Mongodb is schema less, means each and every row or document is having different number of columns

1st row or 1st document having 2 columns with int,string

2nd row or 2 document is having 3 columns String,String,String

3rd row or document can be an embedded document

1. Unless there is a space in the key we don’t need to keep the key in double quotes

db.flights.insertOne({name:””}); -- here no double quotes in the key

1. {} empty braces without condition means select all ex:- db.flights.deleteMany({}); this will delete all
2. Double or singlequotes db.flights.insertOne({name:””}); here single quotes or double quotes doesn’t make any difference , because shell is based on java script
3. We can give any value to \_id field ex:- db.passengers.insertOne( {\_id : “p123654”} );
4. Note 1 document size could be max 16 MB even if u have inner docs or not , so don’t embedded millions of messages inside a single document
5. Show dbs

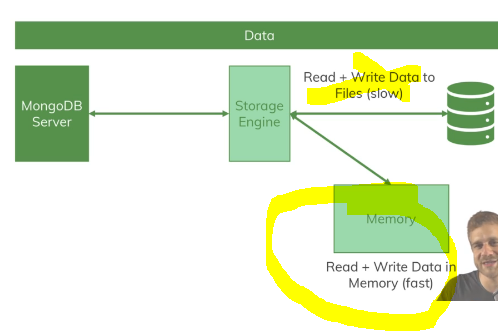
To see the list of dbs

Show collections

To see list of available collections

1. Use shops

To create a database and use it – if this database is not there it will just create one and switch to that



If mongo always interact with file system it will be literally be slow

Mongo Server will interact with storage engine and most latest used data will be maintained in the memory instead of file

|  |  |
| --- | --- |
| In RDBMS table | In mongodb it is called Collection |
| Row | document |
|  |  |

References

* Good one- Mongo db university course 15 hours java <https://learn.mongodb.com/catalog>

<https://learn.mongodb.com/learn/learning-path/mongodb-java-developer-path>

* Learn more about the MongoDB Drivers: <https://docs.mongodb.com/ecosystem/drivers/>
* Dive into the official Getting Started Docs: <https://docs.mongodb.com/manual/tutorial/getting-started/>
* 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/>
* 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/>

Helpful Articles/ Docs:

* insertOne(): <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>
* insertMany(): <https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>
* Atomicity: <https://docs.mongodb.com/manual/core/write-operations-atomicity/#atomicity>
* Write Concern: <https://docs.mongodb.com/manual/reference/write-concern/>
* Using mongoimport: <https://docs.mongodb.com/manual/reference/program/mongoimport/index.html>
* <https://www.mongodb.com/docs/manual/reference/operator/query/>

Shell specific

new Date()

new TimeStamp() are shell specific objects

generally for number field javascript accepts 64 bit floating point value like maximum 12345678901234567000

if u want to store super big numbers store as string

Predefined Functions

db.laptop.insertOne({lapName:”Lenovo”, lapInchSize: NumberInt(3)});

Arrays

Inside arrays you can keep string or Json objects

If u want to add an array to the existing document

db.updateOne(filter,data)

db.passengers.updateOne({"name": "Gordon Black"},{$set : {hobbies:["carom","Chess"]}});

After updating its looking like this

{

\_id: ObjectId("63aaeb3f03c92ae3e009dc80"),

name: 'Gordon Black',

age: 38,

**hobbies: [ 'carom', 'Chess' ]**

}

CreateCollection with constraints

In RDBMS we generally create collections with constraints like not null, unique key

Similarly in mongo dB also we can create collections with not null and column constraints like creating column we can define the datatype so that we can give only that type of data

Kindly check the file named script or “**creating collections with constraints.js”**

**In the below we created a Collection called Employee with 2 mandatory fields empId,empName**

db.createCollection('Employee',{

validator:{

$jsonSchema:{

bsonType:'object',

required:['empId','empName'],

properties:{

title:{

bsonType:'string',

description: 'Title must be a string cant be null'

}}}}}); //similarly declare for all fields

Example 2:-

db.createCollection('user',{

validator:{

$jsonSchema:{

bsonType:'object',

required: ['userName','sex'],

properties: {

userName:{

bsonType:'string',

description:'mandatorieee'

}

}

}

}

});

If any of above 2 columns are missing and even if we gave wrong data type then insertion will fail

As below Uncaught:

MongoServerError: Document failed validation

Additional information: {

failingDocumentId: 123,

details: {

operatorName: '$jsonSchema',

CRUD operations

Delete

deleteOne(filter,options);

deleteMany(filter,options)

