**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.

**Cursor**

In MongoDB, the **find()** method return the cursor, now to access the document we need to iterate the cursor. In the mongo shell, if the cursor is not assigned to a var keyword then the mongo shell automatically iterates the cursor up to **20 documents**. MongoDB also allows you to iterate cursor manually. So, to iterate a cursor manually simply assign the cursor return by the **find()** method to the var keyword Or JavaScript variable.

Note: If a cursor inactive for 10 min, then MongoDB server will automatically close that cursor.

**db.infos.find().pretty()** -> It will fetch the cursor of first 20 elements

**db.infos.find().toArray()** -> It will exhaust the cursor and make all the documents as array of objects

**db.infos.find().count()** -> It give us the count of all the element

**db.infos.find().hasNext()** -> it will say if the cursor has exhausted or not

**db.infos.find().next()** -> it will give the current 20 elements of the cursor

**db.infos.find().forEach((doc)=> printjson(doc))** -> printjson is a method in shell. forEach is a function on the cursor

**db.infos.find().sort({"rating.average" :1})** -> It will sort all the elements on average element on rating.

**db.infos.find().sort({"rating.average" :1, "runtime": -1})** -> It will sort all the elements on **average** element on **rating** and then **runtime** but backwards

**db.infos.find().sort({"rating.average" :1}).skip(10)** -> It will sort all the elements on average element on rating then **skip the first 10 elements**

**db.infos.find().sort({"rating.average" :1}).limit(2)** -> It will sort all the elements on average element on rating then only show the first 2 elements

**db.infos.find().sort({"rating.average" :1}).skip(2).limit(2)** -> It will sort all the elements on average element on rating then **skip 2** elements and **show only 2** elements

**db.infos.find({},{name: 1})** -> It will show only the name and the \_id of first 20 documents. \_id is shown by default.

**db.infos.find({},{\_id: 0, name: 1})** -> It will show only the name of first 20 documents.

**db.infos.find({},{name: 1, "schedule.time": 1})** -> It will show only the **name** and schedule object with only time field and the **\_id** of first 20 documents.

**db.infos.find({genres: "Thriller"},{"genres.$": 1})** -> It will first search for the documents with **genres** with **Thriller** then with **projection** it will show only the first element of **genres** array

**db.infos.find({genres: {$all : ["Drama","Action"]}},{"genres.$": 1})** -> It will first search for the documents with **genres array** with **Drama and Action** then with **projection** it will show only the first element of **genres** array

**db.infos.find({genres: {$all : ["Drama","Action"]}},{"genres" : {$elemMatch: {$eq: "Horror"}}})** -> Here **Querying** and **projecting** works independently. First it will search for **genres array** with **Drama and Action** then with **projection** it will show only the array with **Horror** present or not.

**db.infos.find({},{genres: {$slice: 2},name: 1})** -> **$slice** only works array while projection. **{$slice: 2}** will slice the first **2** elements of the array.

**db.infos.find({},{genres: {$slice: [1,3]},name: 1}) -> {$slice: [1,3]}** will slice the 1st to 3rd elements of the array.

Since shell is made of **JS** so we can use **JS** function

For reference: <https://www.mongodb.com/docs/manual/reference/method/js-cursor/>

**Update**

In the Users db Info collection all the documents in the following type

{

"\_id": {

"$oid": "62ac4ff719cc703713ba43c0"

},

"name": "Max",

"hobbies": [

{

"title": "Sports",

"frequency": 3

},

{

"title": "Cooking",

"frequency": 6

}

],

"phone": 131782734

}

But the object with name chris has the different type of hobbies array

{

"\_id": {

"$oid": "62ac4ff719cc703713ba43be"

},

"name": "Chris",

"hobbies": [

"Sports",

"Cooking",

"Hiking"

]

}

So, we need to update the hobbies. We have two methods for update any object updateOne and updateMany. The names are self-explanatory.

The update method takes two mandatory input one is filter for search and what to update.

“$set” keyword is used to set the change the field value. Other fields will be untouched.

**> db.infos.updateOne({"name" : "Chris"},{ $set: {"hobbies": [{title:"Sports",frequency:5},{title:"Cooking",frequency:3},{title:"Hiking",frequency:1}]}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

If I again run the same query then the modifiedCount will be 0

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }

**> db.infos.updateMany({"hobbies.title":"Sports"},{$set:{isSporty:true}})**

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

With $set operator we can change more than one field at a time as well.

We also have incrementor or decrement operator as these two are very common operation.

**> db.infos.updateOne({name: "Manuel"}, { $inc : { age : 1 }})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

With $inc we can also increment or decrement.

**> db.infos.updateOne({ name: "Manuel" } , { $inc : { age : -1 } , $set : { isSporty : false }})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

We can operate on different field on the same time, but we cannot set age and increment age at the same time.

We have other 3 operators like $inc. those are $min, $max, $mul.

