**Basics**

1. Schema less and stores data in JSON format.
2. Collections in MongoDB = Tables in Oracle
3. Data stored as key-value pair and we can store nested values.
4. On the backend, MONGODB converts JSON data to BSON for better storage.
5. Show dbs will show all existing DBS.
6. use <db\_name> will create new db.
7. db.products.insertOne or db.products.insert({key:value, key:value}) will allow you to create a new collection name products inside the db and insert new value. ID will be auto generated.
8. This newly inserted record can be viewed using db.products.find().pretty().
9. MongoDB operations are atomic on document level including embedded documents i.e. even if you use insertMany then no individual document will be inserted half, a document will either be inserted completely or fail completely.
10. JSON data can be directly imported using the below syntax  
    ***mongoimport filename.json -d dbname -c collectionname –jsonArray –drop [if collection exists then drop it and recreate. Skip drop if you want to append the data.]***

|  |  |  |  |
| --- | --- | --- | --- |
| Create | Read | Update | Delete |
| 1. insertOne(data, options) 2. insertMany(data, options) 3. If you try and insert   duplicate data then the data up to the point where the duplicate was encountered will be inserted and the rest of the data will be ignored. This is called ordered insert. This behaviour can be changed by passing a second flag  {ordered : false} during the insert or insertMany operation. The default value is true   1. writeConcern is another option we can use with insert which helps us confirm our insert has succeeded. We can also use writeConcern to set journaling to on so that the data is initially written to a **journal file**(to-do list). This saves time and the insert is eventually saved to disk. Another option available is the **wtimeout** which allows us to set a timeout for a write operation. **Syntax** db.insertMany( {name:’Jaydeep’ } { writeConcern{ w:1 (default is 1), j:true, (default is value false or undef), wtimeout: any integer value } } ) | 1. find(filter, options) 2. findOne(filter, options) | 1. updateOne(filter, data, options) 2. updateMany(filter, data, options) 3. replaceOne(filter, data, options) | 1. deleteOne(filter, options) 2. deleteMany(filter, options) |

**Examples Update**

1. Update one record : db.products.updateOne({“price”:12.99},{$set: {“author”:”Jaydeep”})

***Sql equivalent UPDATE PRODUCTS SET AUTHOR = ‘JAYDEEP’ WHERE PRICE = ‘12.99’***

1. Update many records : db.products.updateMany({},{$set: {“author”:”Jaydeep”})

**SQL equivalent UPDATE PRODUCTS SET AUTHOR = “JAYDEEP’.**

*P.S :-*

*1) The only difference here is that if the column AUTHOR doesn’t exist in the collection then in MongoDB it will add that data*

*2) If the $SET is missing then the entire collection will get replaced.*

1. db.products.updateOne({‘\_id’: 5eb2a982fb120db60963eebc}, {“author”:”Jaydeep”})

**Examples Find(select)**

1. db.products.find({‘price’:12.99}) is equivalent to SELECT \* FROM PRODUCTS WHERE PRICE = 12.99
2. db.products.findOne({‘price’:12.99}) is equivalent to SELECT \* FROM PRPODUCTS WHERE PRICE = 12.99 LIMIT 1;
3. find() doesn’t give us the entire collection, it returns a cursor which we can iterate over and find new records.
4. Max level of nesting is 100 and max doc size is 16MB
5. For nested documents find can be applied using the .(dot) notation i.e. find({“status.description.responsible”: “Jaydeep Karale”}). This will search for all records with responsible = Jaydeep Karale in 3rd level document.
6. Database = database, collections = tables and documents = rows.
7. Db.stats() shows db information.

**Data Types**

1. Text
2. Boolean
3. Numbers
   1. Integer (int32)
   2. Number Long(int64)
   3. NumberDecimal
4. ObjectId(uuid)
5. ISODate
6. TimeStamp
7. Embedded Document
8. Arrays

**Relations**

1. Relations can be established either by embedded documents or using reference keys.
2. Use embedded data when there is a strong 1-1 relation for example patient and disease summary or car and person as one person generally owns one car. E.g

*db.patients.insertOne{*

*"name":"Jaydeep Karale",*

*"age" : 30,*

*"diseasesumary":{*

*"diseases" :["appendicitis","acidity isssues"]*

*}*

*}*

1. Use references when there is a strong 1-M relationship.
2. Use db.collection.aggregate($lookup: {from: ‘’, localField:’’, foreignFied:’’, aliasname:’’}) to retrieve joined data.

