**Mongo** is a documented oriented db that stores data in collections that contains documents. It is schema-less structure that doesn’t require any pre defined schema.

**Document** is a JSON like structure that stores and represents data. It contains key-value pairs , arrays, and nested documents.

**Collection**  is a grouping of documents. Collections are schema less means documents in same collections may have different schema/structures.

**Primary-Key** the **\_id** field serves as primary key. It uniquely identifies each document in a collection. It must be unique within a collection; it is automatically generated if not provided during document insertion.

**Sharding** is a strategy used to distribute data across numerous servers and clusters.

**Indexes-**  MongoDB employs data structures known as indexes to enhance query performance . it includes single-field index, compound indexes, geospatial indexes, text indexes, etc.

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.

**Creating DB-**  use DBName for creating using mongo shell.

**Data insertion-**  by using insertOne() or insertMany() methods.

**Replica-set-**  Group of servers that maintains same data. It provides **data redundancy and high availability**. One server acts as a primary server while others acts as secondary, that replicates data from primary server.

Replica sets guarantees high availability. If primary node fails, an automatic process elects one of the secondary node to take over the primary

**Update document -**  updateOne(), updateMany() or findOneAndUpdate() methods.

**Delete document -**  deleteOne(), deleteMany(), and findOneAndDelete() methods.

**Cursor** is an iterator to retrieve and process documents from query results. It is used when fetching large datasets, and allow to retrieve data in batches.

**Data Modelling** involves designing the structure of the documents and collection to represent data in a best manner.

**Perform Query -**  using find() method.

**Aggregation** is a powerful tool designing for processing and transforming data within a collection. It consists stages each transforming and process data before passing it to next stage. We can execute various operations such as grouping ,sorting etc.

**See usage in the aggregation usage doc**

**Backup** using mongoDump().

**Default behaviour of rollback** –

db.hobbies.insertMany([{\_id: “sports”, name: “Sports”}, {\_id: “cooking”, name: “Cooking”}])

Suppose if we insert a new document with some same ids and some different ids it will insert the different ids that comes before the older ids . EX-

db.hobbies.insertMany([{\_id: “yoga”, name: “Yoga”}, {\_id: “sports”, name: “Sports”}, {\_id: “cooking”, name: “Cooking”}, {\_id: “hiking”, name: “hiking”])

In the above code the id with **[\_id: yoga]** will be inserted and after that all the duplicate ids will throw error and the new id with [**\_id: hiking**] will not be inserted. This is called ordered insertion. To remove this behaviour we can pass an object as ordered = false in the query. For ex-

db.hobbies.insertMany([{\_id: “sports”, name: “Sports”}, {\_id: “cooking”, name: “Cooking”}], {ordered: false})

**Write concern –** this gives higher security that our write will be succeeded . We can enable the journal confirmation so that our writes takes longer because we just not tell the server for them but need the server to wait for the write operation. For ex;

db.persons.insertOne({**name**: "Abhay Kanojiya", **age**: 30}, {**writeConcern**: {w**:** 1, **j**: true, **wtimeout**: 200}})

here w = 1/0 for acknowledge= true/false, j = journal = true/false , wtimeout for waiting or writing according to need either a low network connection , etc.

**Query Selectors**

1 findOne({}), find(), find({name: “ABCD”})

**Operators**

1 **Comparison operator** = like $eq, $ne, $gt, $gte, etc . ex find({runtime: {$ne : 60}})

2 **Logical Operators** = line $or , $nor, $not, $and

Ex – db.movies.find({$or: [{"rating.average": {$lt:5}}, {"rating.average":{$gt:9.3}}]})

db.movies.find({$and: [{"rating.average": {$gt:9}}, {genres: "Drama"}]}).count()

Note- in json file no keys will be same, and if we specify like [genres : “Drama”, genres: “Horror”] in the find query, this will replace the first case with second one if there is no match. So to avoid this we can use $and operator

3 **Element operators** - $exists, $type. Ex – db.users.find({phone: {$exists: true}})

**4 Evaluation Operator -**  like $regex, $expr. Ex –

db.movies.find({**summary**: {$**regex** : /musical/}})

db.sales.find({$expr : {$gt: [“$volume”, “$target”]}})

db.sales.find({$expr: {$gt: [{$cond: {if : {$gte: ["$volume", 123]}, then: {$subtract: ["$volume",30]}, else: "$volume"}}, "$target"]}}).pretty()

**Querying Embedded Documents**

Ex- db.movies.find({“**rating**.**total**.**average**”: {$**gt**: 7.0}})

if we only include the exact items from the list we can do it like

Ex db.movies.find({**genres**: [“Drama”]}). This will only gives the results containing only Drama in genres.

**Projection**

**Finding documents using certain matching/ filtering conditions:**

**db.movies.find({genres: "Drama"}, {genres: {$elemMatch : {$eq : "Horror"}}})**

**OR**

**db.movies.find({"rating.average": {$gt: 9}}, {genres: {$elemMatch : {$eq : "Horror"}}})**

**slice -- db.movies.find({"rating.average": {$gt: 9}}, {genres: {$slice: 2}, name : 1})**

**LookUps**

It is used to merge data from one collection to another collection.

