to mongodb to execute our queries.

We can specify the location where we want to save our data in local. But it should have data and logs folder inside it. Then start the server like the following:

*mongod --dbpath /path/data --logpath /path/logs/mongo.log*

In Windows there is an option to start mongodb as a service so it will be running all the time in background.

**How do I start/stop MongoDB from running in the background in windows?**

* One liner to start or stop mongodb service using command line in windows.

1. To start the service use: *NET START MONGODB.*

2. To stop the service use: *NET STOP MONGODB.*

**How do I start/stop MongoDB from running in the background in MAC/linux?**

* --fork option is used to run mongoDB in background.

mongod --port 8888 --dbpath /Users/Shared/data/db --logpath /Users/Shared/log/mongo.log –fork

We can shut down the mongodb by first switching to admin db then use this command db.shutdownServer()

**Command to show all the database**: *show dbs*

**Create or use a database**: *use <db\_name>*

**To use a collection and store one data**: *db.products.insertOne({name:"Abhishek Ghosh",age:24})* it will create a document in products collection. After inserting one document it will give one id and acknowledgement. We can also insert nested documents.

**To show all the datas in products collection use this command**: *db.products.find()*

**To show it in a json structure:** *db.products.find().pretty()*

By default, mongodb adds an unique id which is of type ObjectId to every document and we can search items with that and also mongodb create one default index with this \_id by default. We can also add our \_id like the following

*db.products.insertOne({\_id:"abhishek-test-0001",name:"Abhishek Ghosh"})*

**To search any document using \_id**: *db.products.find({\_id:ObjectId('62a6ff6edb132197c5e887a0')})*

Mongodb uses BSON instead of JSON to store data.

CRUD Operations

Create operations:

* insertOne(data, options) -> for inserting one item
* insertMany(data, options) -> for inserting multiple items

Read operations:

* find(filter, options) -> find all the data based on the filter
* findOne(filter, options) -> find the first matching element based on the filter

Update operations:

* updateOne(filter, data, options) -> to update one document
* updateMany(filter, data, options) -> for updating multiple documents
* replaceOne(filter, data, options) -> for replacing the entire document

Delete operations:

* deleteOne(filter, options) -> delete only the first item with matching filter
* deleteMany(filter, options) -> delete all items matching with the filter

**Delete the first element with name with “Abhishek Ghosh”** -> *db.products.deleteOne({name:"Abhishek Ghosh"})*

**Update the age to 24 where name is “Abhishek Pal”** -> *db.products.updateOne({name:"Abhishek Pal"},{$set:{age:24}})*

**Add a field height to all the documents** -> *db.products.updateMany({},{$set:{height:"Unknown"}})*

**{} this means all the documents**

**Insert two items at a time ->**

*db.products.insertMany(*

*... [{name:"Nasim Molla",*

*... age:25},*

*... {name:"Sayan Mandal",*

*... age: 24}])*

**Find all the students whose age is greater than 24** -> *db.products.find({age:{$gt:24}})*

**Print all the names for the student whose age is greater than 24 (no \_id)** -> *db.products.find({age:{$gt:24}},{"name":1,\_id:0})*

**If we use update without $set then the document will be replaced with the data we have provided**.( Rather use replace than update for full replacement)

> **db.products.insertOne({})**

{"acknowledged" : true,"insertedId" : ObjectId("62a7faec7866653913689afd")}

> **db.products.update({\_id:ObjectId("62a7faec7866653913689afd")},{name:"Anirban Ghosh",age:23})**

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> **db.products.find({\_id:ObjectId("62a7faec7866653913689afd")})**

{ "\_id" : ObjectId("62a7faec7866653913689afd"), "name" : "Anirban Ghosh", "age" : 23 }

**What is cursor?**

* When we find anything with shell rather than giving everything in one shot it gives us the cursor of 20 elements and to move to the next 20 we have to enter “it”. To see it we can use toArray method on the cursor which will exhaust the cursor and make one array with all the elements and show that.
* Cursor will fetch only the needed element.
* findOne will not give us cursor object as it will only give us one element.
* db.products.find().toArray()
* db.products.find().forEach((doc)=>{printjson(doc)})

**What is projection?**

* Rather than show all the fields of a document we can choose whatever we want to show.
* It will also helps us to reduce the bandwidth usage as server will not send all the elements.
* To get all the student with age is 24 : *db.products.find({age:24},{"name":1})*
* By default \_id is set to 1 so if we want to remove is as well we have to use this type of query. *db.products.find({age:24},{"name":1,\_id:0})*

**One Document can maximum hold 100 level of nesting**

**Maximum size of document can be 16 mb**

**Set status object for age greater than 24** -> *db.products.updateMany({age:{$gt:24}},{$set:{status:{married:false,single:false}}} )*

I**f we have a list of strings then like hobbies then we can search like this (It will find the first document that has a list of hobbies containing “Drama**” -> *db.products.findOne({hobbies:”Drama”})*

**We can we run a query in nested object** -> *db.products.findOne({“status.single”: false})*

To get rid of your data, you can simply load the database you want to get rid of (use databaseName) and then execute db.dropDatabase().

Similarly, you could get rid of a single collection in a database via db.myCollection.drop().

**Data Types ->**

1. Text -> “Abhishek Ghosh”
2. Boolean -> true
3. Number-> NimberInt() 1, Integer(int32) 55, NumberLong(int64) 1000000000, NumberDecimal 12.0009
4. ObjectId -> ObjectId("62a6fddadb132197c5e8879f")
5. ISODate -> 2022-06-14T05:45:29.379+00:00
6. Timestamp
7. Embedded Documents
8. Arrays

Db.stats() will bring the statistic of the database.

