**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/>

**CREATE**

We have three methods for inserting documents 1. insertOne 2. insertMany 3. insert. Though insert method is flexible enough to handle one document or multiple but still it is deprecated on purpose.

Also, we can directly import from a json file using mongoimort command

If are using insert many and we are inserting multiple documents in a shot then if there is a issue with any document then from that onwards there will be no insertions, only the documents before the wrecked document will be inserted, it will not be rolled back.

Like for the following code there is a issue in third document

**> db.hobbies.insertMany([{\_id:"yoga"},{\_id:"sports"},{\_id:"yoga"},{\_id:"maths"}])**

"errmsg" : "E11000 duplicate key error collection: contacts.hobbies index: \_id\_ dup key: { \_id: \"yoga\" }",

**> db.hobbies.find().toArray()**

[ { "\_id" : "yoga" }, { "\_id" : "sports" } ]

But to remove this one we can pass one argument {ordered: false}. By default, it is true. It defines that the insertion will be ordered or not.

If we again try to run the previous code in shell it will again give us the error, but it will not stop to the error document rather it will insert all the correct documents.

**> db.hobbies.insertMany([{\_id:"yoga"},{\_id:"sports"},{\_id:"yoga"},{\_id:"maths"}],{ordered:false})**

"E11000 duplicate key error collection: contacts.hobbies index: \_id\_ dup key: { \_id: \"yoga\" }", "E11000 duplicate key error collection: contacts.hobbies index: \_id\_ dup key: { \_id: \"sports\" }",

**> db.hobbies.find().toArray()**

[ { "\_id" : "yoga" }, { "\_id" : "sports" }, { "\_id" : "maths" } ]

> use contacts

switched to db contacts

> db.persons.insertOne({name:"Abhishek Ghosh"})

{

"acknowledged" : true,

"insertedId" : ObjectId("62aadb4256184ff0056adbd7")

}

> db.persons.insertMany([{name:"Abhishek Pal"},{name:"Bishal Mukherjee"}])

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("62aadbed56184ff0056adbd8"),

ObjectId("62aadbed56184ff0056adbd9")

]

}

**WriteConcern**

Write concern describes the level of acknowledgment requested from MongoDB for write operations to a standalone mongod or to replica sets or to sharded clusters. In sharded clusters, mongos instances will pass the write concern on to the shards.

the write concern is a specification of MongoDB for write operations that determines the acknowledgement you want after a write operation has taken place. MongoDB has a default write concern of always acknowledging all writes, which means that after every write, MongoDB must always return an acknowledgement (in a form of a document), meaning that it was successful. When asking for write acknowledgement, if none isn't returned (in case of failover, crashes), the write isn't successful. This behavior is very useful specially on replica set usage, since you will have more than one mongod instance, and depending on your needs, maybe you don't want all instances to acknowledge the write, just a few, to speed up writes. Also, when to specify a write concern, you can specify journal writing, so you can guarantee that operation result and any rollbacks required if a failover happens. More information, [here.](https://docs.mongodb.com/manual/reference/write-concern/)

In your case, it depends on how many mongod (if you have replica sets or just a single server) instances you have. Since "always acknowledge" is the default, you may want to change it if you have to manage replica sets operations and speed things up or just doesn't care about write acknowledgement in a single instance (which is not so good, since it's a single server only).

Write concern can include the following fields: {w: <value>, j: <boolean>, wtimeout: <number> }

Exp: {w: 1, j: true, wtimeout: 500}

the w option to request acknowledgment that the write operation has propagated to a specified number of mongod instances or to mongod instances with specified tags.

the j option to request acknowledgment that the write operation has been written to the on-disk journal, and

the wtimeout option to specify a time limit to prevent write operations from blocking indefinitely.

Write Concern Levels

MongoDB has the following levels of conceptual write concern, listed from weakest to strongest:

Unacknowledged:

With an unacknowledged write concern, MongoDB does not acknowledge the receipt of write operations. Unacknowledged is like errors ignored; however, drivers will attempt to receive and handle network errors when possible. The driver’s ability to detect network errors depends on the system’s networking configuration.

Write operation to a ``mongod`` instance with write concern of ``unacknowledged``. The client does not wait for any acknowledgment.

Acknowledged

With a receipt acknowledged write concern, the mongod confirms the receipt of the write operation. Acknowledged write concern allows clients to catch network, duplicate key, and other errors. This is default write concern.