**db.infos.updateOne({ name : "Manual" },{ $max :{ age : 31 }})**

The field age will be updated as the max value between what we have passed and what is the previous value. If the previous value is 20 then the new value of age will be 31 but if its already 35 then the value will not be changed. $min and $max is quite similar operation just that one is taking minimum value and another is taking maximum value.

The $mul operator will multiply the field with the valuw that we have passed.

**db.infos.updateOne({ name : "Manual" },{ $mul :{ age : 1.1 }})**

If the previous value of age is 30 then the new value will be 30\*1.1 = 33

We can also drop any field with **$unset** operator.

**db.infos.updateOne({ name : "Manuel" },{ $unset :{ phone : "" }})**

With this command we can remove the phone field. The value of phone here will be ignored. We can assign any value.

We also can rename the field using the $rename operator.

**> db.infos.updateMany({},{ $rename :{ age : "totalAge" }})**

{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 2 }

The field age will now be converted to totalAge

We can do update and insert operation at the same time, and it is called upsert. Suppose we don’t know we have a document with name as Abhishek or not and we also want to change its value if it is there. So, we can use upsert here.

**> db.infos.updateOne({ name : "Abhishek" } , { $set : { "hobbies": [{ "title": "Sports", "frequency": 3 },{ "title": "Cooking", "frequency": 6 }], "phone": 131782734, "isSporty": true }} , { upsert : true })**

{

"acknowledged" : true,

"matchedCount" : 0,

"modifiedCount" : 0,

"upsertedId" : ObjectId("62da1c2f60336bad54ef7227")

}

So, to use upsert we must pass the upsert value in the last parameter. By default, its value is true.

MongoDB is smart enough to determine that if we are querying with equality operator then the name value must be there. So, in the new object name, hobbies, phone, isSporty all these values will be present.

**Array update operations:**

Suppose we have to find the documents where hobbies array has title value of Sports and frequency value greater than 3. Then the query will be like.

**> db.infos.find({ hobbies : { $elemMatch : { title : "Sports" , "frequency" : { $gte : 3}}}})**

{ "\_id" : ObjectId("62ac4ff719cc703713ba43be"), "name" : "Chris", "hobbies" : [ **{ "title" : "Sports", "frequency" : 5 },** { "title" : "Cooking", "frequency" : 3 }, { "title" : "Hiking", "frequency" : 1 } ], "isSporty" : true }

{ "\_id" : ObjectId("62ac4ff719cc703713ba43c0"), "name" : "Max", "hobbies" : [ **{ "title" : "Sports", "frequency" : 3 },** { "title" : "Cooking", "frequency" : 6 } ], "phone" : 131782734, "isSporty" : true }

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ **{ "title" : "Sports", "frequency" : 3 },** { "title" : "Cooking", "frequency" : 6 } ], "isSporty" : true, "phone" : 131782734 }

Suppose we want to update the inner array document that we have found and highlighted above.

**> db.infos.updateMany({ hobbies : { $elemMatch : { title : "Sports" , "frequency" : { $gte : 3}}}}, { $set : { "hobbies.$.highFrequency" : true }})**

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

This $ represents the same element. Here we are adding new field that is why we are using **"hobbies.$.highFrequency"** but if we want to override that document then we can simply do this **"hobbies.$" : {"title" : "Sports", "frequency" : 5}**

Suppose we want to update all the documents of the array. So, we can use **$[]** operator it means all the array documents.

**goodHobby** field added for all the inner array documents.

**> db.infos.updateMany({ hobbies : { $elemMatch : { title : "Sports" , "frequency" : { $gte : 3}}}}, { $set : { "hobbies.$[].goodHobby" : true }})**

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> **db.infos.find({ hobbies : { $elemMatch : { title : "Sports" , "frequency" : { $gte : 3}}}}).pretty()**

{

"\_id" : ObjectId("62ac4ff719cc703713ba43be"),

"name" : "Chris",

"hobbies" : [

{

"title" : "Sports",

"frequency" : 5,

"highFrequency" : true,

**"goodHobby" : true**

},

{

"title" : "Cooking",

"frequency" : 3,

**"goodHobby" : true**

},

{

"title" : "Hiking",

"frequency" : 1,

**"goodHobby" : true**

}

],

"isSporty" : true

}

If we have a criterion to upadate only some certain documents then we can **$[ el ]** and later we will define the **el** condition in the thir parameter **arrayFilers** part.

**> db.infos.updateOne({ name: "Abhishek"}, { $set : { "hobbies.$[el].goodFrequency" : true}} , { arrayFilters : [{ "el.frequency" :{ $gte : 3}} ]} )**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, **"goodFrequency" : true** }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, **"goodFrequency" : true** } ], "isSporty" : true, "phone" : 131782734 }

In array filter we can pass as many conditions as we want.

We can also add new element in our array. With **$push** we can ad new element to the existing array.