**Schema Validation**

1. Used to ensure integrity of data that is being stored in the collection.
2. We can either have strict validation where all inserts and updates are checked, or we can have moderate validation where insert and update on only existing documents is done.
3. Once we apply validation we can either reject the record or log a warning and allow the insert or update anyway.

*Syntax :- 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'*

*}*

*}*

*}*

*}*

*}*

*}*

*}*

*});*

*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'*

*});*

**Read Operations: Detailed**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Explanation** | **SQL Equivalent** |
| $eq equal operator | is same as normal find | = |
| $gt greater than | find({ key : {$gt : value}}) | > |
| $gte greater than equal | find({ key : {$gte : value}}) | >= |
| $ne not equal | find({ key : {$ne : value}}) | <> or != |
| $lt less than | find({ key : {$lt : value}}) | < |
| $lte greater than equal | find({ key : {$lte : value}}) | <= |
| $in in | find({ key : {$in : [ value1, value2, value3]}}) | In () |
| $nin Not IN | find({ key : {$NOT IN : [ value1, value2, value3]}}) | NOT IN() |
| $or and $nor | find({$or : [{ condition1},{ condition 2}]})  find({$nor : [{ condition1},{ condition 2}]}) | OR. NOR doesn’t exist in SQL |
| $and   this is the default concat method. It only exits because certain language drivers do not support multiple conditions separated(shown in second example) by commas and need explicit AND where without AND they just replace second key with first. | find({$and : [{ condition1},{ condition 2}]})  ***THIS IS SAME AS***  find({ condition1},{ condition 2}) | AND |
| $type find documents for which column type matches a certain data type. | find({phonenumber : {$type : “double”}})  find({phonenumber : {$type : Array}}) | Does not exist in SQL |
| $exists returns documents which have a certain field. | find({age :{ $exists : true})  find({age : {$exists: true, $gt : 30}})   this works for NULL values too. Only if you set $exists :FALSE then it won’t return the fields with NULL values.  find({age : {$exists : true, $ne: NULL}}) | Same as EXISTS |
| $regex regex module |  | REGEX |
| $expr used to evaluate expressions | find({$expr :{$gt : [‘$value’,’$target’]}}) | Just like  SELECT \* FROM TABLE WHERE COL1 > COL2 |
| $size (only relevant to arrays) | find(arraykey : {$size : somevalue}).pretty() | Something on the lines of count(\*) > somevalue |
| $all (only relevant to arrays) | Find({arraykey :{$all :[‘value1’, ‘value2’]}) | Something like  SELECT \* FROM TABLE WHERE ARRAYCOLUMN IN(VALUE1, VALUE2) |
| $elemMatch(only relevant for arrays) |  |  |

**Cursors**

1. db.collection.find() returns all the elements but this is not memory efficient. Hence use cursors.
2. const cursorName = db.collection.find()  
   cursorName.next() will then only return next element  
   other methods available are forEach() and hasNext()

**Sorting**

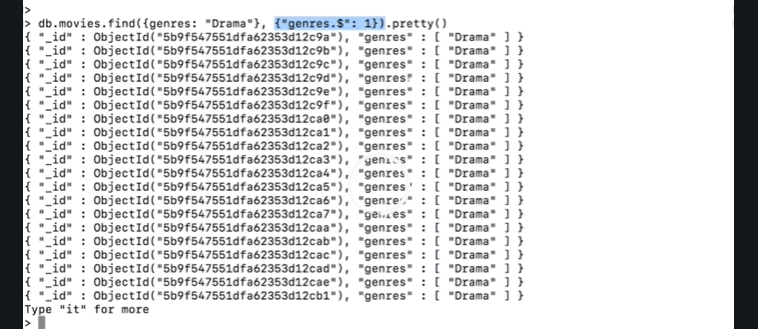
1. db.collection.find().sort({column : -1 or 1 , column2 : -1 or 1 and so on}).pretty()
2. -1 is descending and 1 is ascending

**Skip and Limit**

1. db.collection.find().sort({column : -1 or 1 , column2 : -1 or 1 and so on}).skip(number)pretty()
2. db.collection.find().sort({column : -1 or 1 , column2 : -1 or 1 and so on}).limit(number)pretty()

**PROJECTION**

1. This is nothing but SELECT COL, COL2, COL3 from TABLE
2. Whichever columns we specify as 1 will be displayed
3. db.collection.find({},{col1 :1, col2: 1, col3:1, col4.subcol :1 }).pretty()