Write operation to a ``mongod`` instance with write concern of ``acknowledged``. The client waits for acknowledgment of success or exception.

Journaled

With a journaled write concern, the MongoDB acknowledges the write operation only after committing the data to the journal. This write concern ensures that MongoDB can recover the data following a shutdown or power interruption.

You must have journaling enabled to use this write concern.

Write operation to a ``mongod`` instance with write concern of ``journaled``. The ``mongod`` sends acknowledgment after it commits the write operation to the journal.

Replica Acknowledged

Replica sets present additional considerations with regards to write concern. The default write concern only requires acknowledgement from the primary. With replica acknowledged write concern, you can guarantee that the write operation propagates to additional members of the replica set.

Write operation to a replica set with write concern level of ``w:2`` or write to the primary and at least one secondary.

Write operation to a replica set with write concern level of w:2 or write to the primary and at least one secondary.

For reference

* + <https://www.mongodb.com/docs/manual/reference/write-concern/>
  + <https://www.mongodb.com/docs/manual/core/journaling/>

When we have millions of records inserting in seconds then on that time, we can skip the acknowledgement and use **w: 0**. By default is

If **j: true**, then inserting will take some extra time as it will write on journal. By default, is undefined. Here it has the higher security

**Atomicity**

It means when we are inserting any document then either it will be saved as a whole, or it will not be saved at all if there is any issue. MongoDB provides atomic transaction guarantee.

Lastly, we can import json file in and save it mongodb

**mongoimport --db dbName --collection collectionName --file /path/fileName.json -**> if it is a single document

**mongoimport --db dbName --collection collectionName --file** **/path/fileName.json --jsonArray** -> if it is a array of documents

if we add --drop then it will delete previous data

**Read**

1. Methods, Filters and Operators
2. Query Selectors
3. Projection Operators

Operators are reserved fields started with dollar like $gt, $gte, $lt, $lte

There are two methods ->

1. find -> returns all the documents which satisfies the criteria (basically it returns the cursor object)
2. findOne -> it returns a first document that satisfies the criteria

**> db.products.findOne({age:24})** -> to get the document where age is 24

**> db.products.findOne({age:{$gt:24}})** -> to get the document where age is greater than 24

Query Selectors:

1. Comparison
2. Evaluation
3. Logical
4. Array
5. Element
6. Comments
7. Geospatial

Projection Operator:

1. $
2. $elemMatch
3. $meta
4. $slice

find method gives a cursor of 20 objects

**db.infos.find({"name": "Under the Dome"},{"name":1,"type":1,"language":1})** -> first it will search the document where name is "Under the Dome" then it only return name, type and language

**db.infos.findOne({runtime:60}) / db.infos.findOne({runtime:{$eq:60}})** -> runtime equal to 60

**db.infos.findOne({runtime:{$ne:60}})** -> runtime not equal to 60

**db.infos.findOne({runtime:{$gt:60}})** -> runtime greater than 60

**db.infos.findOne({runtime:{$gte:60}})** -> runtime greater than equal to 60

**db.infos.findOne({runtime:{$lt:60}})** -> runtime less than 60

**db.infos.findOne({runtime:{$lte:60}})** -> runtime less than equal to 60

**db.infos.find({runtime: {$in: [30,42]}}) ->** it will find all the documents where runtime is either 30 or 42.

**db.infos.find({runtime: {$nin: [30,42]}}) ->** it will find all the documents where runtime is neither 30 nor 42.

**> db.infos.findOne({"rating.average": {$gt: 9}})** -> average is a field which is inside of rating, so to querying anything in average we can use something like this layer1.layer2.layer3.targetField then our query operator

**> db.infos.findOne({"genres": "Drama"})** -> here genres is a array. If we search for this, it will not equate as a string it will check that **genres** contain **Drama** or not

**db.infos.find({$or : [{"rating.average": {$gt:8}},{"rating.average": {$lt:7}}]})** -> $or operator takes an array of queries. Here average is either greater than 8 or less than 7. We can combine more than two queries.

**db.infos.find({$nor : [{"rating.average": {$gt:8}},{"rating.average": {$lt:7}}]})** -> $nor operator takes an array of queries. Here average is neither greater than 8 nor less than 7. We can combine more than two queries.