**> db.infos.updateOne({ name: "Abhishek"}, { $push : { hobbies : { title: "Hiking" , frequency : 1}}} )**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true }, **{ "title" : "Hiking", "frequency" : 1 }** ], "isSporty" : true, "phone" : 131782734 }

We can also add more than one documents with **$each** operator.

**> db.infos.updateOne({ name: "Abhishek"}, { $push : { hobbies : { $each : [{ title: "Hiking" , frequency : 1},{title : "wine", frequecy: 1}]}}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Hiking", "frequency" : 1 }, **{ "title" : "Hiking", "frequency" : 1 }, { "title" : "wine", "frequecy" : 1 }** ], "isSporty" : true, "phone" : 131782734 }

We can also add sort or slice operator to add the element in sorted order or we can also take only one element.

But there is issue with **$push** operator. If the values are already existing, then also it will add the value. We can use $addToSet operator for to add unique element in the array.

**> db.infos.updateOne({ name: "Abhishek"}, { $addToSet : { hobbies : { $each : [{ title: "Hiking" , frequency : 1},{title : "wine", frequecy: 1}]}}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Hiking", "frequency" : 1 }, { "title" : "Hiking", "frequency" : 1 }, { "title" : "wine", "frequecy" : 1 } ], "isSporty" : true, "phone" : 131782734 }

We can also pull the element from an array with **$pull** operator.

**> db.infos.updateOne({ name: "Abhishek"}, { $pull : { hobbies : { title : "Hiking" }}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true }, { "title" : "wine", "frequecy" : 1 } ], "isSporty" : true, "phone" : 131782734 }

**{ $pull : { hobbies : { title : "Hiking" }}}** it means pull from the hobbies array where title is Hiking. We can also add other queries.

If we want to remove the last element from the array, then we can use $pop operator with value of 1 and if we want to remove the first element then we can assign the value with -1.

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true }, { "title" : "wine", "frequecy" : 1 } ], "isSporty" : true, "phone" : 131782734 }

**> db.infos.updateOne({ name: "Abhishek"}, { $pop : { hobbies : 1}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Sports", "frequency" : 3, "highFrequency" : true, "goodHobby" : true, "goodFrequency" : true }, { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true } ], "isSporty" : true, "phone" : 131782734 }

**> db.infos.updateOne({ name: "Abhishek"}, { $pop : { hobbies : -1}})**

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

**> db.infos.find({name:"Abhishek"})**

{ "\_id" : ObjectId("62da1c2f60336bad54ef7227"), "name" : "Abhishek", "hobbies" : [ { "title" : "Cooking", "frequency" : 6, "goodHobby" : true, "goodFrequency" : true } ], "isSporty" : true, "phone" : 131782734 }

**Delete**

To delete any document, we have deleteOne and deleteMany methods.

**db.infos.deleteOne({ name : “Abhishek” })**

In the 2nd parameter we can also give writeConcerns.

**db.infos.deleteMany({ age : { $gte : 40} })**

**db.infos.deleteMany({ age : { $exists : false } })**

We can delete all the documents in a collection by using **db.infos.deleteMany({})**

We can drop the collection by **db.infos.drop()**

We can drop the database by **db.dropDatabase()**

**Indexes**

**Why Indexes?**

An index can speed up our find update and delete query. If our query is like **db.products.find({ seller : “Max” })** then MongoDB will search for the entire collection for the seller name “Max”, which is also called as **COLLSCAN** and this can take a while if there is million record. So, in that case we can create a Index on Seller field. MongoDB will create an Ordered list with all the values of the Sellers and all the items of this list will have a pointer to the actual document in the collection. Now if we run the exact query then Mongodb will see that there is an Index on Seller so MongoDB will run **IXSCAN** and directly jump to “M” which will speed up the querying. But we should not overdo the indexes. If we can index on all fields, then it will certainly improve the performance for the find query but for the insert query it will slow down. As now it will again have to update the Ordered list for every field index for every insert and update.

To see the all the index present on the collection:

**> db.infos.getIndexes()**

[ { "v" : 2, "key" : { "\_id" : 1 }, "name" : "\_id\_" } ]

By default, mongodb will create an index on **\_id** field.

To create an index on specific fields:

**db.infos.createIndex( { “dob.age” : 1} } )**

**db.infos.createIndex( { “dob.age” : -1} } )**

1 means increasing and -1 means decreasing Though that does not matter as mongoDB can traverse both ways. We can also create index with more than on field. The order matters here.

**db.infos.createIndex( { “email” : 1, “dob.age” : 1} } )**

This means that mongoDb will create a compound index and first the index with email then dob.age

Example: ([a@test.com,23](mailto:a@test.com,23)) will come before ([a.@test.com,24](mailto:a.@test.com,24))

To drop the index use:

**> db.infos.dropIndex({ "dob.age": 1 })**

{ "nIndexesWas" : 2, "ok" : 1 }

We can also drop index by name.