Embedded documents

U can have a document inside a document, such total 1 document can have totally 300 embedded documents inside it, but maximum of 16MB only in total for a collection

Db.flights.updateMany(filter,data);

If u want to update general field without embedded document

db.flights.updateMany({} ,{$set: {engineDetails: “garuda”}});

db.flights.updateMany({} ,{$set: {engineDetails: {horsepower:1600, engineName: ‘garuda’} }});

Result as

{ "\_id": "i1", "flightName": "Indigo", "capcacity": 200, "manufacturedBy": "Tata", "wheels": 4,

**"engineDetails": {**

**"horsepower": 1600,**

**"engineName": "garuda"**

}

}

Delete collection

Db.collectionName.drop()

Ex:- db.Employee.drop() This table will be dropped

deleteMany

db.collectionName.deleteMany({}) means it will delete everything all the rows

Find

1. To fetch all records use find method with no condition

db.products.find().pretty(); -- as no condition was given it will fetch all records

lets say if this find method is giving 20,000 lakh records, but in general find will not give all records at once , it will just gives us the cursor

1. find(filter,Options)

findOne(filter,Options) // it will gives only 1st matching element

db.flights.find({intercontinental:true});

output :-

[ { \_id: ObjectId("63a9c0a20d7ed583364551e0"), departureAirport: 'MUC',

arrivalAirport: 'SFO', aircraft: 'Airbus A380', distance: 12000, intercontinental: true }]

find with $gt

find all flights whose distance is greater than 1000

db.flights.find( {distance: {$gt:900} } ); //gives 2 docs

db.flights.findOne( {distance: {$gt:900} } ); //gives 1 doc

Accessing inner fields

Here we accessed inner field alone

db.flights.findOne({"\_id":"Air1"}).engineDetails;

|  |
| --- |
| { horsepower: 1600, engineName: 'garuda'} |

Now we can access horsepower field also

db.flights.findOne({"\_id":"Air1"}).engineDetails.horsepower;

Searching with array field

This is the actual document

db.flights.findOne({"\_id":"Air1"});

|  |
| --- |
| { \_id: **'Air1'**, |
| flightName: **'AirIndia'**, |
| capcacity: 250, |
| manufacturedBy: **'CTS'**, |
| designedBy: **'Accenture'**, |
| wheels: 4, |
| |  | | --- | | engineDetails: { horsepower: 1600, engineName: **'garuda'** }, | | stops: [ **'hyd'**, **'chennai'** ] } | |

}

Query array **contents : here sops is an array field**

**db.flights.findOne({stops: 'hyd'});**

|  |
| --- |
| { \_id: **'Air1'**, |
| flightName: **'AirIndia'**, |
| capcacity: 250, |
| manufacturedBy: **'CTS'**, |
| designedBy: **'Accenture'**, |
| wheels: 4, |
| engineDetails: { horsepower: 1600, engineName: **'garuda'** }, |
| stops: [ **'hyd'**, **'chennai'** ] }  searching with inner elements |

db.flights.findOne({"engineDetails.horsepower":1600});

Insert

Even though there is no collection named products – it will create one

1. insertOne(data,Options)
2. insertMany(data,Options)

Unless there is a space in the key we don’t need to keep the key in double quotes

InsertOne

No need to create collection, directly if u insert if document is collection is present it will add this document

Else it will create the collection and add this document

db.products.insertOne({product:”Book”,”cost”:12.99 })

here u can give \_id also – and u don’t need to give objectId type and type is fine 1 record can be of ObjectId type another maynot

db.flights.insertOne( {\_id:”i1”,flightName: “Indigo”,capcacity:200, manufacturedBy: “Tata” });

InsertMany

insertMany(Array arr) this function expects to pass an array

**db.flights.insertMany(**

[ { "departureAirport": "MUC", "arrivalAirport": "SFO", "aircraft": "Airbus A380", "distance": 12000, "intercontinental": true },

{ "departureAirport": "LHR", "arrivalAirport": "TXL", "aircraft": "Airbus A320", "distance": 950, "intercontinental": false }]**) ;**