**db.infos.find({$and : [{"rating.average": {$lt:8}},{"rating.average": {$gt:7}}]})** -> $and operator takes an array of queries. Here average is less than 8 and greater than 7. We can combine more than two queries. We have a short cut for and query.

1. **db.infos.find({$and : [{"rating.average": {$lt:8}},{"runtime": {$gte:60}}]})**
2. **db.infos.find({"rating.average": {$lt:8}, "runtime": {$gte:60}})**

these two queries are same as mongodb by default does the and operation and equal to operation

we have also **$not** operator that we can use like this. $not is just like another wrapper to the existing query

not of this query **db.infos.find({"rating.average": {$lt:8}}).count()** will be **db.infos.find({"rating.average": {$not :{$lt:8}}}).count()**

There are two element type operators **$exist** and **$type**

As mongodb is schemaless so sometimes there may be a case a field may or may not be exist so we can check that a field is exist or not like this:

**db.users.findOne({“age”: {$exists: true}}) ->** age field exists

We can use exists with another query as well:

**db.users.findOne({“age”: {$exists: true, $gte: 30}}) ->** age field exists and greater than 30

**db.users.findOne({“age”: {$exists: true, $ne: null}}) ->** age field exists and not equal to null

As mongodb is schemaless so sometimes there may be a case a field may or may not have the same data type for all the document so we can check that a field has the datatype or not with **$type:**

**db.users.findOne({“phoneNo”: {$type: “double”}})** -> phone no is double in which document

**db.users.findOne({“phoneNo”: {$type: “string”}})** -> phone no is string in which document

**db.users.findOne({“phoneNo”: {$type: [“double”, “string”]}})** -> phone no is string or double in which document. We can use array. It will act as OR operator here

**db.infos.find({summary: {$regex: /musical/}})** -> It will use regex to search any document have the musical word in the summary or not. But it is not that efficient better to use text indexing

**db.infos.find({$expr: {$gt: ["$weight", "$runtime"]}})** -> it will search all the documents where weight is greater that runtime. We can use **$expr** like this where it will take the query inside it.

We can use **if, then** an inside **$cond** and the **$expr** will evaluate everything.

**Querying to Arrays**

Let’s say experience is an array having many fields like college name, company name, start date end date etc

**db.products.find({"experiences.companyName": "Kreeti"})** -> it will search the document where in **experiences array** there will be a object in which **companyName** field will be **Kreeti**

We can use dot operator with array and embedded documents

**db.products.find({"experiences": {$size: 3}})** -> find all the documents where experience is length of **3**. **$size** operator takes only **equality** it will not work with **$gt** or **$lt** like the following query: **db.products.find({"experiences": {$size: {$gt: 2}}})** . It will give us the exception.

**db.infos.find({genres: ["Drama", "Crime", "Thriller"]})** -> It will only search for the documents where **genres** is **["Drama", "Crime", "Thriller"]** particularly in this order but if the order does not matter for us then we can use **$all**

**db.infos.find({genres: {$all: ["Drama", "Crime", "Thriller"]}})** -> It will search for all the documents where these three items **["Drama", "Crime", "Thriller"]** are there in the **genres** array.

Certainly, these two queries will not give us the same result:

1. **db.infos.find({genres: {$all: ["Drama", "Crime"]}}).count()** -> 47
2. **db.infos.find({genres: ["Drama", "Crime"]}).count()** -> 12

**Find how many persons are working in TCS or not:**

**Probable answers are :**

1. **db.products.find({"experiences.companyName": "TCS","experiences.currentlyInHere": true}).count()**
2. **db.products.find({$and: [{"experiences.companyName": "TCS"},{"experiences.currentlyInHere": true}]}).count()**

If we use this query ideally it should return 1 as there is only one document where in one experience item **companyName** is **TCS** and **currentlyHere** is **true** but this query does not work like that it will check in the arrays that if any object has the **companyName** as **TCS** and **currentlyHere** is **true.** It does not need to be the same object in the array. Here we could use the **$elemMatch**. It will search for all the queries in the same item of the array.

We can achieve our requirement of any person who is currently working in TCS or not with the below query:

**db.products.find({experiences: {$elemMatch: {companyName: "TCS",currentlyInHere: true}}}).count()**

**$elemMatch** will match all the queries for every element in the array.