**> db.infos.dropIndex("dob.age\_1")**

{ "nIndexesWas" : 2, "ok" : 1 }

To analyse how a query will execute mongodb has a unique method that is **explain().**

**> db.infos.explain().find( { "dob.age" : { $gt : 60 }} )**

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

"parsedQuery" : {

"dob.age" : {

"$gt" : 60

}

},

"queryHash" : "FC9E47D2",

"planCacheKey" : "A5FF588D",

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

**"winningPlan" : {**

**"stage" : "COLLSCAN",**

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

"direction" : "forward"

},

**"rejectedPlans" : [ ]**

},

"command" : {

"find" : "infos",

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

In the winning plan we can see **COLLSCAN** as mongodb searched the entire collection for this query.

There is also a rejected plans array but currently it is empty as mongodb has no other option than searching the entire array.

We can also add additional properties in explain(). It will print some additional information

**> db.infos.explain("executionStats").find( { "dob.age" : { $gt : 60 }} )**

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

"parsedQuery" : {

"dob.age" : {

"$gt" : 60

}

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

**"winningPlan" : {**

**"stage" : "COLLSCAN",**

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

**"nReturned" : 1222,**

**"executionTimeMillis" : 3,**

**"totalKeysExamined" : 0,**

**"totalDocsExamined" : 5000,**

**"executionStages" : {**

**"stage" : "COLLSCAN",**

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

**"nReturned" : 1222,**

"executionTimeMillisEstimate" : 0,

"works" : 5002,

"advanced" : 1222,

"needTime" : 3779,

"needYield" : 0,

"saveState" : 5,

"restoreState" : 5,

"isEOF" : 1,

"direction" : "forward",

"docsExamined" : 5000

}

},

"command" : {

"find" : "infos",

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

Here we can see some other additional informations like totalDocumentScan, totalDocumentReturn, executionTimeMillis.

Now if we do the indexing on dob.age and run the same query with explain

**> db.infos.createIndex( { "dob.age" : 1} )**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

**> db.infos.explain("executionStats").find( { "dob.age" : { $gt : 60 }} )**

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

"parsedQuery" : {

"dob.age" : {

"$gt" : 60

}

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

**"winningPlan" : {**

**"stage" : "FETCH",**

**"inputStage" : {**

**"stage" : "IXSCAN",**

"keyPattern" : {

"dob.age" : 1

},

"indexName" : "dob.age\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"dob.age" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"(60.0, inf.0]"

]

}

}

},

"rejectedPlans" : [ ]

},

**"executionStats" : {**

**"executionSuccess" : true,**

**"nReturned" : 1222,**

**"executionTimeMillis" : 50,**

**"totalKeysExamined" : 1222,**

**"totalDocsExamined" : 1222,**

**"executionStages" : {**

**"stage" : "FETCH",**

**"nReturned" : 1222,**

"executionTimeMillisEstimate" : 0,

"works" : 1223,

"advanced" : 1222,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"docsExamined" : 1222,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 1222,

"executionTimeMillisEstimate" : 0,

"works" : 1223,

"advanced" : 1222,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"keyPattern" : {

"dob.age" : 1

},

"indexName" : "dob.age\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"dob.age" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"(60.0, inf.0]"

]

},

"keysExamined" : 1222,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0

}

}

},

"command" : {

"find" : "infos",

"filter" : {

"dob.age" : {

"$gt" : 60

}

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

Now the query did not search for the entire collection it has done an **IXSCAN**

**Indexes Behind the Scenes**

What does createIndex() do in detail?

Whilst we can't really see the index, you can think of the index as a simple list of values + pointers to the original document.

Something like this (for the "age" field):

(29, "address in memory/ collection a1")

(30, "address in memory/ collection a2")

(33, "address in memory/ collection a3")

The documents in the collection would be at the "addresses" a1, a2 and a3. The order does not have to match the order in the index (and most likely, it indeed won't).

The important thing is that the index items are ordered (ascending or descending - depending on how you created the index). createIndex({age: 1}) creates an index with ascending sorting, createIndex({age: -1}) creates one with descending sorting.

MongoDB is now able to quickly find a fitting document when you filter for its age as it has a sorted list. Sorted lists are way quicker to search because you can skip entire ranges (and don't have to look at every single document).

Additionally, sorting (via sort(...)) will also be sped up because you already have a sorted list. Of course, this is only true when sorting for the age.

Let’s say all our document has age greater than 50 and in query [ **db.infos. find( { "dob.age" : { $gt : 20 }} )]** we are trying to find the documents greater than 20 so it will return all the documents. So, in this case IXSCAN has the less performance as it will introduce an extra step. As at first the mongodb will scan the entire index then it will go to the actual mongodb collection. If we delete the index, then it will again search with **COLSCAN** and eventually that will have a better performance. So, it is recommended that only to use index when the query will return a small subset of the actual collection.