1. $slice used as db.collection.find({‘rating.average’ : {$gt : 9}, {genres:{$slice :[1,3], name :1} }}). pretty() will print name and genres of all movies with rating greater than 9 but ***will skip first genre and display next 3***

**UPDATE IN DETAIL**

1. db.users.updateOne({\_id:ObjectId("5ebcb1e6a13a540fc440ac5e")},{$set :{hobbies : [{title:'Sports', frequency:3},{title:'Hiking', frequency:2},{title:'Cooking',frequence:5}]}})
2. db.users.updateOne({\_id:ObjectId("5ebcb1e6a13a540fc440ac5e")},{$set : {age:40, email : [chris@chris.com}}](mailto:chris@chris.com%7d%7d))
3. **$inc** is used to increment any value by a set amount. Example will increase age of Manuel by 2

db.users.updateOne({name:’Manuel’},{$inc:{age :2}})

db.users.updateOne({name:’Manuel’},{$inc:{age : - 2}}) will decrement age by 2

db.users.updateOne({name:’Manuel’},{$inc:{age : -2, $set :{isSporty : True}}}) will increase age by 2 and set isSporty to True

1. **$min** changes a value if existing value is lower than the current value.

db.users.updateOne({name:Chris},{$min: {age : 35}}) will be update age to 35 but after this if you run db.users.updateOne({name:Chris},{$min: {age : 38}}) no change will happen **as old value is the min value**.

1. **$max** changes a value if existing value is higher than the current value.

db.users.updateOne({name:Chris},{$max: {age : 42}}) will be update age to 42 but after this if you run db.users.updateOne({name:Chris},{$max: {age : 38}}) no change will happen **as old value is max value**.

1. **$mul** db.users.updateOne({name:Chris},{$max: {age : 1.1}}) will multiply age by 1.1
2. **$unset** can be used to get rid of values db.users.updateMany({},{$unset: {phone : “”}}) will delete all phone number fields.
3. **$rename** is used to rename fields db.users.updateMany({},{$rename: {age : “totalAge”}})
4. **Upsert** operation i.e. insert if missing else update canapplied in the following way
5. db.users.updateOne({name:”Maria”},{$set :{age: 29, hobbies :[{title: ‘Sports’, frequence:2},{title : ‘Hiking’, frequence: 3}, isSporty : True, {**upsert:true**} ]}})
6. Updating arrays is slightly complicated and is done using the $elemMatch in the following way.  
   Consider dataset like

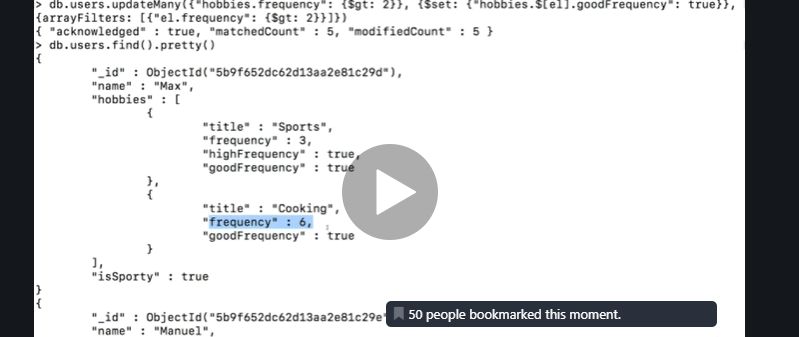


Now to add a new highFrequency field to hobbies we can use the below query.  
db.users.updateMany({hobbies : {$elemMatch :{title :’Sports’, frequency :{$gte :3}}}}, {$set : {‘hobbies.$.highFrequency’ :true}})

**$ is a special symbol which gives access to the array of the matched element**

1. db.users.updateMany({‘hobbies.frequncy’ : {$gt :2}},{$set :{‘hobbies.$.goodFrequencey’ : true}})  
   in this case all the records with frequency greater than 2 will now have a goodFrequency true but the catch is only first hobby matched will be updated. So if more than one hobby has a frequency greater than 2 only the first match will have this new goodFrequency field.
2. Db.users.updateMany({totalAge : {$gt : 20 }},{$inc : {‘hobies.$[].frequency’ : -2 }}) this will decrement the frequency for all records by 2. **The square brackets are important.**
3. Now in scenarios where multiple hobbies exist there and we need to update hobbies only with the matching condition we need a new syntax. Consider the hobbies array where there are more than 1 hobbies and each can have a frequency of 2 or more but you only want to update hobbies with frequency > 2. In this case the syntax would be  
   db.users.updateMany({‘hobbies.frequencey :{$gt : 3}’},{$set :{ ‘hobbies.$[el].bestAt’: ‘sushi’}},{ arrayFilters :{ [ ‘el.frequency’ : {$gt : 4} ]}})