Ex :- db.books.aggregate([

{$**lookup**:{

from : “authors”,

localfield: “authors”,

foreignField: “\_id”,

as: “creators”

}

])

**Update**

Adding a new field to an existing document:

updateOne()- db.persons.updateOne({\_id: ObjectId('661c9e1c91b8aa172d16c9b5')}, {$set:{hobbies: [{title: "singing", frequency: 5 }, {title: "cooking", frequency: 2}, {title: "reading", frequency: 10}]}})

updateMany()-- db.persons.updateMany({"hobbies.title": "playing"}, {$set: {isSporty: true}})

**increment Values- db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$inc:{age: 1}})**

**decrement Values- db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$inc:{age: -1}})**

**we can use $min, $max, $mul for document updation**

**- db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$min:{age: 30}})**

**- db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$max:{age: 32}})**

**- - db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$mul:{age: 1.1}})**

**Removing a field from document -- $unset()**

**db.persons.updateOne({\_id: ObjectId('661c9f4391b8aa172d16c9b8')}, {$unset: {phone : “”} })**

**Renaming a field -- $rename()**

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

**upsert() – It is used to add new values in case if there is no matching field. EX-**

**db.persons.updateOne({name: "Chhaya"}, {$set: {age: 29, phone: 1234560000, hobbies: [{title: "Cooking", frequency: 5}], isSporty: true}}, {upsert: true} )**

**Adding a new field in an array using update()**

**db.persons.updateMany({hobbies: {$elemMatch: {title: "playing"}}}, {$set:{"hobbies.$.highFrequency":true}})**

**updation of array fields using array operators**

**db.persons.updateMany({totalAge: {$gt: 27}},{$inc: {"hobbies.$[].frequency": -2}} )**

**Adding new elements to an existing array.**

**$push()- db.persons.updateOne({name: "Poonam"}, {$push: {hobbies: {title: "Sports", frequency: 1}}})**

**Removing elements to an existing array.**

**$pull()- it delete all the matching fields**

**db.persons.updateOne({name: "Poonam"}, {$pull: {hobbies: {title: "Sports"}}})**

**$pop()- it delete last matching fileds**

**db.persons.updateOne({name: "Chhaya"}, {$pop: {hobbies: 1}})**

**Creating Indexes**

1 **Single field indexes**-

db.contacts.explain().find({"dob.age": {$gt: 60}}) // to check the execution time

db.contacts.explain(“executionTime”).find({"dob.age": {$gt: 60}})

db.contacts.createIndex({"dob.age": 1}) // creating Index

db.contacts.dropIndex({"dob.age": 1}) // dropping Index

**2 Compound Indexes- index on multiple fields.**

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

Note: in compound index data will be sorted from left to write.

**Configuring Indexes**

db.contacts.createIndex({email : 1}, {unique : true}) // here for unique indexes we will get an error if there will be any duplicate data

**Partial Filters – Used to create indexes using according to given filters**

**db.contacts.createIndex({"dob.age":1}, {partialFilterExpression : {gender : "male"}}) // Note – here we have to perform query for the given filters also. Other wise it will perform COLLSCAN not IXSCAN . For ex**

**db.contacts.explain("executionStats").find({"dob.age": 60}) // if we run this query it will do a COLLSCAN because here we did not give the index for gender field**

**db.contacts.explain("executionStats").find({"dob.age": 60, gender : "male"}) // Now this will do an IXSCAN**

**TTL indexex – Time To Live Indexes**

**db.sessions.createIndex({createdAt : 1}, {expireAfterSeconds : 10}) // If we insert another data this will re-evaluates the document and delete all documents after given time**

**Note – TTL can only be used on single field indexes and on date objects**

**Text index- It is used to search according to specified text. For ex;**

**db.products.createIndex({description : “text”}) // Text keyword will specify to perform query according to given text in the query. Like =**

**db.products.find({$text : {$search : “book awesome”}}) // For all the documents containing the matched text.**

**db.products.find({$text : {$search : “\”book awesome\””}}) // only for specific text**

**To exclude text we can do like:**

**db.products.find({$text : {$search : “awesome -t-shirt”}})**

**Geospatial Queries – Finding Places**

Adding **geoJSON**

db.places.insertOne({name: "Pashupati Nath Temple", location : {type : "Point", coordinates : [27.710512, 85.3462376]}})

Geospatial index to track the distances –

db.places.createIndex({location : "2dsphere"})

db.places.find({location : {$near : {$geometry : {type: "Point", coordinates: [27.7078097,85.3427883]}, $maxDistance : 700, $minDistance: 50}}}) // finding near locations

Places inside a certain area –

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

Authentication and Authorization

**1 Creating user - db.createUser({user: "alok", pwd: "alok", roles: ["userAdminAnyDatabase"]})**

**To authenticate the created user - db.auth(“alok”, “alok”)**

**Note –**

**we can auth the user by mongo –u alok –p alok –authenticationDatabase admin**

**2 Assigning roles to users and databases-**

**db.createUser({user: "poonam", pwd: "1234", roles: ["readWrite"]})**

**Capped Collections**

It used to specify the size of collection at the time of collection creation

. When new data comes in and size exceeds older data will be deleted

db.createCollection("capped", {capped: true, size: 10000, max: 3})

Note – If we insert more than 4th document then 1st will be deleted and so on

USsjnOvIdDz6xZu4

**Transactions**

To start transactions first we create sessions

**Sessions creation –** const session = db.getMongo().startSession()

**Start transaction – session.startTransaction()**

**collection to connect the session =**

**const userCol = session.getDatabase(“blog”).users**

**const userCol = session.getDatabase(“blog”).posts**

**Using Atlas from Application**

**Creating collection**: **client.db().collection(‘products’)**

**Inserting document: client.db().collection(‘products’).insertOne(newProduct)**

**Using 128bit decimal :**

const decimal = mongodb.Decimal128;

price: decimal.fromString(req.body.price)

**Adding Pagination**

const queryPage = req.query.page;

  const pageSize = 2;

db.getDb()

    .db()

  .collection('products')

    .find()

    .sort({ price: -1 })

    .skip((queryPage - 1) \* pageSize)

    .limit(pageSize)

…..