Index on Boolean value does not make much sense.

Compound index

**> db.infos.createIndex({ "dob.age" : 1, "gender" : 1})**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

If we search with dob.age and gender then mongodb will use this compound index.

> db.infos.explain("executionStats").find({"dob.age" : 35, "gender" : "male"})

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

"parsedQuery" : {

"$and" : [

{

"dob.age" : {

"$eq" : 35

}

},

{

"gender" : {

"$eq" : "male"

}

}

]

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

**"stage" : "IXSCAN",**

**"keyPattern" : {**

**"dob.age" : 1,**

**"gender" : 1**

**},**

**"indexName" : "dob.age\_1\_gender\_1",**

**"isMultiKey" : false,**

"multiKeyPaths" : {

"dob.age" : [ ],

"gender" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"[35.0, 35.0]"

],

"gender" : [

"[\"male\", \"male\"]"

]

}

}

},

"rejectedPlans" : [ ]

},

**"executionStats" : {**

**"executionSuccess" : true,**

**"nReturned" : 43,**

**"executionTimeMillis" : 19,**

**"totalKeysExamined" : 43,**

**"totalDocsExamined" : 43,**

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 43,

"executionTimeMillisEstimate" : 11,

"works" : 44,

"advanced" : 43,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"docsExamined" : 43,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 43,

"executionTimeMillisEstimate" : 11,

"works" : 44,

"advanced" : 43,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"keyPattern" : {

"dob.age" : 1,

"gender" : 1

},

**"indexName" : "dob.age\_1\_gender\_1",**

"isMultiKey" : false,

"multiKeyPaths" : {

"dob.age" : [ ],

"gender" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"[35.0, 35.0]"

],

"gender" : [

"[\"male\", \"male\"]"

]

},

"keysExamined" : 43,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0

}

}

},

"command" : {

"find" : "infos",

"filter" : {

"dob.age" : 35,

"gender" : "male"

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

If we just look for the age, then also mongodb will use this index as “dob.age” comes first in the index order.

> db.infos.explain("executionStats").find({"dob.age" : 35})

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

**"parsedQuery" : {**

**"dob.age" : {**

**"$eq" : 35**

**}**

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

**"stage" : "IXSCAN",**

"keyPattern" : {

"dob.age" : 1,

"gender" : 1

},

"indexName" : "dob.age\_1\_gender\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"dob.age" : [ ],

"gender" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"[35.0, 35.0]"

],

"gender" : [

"[MinKey, MaxKey]"

]

}

}

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

**"nReturned" : 95,**

**"executionTimeMillis" : 0,**

**"totalKeysExamined" : 95,**

**"totalDocsExamined" : 95,**

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 95,

"executionTimeMillisEstimate" : 0,

"works" : 96,

"advanced" : 95,

"needTime" : 0,

"needYield" : 0,

"saveState" : 0,

"restoreState" : 0,

"isEOF" : 1,

"docsExamined" : 95,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 95,

"executionTimeMillisEstimate" : 0,

"works" : 96,

"advanced" : 95,

"needTime" : 0,

"needYield" : 0,

"saveState" : 0,

"restoreState" : 0,

"isEOF" : 1,

"keyPattern" : {

"dob.age" : 1,

"gender" : 1

},

**"indexName" : "dob.age\_1\_gender\_1",**

"isMultiKey" : false,

"multiKeyPaths" : {

"dob.age" : [ ],

"gender" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"dob.age" : [

"[35.0, 35.0]"

],

"gender" : [

"[MinKey, MaxKey]"

]

},

"keysExamined" : 95,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0

}

}

},

"command" : {

"find" : "infos",

"filter" : {

"dob.age" : 35

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

But if we only search will the gender then index has no use because gender is not sorted primarily. It is secondary sort on the dob.age. Here mongodb will use the full **COLLSCAN**.

> db.infos.explain("executionStats").find({"gender" : "male"})

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

**"parsedQuery" : {**

**"gender" : {**

**"$eq" : "male"**

**}**

**},**

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

"winningPlan" : {

"stage" : "COLLSCAN",

"filter" : {

"gender" : {

"$eq" : "male"

}

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"executionStats" : {

**"executionSuccess" : true,**

**"nReturned" : 2435,**

**"executionTimeMillis" : 4,**

**"totalKeysExamined" : 0,**

**"totalDocsExamined" : 5000,**

**"executionStages" : {**

**"stage" : "COLLSCAN",**

"filter" : {

"gender" : {

"$eq" : "male"

}

},

"nReturned" : 2435,

"executionTimeMillisEstimate" : 0,

"works" : 5002,

"advanced" : 2435,

"needTime" : 2566,

"needYield" : 0,

"saveState" : 5,

"restoreState" : 5,

"isEOF" : 1,

"direction" : "forward",

"docsExamined" : 5000

}

},

"command" : {

"find" : "infos",

"filter" : {

"gender" : "male"

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

**Sorting with indexing:**

If we are sorting on any field and that field has an indexing, then mongodb will not sort it will directly use the indexed records as mongodb already has a sorted list on that field.