If we consider below document then only cooking will have new field bestAt with value Sushi



1. **$push can be used to add elements** **to array. $push allows duplicate values so ue $addToSet to avoid duplicate insertions** as  
   db.users.updateOne({name : ‘Maria’},{$push : { hobbies : {title : ‘trekking’, frequency : 1}}}})  
     
   Another example to add multiple values to array is as below. It uses the **$each** keyword and the **$sort** is just used to ensure elements are inserted in certain order as good programming practice.  
   db.users.updateOne({name : ‘Maria’},{$push : { hobbies : {$each : [{title : ‘trekking’, frequency : 1},{title : ‘wine tasting’, frequency: 3}, sort(frequency : -1) }}}})
2. **$pull can be used to remove elements**db.users.updateOne({name : ‘Maria’},{$pull : { hobbies : {title : ‘trekking’}}}})
3. **$pop can be used to remove the last added element**

db.users.updateOne({name : ‘Maria’},{$pull : { hobbies : {title }}}})

**INDEXES**

1. db.collection.explain().find() will give the explain plan
2. db.collection.explain(‘executeStats’).find() will give you details such as time required to execute queries etc.
3. Indexes can be created on top level fields or embedded fields.

**db.collection.createIndex({‘fieldname’ : -1 or 1}, {background : true}) use background to create index In background process.**

**db.collection.createIndex({‘fieldname’ : -1 or 1, fieldname2 : -1 or 1) for composite index**

**IMP : For composite index the index sequence is used from left to right i.e. the above index will work on queries for fieldname1 and fieldname2 or queries which have just fieldname i.e left most field. It won’t work on 2nd field onwards.**

1. db.collection.getIndexes() **is used to check all indexes on a collection.**
2. db.collection.createIndex(‘fieldname’ : -1 or 1, {unique : true}) **to create unique index. Unique indexes give constraint error on null values as well. So if a document has two people where fieldname is null then we cannot create this index.**
3. db.collection.createIndex(‘dob.age : -1 or 1, {partialFilterExpression : {‘dob.age’ :{$gt : 60}}}) **will create index on age where age is greater than 60.** In the above scenario (point #5) we saw that unique indexes fail on null values. This can be taken care of by using partalFilterExpression as well like

db.collection.createIndex(‘fieldname’ : -1 or 1, {unique : true, partialFilterExpression : {email : { $exists : true}}})

1. **Time To Live index** can be used for applications where there is session specific data. **This also only works on dates.**  
   db.collection.createIndex({createdAt : 1},{expiresAfterSeconds : 10})  
   Any value added after this index was created will only remain n the collection for 10 seconds.
2. In case of array fields best to use multi key indexes which are nothing but indexes on fields in the array. Eg the below index would be used when we sear  
   db.collection.createIndex({hobbies.title : 1})
3. Super important to note that compound or composite indexes for arrays are valid only when one array field is used on one index. If an index has two array fields then it will fail beause it has to create cartesian product.
4. **Text Index** are created as db.collection.createIndex({hobbies.title : “text”}). Text indexes are expensive and should only be applied on one of the columns. Usage is as below.  
   **db.collection.find({$text :{$search : ‘some text’, {$caseSensitivity : true,}}})**. **Score** is another important parameter and cab be used to sort the matches returned in text index. Combined text index can be created using the same syntax and separating values using commas.  
     
   **db.collection.createIndex({title:”text”, description:’text”},{ default\_language:””, weight:{ title:1. Description:10}})** This is another variant where description is 10 times valuable than title and the default language is English. Common English words are removed while creating this index.



**GEOSPATIAL QUERIES**

1. Very useful for storing locations. The data needs to be stored in a GEOJson format which looks like below and has a type and coordinates.  
   ***The coordinates are available from GMAPS(after @) the first coordinate is latitude and the second is the longitude. So basically store in GEOJson in reverse of what is available on GMAPS***  
     
   {



"\_id" : ObjectId("5ebe2c63c405b7bb97cd088b"),

"name" : "Ascendas SingBridge",

"location" : {

"type" : "Point",

"coordinates" : [

73.6815152,

18.5920244

]