MongoDB has a couple of hard limits - most importantly, a single document in a collection (including all embedded documents it might have) must be <= 16mb. Additionally, you may only have 100 levels of embedded documents.You can find all limits (in great detail) here: https://docs.mongodb.com/manual/reference/limits/

For the data types, MongoDB supports, you find a detailed overview on this page: <https://docs.mongodb.com/manual/reference/bson-types/>

Important data type limits are:

* Normal integers (int32) can hold a maximum value of +-2,147,483,647
* Long integers (int64) can hold a maximum value of +-9,223,372,036,854,775,807
* Text can be as long as you want - the limit is the 16mb restriction for the overall document

It's also important to understand the difference between int32 (NumberInt), int64 (NumberLong) and a normal number as you can enter it in the shell. The same goes for a normal double and NumberDecimal.NumberInt creates a int32 value => NumberInt(55) and NumberLong creates a int64 value => NumberLong(7489729384792)If you just use a number (e.g. insertOne({a: 1}), this will get added as a normal double into the database. The reason for this is that the shell is based on JS which only knows float/ double values and doesn't differ between integers and floats.NumberDecimal creates a high-precision double value => NumberDecimal("12.99") => This can be helpful for cases where you need (many) exact decimal places for calculations.

When not working with the shell but a MongoDB driver for your app programming language (e.g. PHP, .NET, Node.js, ...), you can use the driver to create these specific numbers.

Example for Node.js: http://mongodb.github.io/node-mongodb-native/3.1/api/Long.html

This will allow you to build a NumberLong value like this:

const Long = require('mongodb').Long;

db.collection('wealth').insert( {

value: Long.fromString("121949898291")

});

Embedded documents vs reference id

Embedding is better for...

* Small subdocuments
* Data that does not change regularly
* When eventual consistency is acceptable
* Documents that grow by a small amount
* Data that you’ll often need to perform a second query to fetch Fast reads

References are better for...

* Large subdocuments
* Volatile data
* When immediate consistency is necessary
* Documents that grow a large amount
* Data that you’ll often exclude from the results
* Fast writes

Refference : <https://www.mongodb.com/docs/manual/core/data-model-design/>

**We can also use aggregation framework for joining.**

The MongoDB Lookup operator, by definition, “Performs a left outer join to an unshared collection in the same database to filter in documents from the “joined” collection for processing.” Simply put, using the MongoDB Lookup operator makes it possible to merge data from the document you are running a query on and the document you want the data from.

**More can be found in the following links**

<https://hevodata.com/learn/mongodb-lookup/#:~:text=The%20MongoDB%20Lookup%20operator%2C%20by,a%20query%20on%20and%20the>

<https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/>

Though Mongodb is schema less but we real life scenario we must have certain type of structure. We can add validators when we are creating any collection.

db.createCollection('posts', {

  validator: {

    $jsonSchema: {

      bsonType: 'object',

      required: ['title', 'text', 'creator', 'comments'],

      properties: {

        title: {

          bsonType: 'string',

          description: 'must be a string and is required'

        },

        text: {

          bsonType: 'string',

          description: 'must be a string and is required'

        },

        creator: {

          bsonType: 'objectId',

          description: 'must be an objectid and is required'

        },

        comments: {

          bsonType: 'array',

          description: 'must be an array and is required',

          items: {

            bsonType: 'object',

            required: ['text', 'author'],

            properties: {

              text: {

                bsonType: 'string',

                description: 'must be a string and is required'

              },

              author: {

                bsonType: 'objectId',

                description: 'must be an objectid and is required'

              }

            }

          }

        }

      }

    }

  }

});

If the collection is already created, then we can use run command to add validations and also we can add validation level

db.runCommand({

  collMod: 'posts',

  validator: {

    $jsonSchema: {

      bsonType: 'object',

      required: ['title', 'text', 'creator', 'comments'],

      properties: {

        title: {

          bsonType: 'string',

          description: 'must be a string and is required'

        },

        text: {

          bsonType: 'string',

          description: 'must be a string and is required'

        },

        creator: {

          bsonType: 'objectId',

          description: 'must be an objectid and is required'

        },

        comments: {

          bsonType: 'array',

          description: 'must be an array and is required',

          items: {

            bsonType: 'object',

            required: ['text', 'author'],

            properties: {

              text: {

                bsonType: 'string',

                description: 'must be a string and is required'

              },

              author: {

                bsonType: 'objectId',

                description: 'must be an objectid and is required'

              }

            }

          }

        }

      }

    }

  },

  validationAction: 'warn'

});

Helpful Articles/ Docs:

* The MongoDB Limits: <https://docs.mongodb.com/manual/reference/limits/>
* The MongoDB Data Types: <https://docs.mongodb.com/manual/reference/bson-types/>
* More on Schema Validation: <https://docs.mongodb.com/manual/core/schema-validation/>

We can configure mongodb server in with various arguments. We can check all in mongod –help command.

We can also use mongod.cfg to put all our configurations in a file and we can put it inside any folder and to we have use that file when we are about to start the server.

storage:

  dbPath: "/your/path/to/the/db/folder"

systemLog:

  destination:  file

  path: "/your/path/to/the/logs.log"

mongod -f /path/mongod.cfg

Reference: <https://www.mongodb.com/docs/manual/reference/configuration-options/>

We can anytime check mongo –help to find all the commands.

Helpful Articles/ Docs:

* More Details about Config Files: <https://docs.mongodb.com/manual/reference/configuration-options/>
* More Details about the Shell (mongo) Options: <https://docs.mongodb.com/manual/reference/program/mongo/>
* More Details about the Server (mongod) Options: <https://docs.mongodb.com/manual/reference/program/mongod/>