If we are trying to sort on a large number of documents, then it will time out. MongoDB has a memory of 32 megabytes of memory of sorting. By default, mongodb loads all the documents on its memory then it sorts on them. So, without indexing sometimes it is not possible to get the sorted documents.

When we are creating any index on that time, we can specify that the index will be unique or not. By default, the indexing on $id holds unique criteria.

**> db.infos.createIndex({ email : 1 }, { unique : true })**

Before creating index if there is already any duplicate email available then it will throw an error.

**> db.infos.createIndex({ email : 1 }, { unique : true })**

{

"ok" : 0,

**"errmsg" : "Index build failed: 8aff9b57-7fce-4ff9-8631-4f22c63ddaff: Collection persons.infos ( c6d8709f-2a51-4bda-ac9e-343a639304d6 ) :: caused by :: E11000 duplicate key error collection: persons.infos index: email\_1 dup key: { email: \"abigail.clark@example.com\" }",**

"code" : 11000,

"codeName" : "DuplicateKey",

"keyPattern" : {

"email" : 1

},

"keyValue" : {

"email" : "abigail.clark@example.com"

}

}

**Partial filter/Indexing**

We can always use compound indexing but the problem with the compound indexing is that it takes much space in discs. So, in that case we can use partial filter like if we know that gender male is frequently queried rather than female. So, we can create a partial index with gender male.

**> db.infos.createIndex({"dob.age" : 1}, {partialFilterExpression : {"gender" : 1}} )**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

**> db.infos.getIndexes()**

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"key" : {

"dob.age" : 1

},

"name" : "dob.age\_1",

"partialFilterExpression" : {

"gender" : 1

}

}

]

Drawback of this partial filter is that now when we just query for the “dob.age” it will not use **IXSCAN** it will use the **COLLSCAN**. But if we also mention gender male then it will use the **IXSCAN.**

Advantage of partial filter is that now the write query is more efficient as the size of the ordered list is small.

If we have a index on email and unique true and if we enter document without email then mongodb will treat that document as email equal to null. Again, if we try to insert any document without email then mongoDB will throw an exception as email null is already stored in ordered list. We can not add null value again.

To allow this condition we can use unique true with partial filter expression.

**> db.infos.createIndex({"dob.age" : 1}, {unique : true, partialFilterExpression : {"email" : {exists : true}}} )**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

**Time to live index**

It is only applicable for date or timestamp. With this indexing after certain time the document will automatically be deleted.

If there is already some document and then we are adding this index, then at the time of index creation it will not check the existing documents. When we insert any new data then it will evaluate all the documents again and then it will use **TTL** index.

**> db.sessions.createIndex({ createdAt : 1} , {expireAfterSeconds : 10})**

**> db.sessions.insertOne({data : "I am Abhishek", createdAt : new Date()})**

{

"acknowledged" : true,

"insertedId" : ObjectId("62da72b385a6e4bfe5a374cb")

}

**> db.sessions.findOne()**

{

"\_id" : ObjectId("62da72b385a6e4bfe5a374cb"),

"data" : "I am Abhishek",

"createdAt" : ISODate("2022-07-22T09:49:39.459Z")

}

**> db.sessions.createIndex({ createdAt : 1} , {expireAfterSeconds : 10})**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

Now with this index the documents will be delete after 10 seconds.

This can be useful for session or carts in online shopping where the cart item automatically deletes after on day.

Query Diagnosis and & Query Planning

**explain()** method takes three type of string.

1. “**queryPlanner”:** Show summary for executed query and winning plan
2. **“executionStats”:** Show detailed summary for executed query and winning plan and rejected plans.
3. **“allPlansExecution”:** Show detailed summary for executed query and winning plan and winning plan decision process.

For determining the query is efficient or not we must check following things:

Processing time in milliseconds, no of keys examined (if index scan happened), No of documents examined, no of documents returned.

The keys and documents examined should be close together and documents examined and returned should be closed or documents should be zero so that it looked at zero documents. In a so-called covered query, it will be happening.

**Covered query:**

If we have an indexing on name and we are only querying for name on that time mongodb will not even look to the documents, instead it will directly return the name from the indexed ordered list.

Example of this type of query is like:

**db.infos.findOne({ “name” : “Abhishek”}, { \_id: 0, name : 1})**

Suppose we have an index on name and another index one age and name (the ordering is important here). Now if we search for any document with name and age then mongodb will use the compound index, it will use the single index on name. If we do an **explain(“executionStats”)** then age\_1\_name\_1 will fall under winning plan and name\_1 will fall under rejected plans.