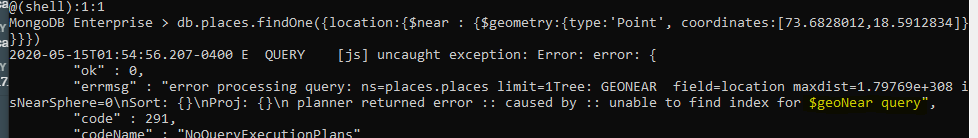
}

}

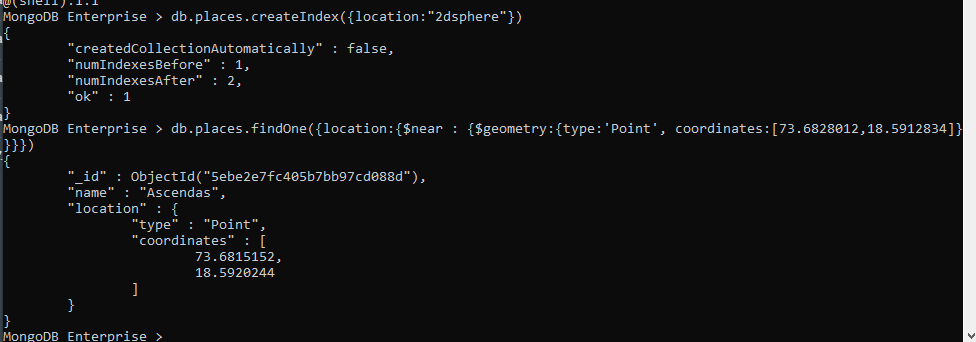
1. You can then query this record as follows  
   **db.places.findOne({location:{$near : {$geometry:{type:'Point', coordinates:[73.6828012,18.5912834]}}}})**

**This gives the below error because sometimes the GESPATIAL INDEX is required. It is only required sometimes and not always. Syntax for this index is**

**db.collection.createIndex({location : “2dsphere”})**



And post the index creation the same query works like a charm



1. The serach query can be made more comprehensive by adding in a **max and min distance** which is the distance to a place in meters. This distance is also available on GMAPS  
     
   **db.places.findOne({location:{$near : {$geometry:{type:'Point', coordinates:[73.6828012,18.5912834]},$maxDistance: 10, $minDistance:30}}})**
2. An interesting use case of this is also finding places within a certain area as below. Give coordinates of all places inside the array of coordinates as a new array  
     
   **db.places.findOne({location:{$geoWithin : {$geometry:{type:’Ploygon’, coordinates:[[place1, place2, place3 , place1]} }})**

Below is a detailed example where we try to find all elements near Ascendas except Hexaware

db.places.drop()

db.places.insertOne({name:'Ascendas', location:{type:'Point',coordinates:[73.6815152,18.5920244]}})

db.places.findOne({location:{$near : {$geometry:{type:'Point', coordinates:[73.6828012,18.5912834]}}}})

db.places.insertOne({name:'Ascendas', location:{type:'Point',coordinates:[73.6815152,18.5920244]}})

const cdc = [73.68132, 18.59278]

db.places.insertOne({name:'cdc', location:{type:'Point',coordinates:[73.68132, 18.59278]}})

const singbidge = [73.68649, 18.59105]

db.places.insertOne({name:'singbridge', location:{type:'Point',coordinates:[73.68649, 18.59105]}})

const snacks = [ 73.68503, 18.59166]

db.places.insertOne({name:'snacks', location:{type:'Point',coordinates:[73.68503, 18.59166]}})

const hexaware = [ 73.67917, 18.59095]

db.places.insertOne({name:'hexaware', location:{type:'Point',coordinates:[73.67917, 18.59095]}})

db.places.find({location:{$geoWithin : {$geometry:{type:'Polygon', coordinates:[[cdc, singbidge, snacks , cdc]]} }}})



1. Another useful use case is whether a coordinate or person is inside a certain area. Say you want to check if a person is inside the above area. In this case we need to

5.1) Store the area as a document.   
db.places.insertOne({area:’My Office’ : location:{type:'Polygon', coordinates:[[cdc, singbidge, snacks , cdc]]} })  
  
5.2) db.places.createIndex(area:’2dsphere’)

5.3) db.places.find({area: {$geoIntersects :{$geometry :{ type : ‘Point’, coordinates :[longitude, latitude]}}}})

1. Another use case is finding places within a radius. The centreSphere should be supplied two arguments, 1) the coordinates of the centre of the location and the radians. The conversion of distance from meters to radians needs to be done manually and the conversion is available in the docs.  
     
   db.places.find({area: {$centreSphere : [[longitude,latitude]]}})