To find the winning plan mongodb check the query and available index then it will choose among them. So every time there is a query mongodb tries to find a winning plan, but again it will be having the extra step to find among all the plans. So mongodb save the winning plan in the caches for the query. This cache is not for forever. Mongodb resets the cache after db restarts, after few inserts or there is any rebuilt of index or changes in index.

**Multikey index:**

We can also create indexes on array values. Lets say we are adding one document like this.

**> db.infos.insertOne({"name" : "Abhishek", "gender" : "male", "hobbies" : ["Sports", "Coding"]})**

{

"acknowledged" : true,

"insertedId" : ObjectId("62dcaf7185a6e4bfe5a374cc")

}

**> db.infos.createIndex({hobbies: 1})**

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

> db.infos.explain().find({hobbies : "Coding"})

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "persons.infos",

"indexFilterSet" : false,

"parsedQuery" : {

"hobbies" : {

"$eq" : "Coding"

}

},

"queryHash" : "895C9692",

"planCacheKey" : "439794C9",

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

**"winningPlan" : {**

**"stage" : "FETCH",**

**"inputStage" : {**

**"stage" : "IXSCAN",**

**"keyPattern" : {**

**"hobbies" : 1**

**},**

**"indexName" : "hobbies\_1",**

**"isMultiKey" : true,**

"multiKeyPaths" : {

"hobbies" : [

"hobbies"

]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"hobbies" : [

"[\"Coding\", \"Coding\"]"

]

}

}

},

"rejectedPlans" : [ ]

},

"command" : {

"find" : "infos",

"filter" : {

"hobbies" : "Coding"

},

"$db" : "persons"

},

"serverInfo" : {

"host" : "LAPTOP-V7ATCH7F",

"port" : 27017,

"version" : "5.0.6",

"gitVersion" : "212a8dbb47f07427dae194a9c75baec1d81d9259"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

Here multikey is true.

When we are creating index on array values. On that time there will be a ordered list with all the elements with array. It polls out all the elements of the array and stores as a separate element. So, it is larger than the size of the document.

If the array consists of documents, then we have to query with that document otherwise it will not use **IXSCAN**. Suppose we have address array with homeAddress and we are creating index on arrays.

**{“address”: [{“homeAddress” : “18 No alep khan mahalla road”}, {“homeAddress” : “Rameswara waterview block 1,4B”}]}**

Here we have to search like this:

**db.infos.find({“address”: {“homeAddress” : “18 No alep khan mahalla road”}})**

Otherwise indexing will not work. We can also use on **“address.homeAddress”.**

We can create compound indexes with multikey index like with **name and address array**.

It will do a cartesian product of the name and address values. Then it will store in the ordered list.

But we can not create a compound index if both values are array.

**Text index:**

If we search using regex that is very low in performance rather, we can use text indexes.

Text string is just an array of words. So, mongodb stores the main keywords and removes the stop words like **“is”, “the”, “a” etc.**

The main thing with text index it we can only create on index, which is type of **text**, because it is expensive to store all the keywords. If we have any criteria to use on both rather, we can create compound index of type text with the two fields.

**db.infos.createIndex({ “description” : “text” })**

We can not specify 1 or -1 while creating the index.

We can search like this.

**db.infos.find({ “$text” : { “$search” : “pretty” }})**

We cannot user regular queries. Text index is expensive, and we have use it like this.

**db.infos.find({ “$text” : { “$search” : “red book” }})**

If we search for “red book” then the upper query will not work as it will split the query string into multiple word, then it will search individually like it will search for red and it will search for book then it will combine the result. So, we have to use quotation mark around our query if we are searching for phrases.

**db.infos.find({ “$text” : { “$search” : “\”red book\”” }})**

If we have more than one result, then behind the scenes mongodb assigns meta score to the documents. Higher the score means that the document matches with our query better. To see the score with have to project the score as well.

**db.infos.find({ $text : { $search : “awesome book” }} , { score : { $meta : “textScore” }})**

we can also show the results with sorted based on scores.

**db.infos.find({ $text : { $search : “awesome book” }} , { score : { $meta : “textScore” }}).sort({ score : { $meta : “textScore” }})**

It will be a decreasing type of sorting.

We can use more than field for text index. **To drop an index of text, we must drop by name.**

**db.infos.createIndex({ title : “text” , description : text })**

it will create an index using the keywords of both fields. We can search like previous. Case does not matter for text index.

**db.infos.find({ “$text” : { “$search” : “pretty” }})**

We can also rule out for the specific words.

**db.infos.find({ “$text” : { “$search” : “pretty -books” }})**

we have to add minus (-) before that word. It will for pretty word where book word is not present.

We can also use language in text index as stop words for different language is different. Default language is English though. There is list of supported language that we can use. Default language is very important when it comes to text index.

**db.infos.createIndex({ title : “text” }, { default\_laguage : “germany”})**

We can also assign weight to the fields which will be used to create text index.

**db.infos.createIndex({ title : “text” , summary : “text” }, { weights : { title : 5 , summary : 1 })**

We also search in case sensitive way like the following.

**db.infos.find({ “$text” : { “$search” : “pretty” } , $caseSentitive : true })**

**Building Index:**

When we are creating any index using createIndex method on that time the collection got locked. On that time if we try to insert any document then we have to wait for a certain time. The down time will depend on the size of the collection. It is adjustable in lower environment, but we cannot afford this in production. To deal with this create index in **background**. The time taken for creating the index is slow in background than foreground.

**db.infos.createIndex({ “age” : 1 } , { “background” : true })**

**Geospatial Data**

We can also store and retrieve geo location (2D) data and use indexes on that. It will be stored as [x,y] where x must be the longitude and y must be the latitude. It follows the geoJSON format only.

**> db.infos.insertOne({name : "Home" , location : { type : "Point" , coordinates : [24.0814946,88.2408234,13.38]}})**

{

"acknowledged" : true,

"insertedId" : ObjectId("62dcff4c85a6e4bfe5a374cd")

}

To store the coordinates, we must follow this structure of the embed document { **location : { type : "Point" , coordinates : [24.0814946,88.2408234]}}.**

We can change the name of field “location”, but the structure must be same.

We can also store area or polygon

Let’s create 4 points

**const p1 = [24.08409, 88.24231]**

**const p2 = [24.09149, 88.24707]**

**const p3 = [24.08879, 88.25578]**

**const p4 = [24.08048, 88.24934]**

**> db.infos.insertOne({name : "Gorabazar area" , location : { type : "Polygon" , coordinates : [[p1,p2,p3,p4,p1]]}})**

{

"acknowledged" : true,

"insertedId" : ObjectId("62dd0e1d85a6e4bfe5a374d3")

}

It is to better to create a geospatial index as most of the geospatial queries require indexing.

We can check any points are near to the queried point or not. For that we have a special syntax.

**> db.infos.find({ location : { $near : { $geometry : { type : "Point", coordinates : [24,88]}}}}).pretty()**

{

"\_id" : ObjectId("62dcff4c85a6e4bfe5a374cd"),

"name" : "Home",

"location" : {

"type" : "Point",

"coordinates" : [

24.0814946,

88.2408234

]

}

}

We can also specify other things along side $geometry like $maxDistance and $minDistance. The unit will be in meters.

**> db.infos.find({ location : { $near : { $geometry : { type : "Point", coordinates : [24,88]}, $minDistance : 10, $maxDistance : 26809}}}).pretty()**

**> db.infos.find({ location : { $near : { $geometry : { type : "Point", coordinates : [24,88]}, $minDistance : 10, $maxDistance : 26810}}}).pretty()**

{

"\_id" : ObjectId("62dcff4c85a6e4bfe5a374cd"),

"name" : "Home",

"location" : {

"type" : "Point",

"coordinates" : [

24.0814946,

88.2408234

]

}

}

To find a place inside any special region or not we can do this. First, we can create our own map (from google maps -> your places -> see all your maps) and create one area or polygon.

We can validate all the points are inside of these 4 coordinates or not. In map we will see these 4 points made a rectangle.

We will also insert some of the points.

**> db.infos.insertOne({name : "Murshidabad medical college" , location : { type : "Point" , coordinates : [24.089473,88.2513618]}})**

**> db.infos.insertOne({name : "Gorabazar ICI" , location : { type : "Point" , coordinates : [24.0930413,88.2483631]}})**

**> db.infos.insertOne({name : "Mary immaculate school" , location : { type : "Point" , coordinates : [24.0930413,88.2483631]}})**

**> db.infos.insertOne({name : "Berhampore head post office" , location : { type : "Point" , coordinates : [24.0947532,88.2510873]}})**

**> db.infos.insertOne({name : "Mohon cinema hall" , location : { type : "Point" , coordinates : [24.0947532,88.2510873]}})**

Again, we must follow some specific syntax for with in query.

**> db.infos.find({ location : { $geoWithin : { $geometry : { type : "Polygon", coordinates : [[p1,p2,p3,p4,p1]]}}}})**

Keyword is $geoWithin and type is Polygon and coordinates will be in 2nd layer of nested arrays and the first and the last point should be same.

We can also search for the opposite query. We can find an poly where a point belongs or not.

**> db.infos.find({ location : { $geoIntersects : { $geometry : { type : "Point" , coordinates : [24.089473,88.2513618] } }}})**

We can also search in circle within a radius.

**> db.infos.find({ location : { $geoWithin : { $centerSphere : [[24.089473, 88.2513618], 1/6378.1]}}})**

Where 1st one is the 2d coordinate and 2nd one is radius. 1 is in kilometre. 6378.1 is the constant. Check this on official documentation.

**Aggregation Framework**

Aggregation framework is just another find method we could say but it has some other advantages too. In aggregation framework we basically create pipeline of steps which operates on datas of that collection.