InsertOne With Arrays of objects

Db.companies.insertOne({\_id:"c2", name:"Fresh Apple Inc","isStartUp":true, employees:33,funds:01234567890123456798,locations:[ {hyderabad:50000,mainCampus:"deccan park"} , {chennai:60000,mainCampus:"siruseri"}], foundedDate: new Date()});

|  |
| --- |
| { acknowledged: true, insertedId: 'c2' }  Result is |

{

\_id: 'c2',

name: 'Fresh Apple Inc',

isStartUp: true,

employees: 33,

funds: 1234567890123456800,

locations: [

{ hyderabad: 50000, mainCampus: 'deccan park' },

{ chennai: 60000, mainCampus: 'siruseri' }

],

foundedDate: ISODate("2022-12-28T06:25:04.409Z")

}

Inserting all records parallelly With ordered

Db.collName.insertMany(document , options)

While doing insertMany if u are trying to insert 10 records, if 1st record fails (due to any issue like duplicate key exception), remaining 9 records will not be inserted,

But if u want all records to be inserted parallelly irrespective of failing of previous records then use the key order:false option , so that even if the first record fails it doesn’t impact, 9 records will be inserted paralelly

shop> db.pen.insertMany([{\_id:'Renault', cost:3},{\_id:'Hero',cost:4}]);

db.pen.insertMany([{\_id:'Renault', cost:31}},{\_id:'HeroCycle',cost:41} ]); //Because same id

now as per above I will try to insert Renault pen it will throw error because same id, but the problem is it will not insert the remaining document

db.pen.insertMany([{\_id:'Renault', cost:31},{\_id:'HeroCycle',cost:41} ], { ordered:false});

writeConcernErrors: [],

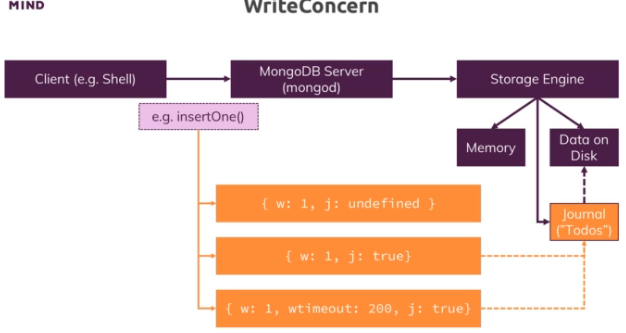
insertedIds: [ { index: 0, \_id: 'Renault' }, { index: 1, \_id: 'HeroCycle' } ],

nInserted: 1,

nUpserted: 0,

Now in this case also it will throw error but it will successfully insert second document eventhough first document failed

WriteConcern



Mongo Import

Where tvshows.json is the file to import

Mongoimport tvshows.json -d dbname –c collectionName –jsonArray –drop

Operators

|  |  |
| --- | --- |
| [$eq](https://www.mongodb.com/docs/manual/reference/operator/query/eq/#mongodb-query-op.-eq) | Matches values that are equal to a specified value. |
| [$gt](https://www.mongodb.com/docs/manual/reference/operator/query/gt/#mongodb-query-op.-gt) | Matches values that are greater than a specified value. |
| [$gte](https://www.mongodb.com/docs/manual/reference/operator/query/gte/#mongodb-query-op.-gte) | Matches values that are greater than or equal to a specified value. |
| [$in](https://www.mongodb.com/docs/manual/reference/operator/query/in/#mongodb-query-op.-in) | Matches any of the values specified in an array. |
| [$lt](https://www.mongodb.com/docs/manual/reference/operator/query/lt/#mongodb-query-op.-lt) | Matches values that are less than a specified value. |
| [$lte](https://www.mongodb.com/docs/manual/reference/operator/query/lte/#mongodb-query-op.-lte) | Matches values that are less than or equal to a specified value. |
| [$ne](https://www.mongodb.com/docs/manual/reference/operator/query/ne/#mongodb-query-op.-ne) | Matches all values that are not equal to a specified value. |
| [$nin](https://www.mongodb.com/docs/manual/reference/operator/query/nin/#mongodb-query-op.-nin) | Matches none of the values specified in an array. |
| [$and](https://www.mongodb.com/docs/manual/reference/operator/query/and/#mongodb-query-op.-and) | Joins query clauses with a logical AND returns all documents that match the conditions of both clauses. |
| [$not](https://www.mongodb.com/docs/manual/reference/operator/query/not/#mongodb-query-op.-not) | Inverts the effect of a query expression and returns documents that do not match the query expression. |
| [$nor](https://www.mongodb.com/docs/manual/reference/operator/query/nor/#mongodb-query-op.-nor) | Joins query clauses with a logical NOR returns all documents that fail to match both clauses. |
| [$or](https://www.mongodb.com/docs/manual/reference/operator/query/or/#mongodb-query-op.-or) | Joins query clauses with a logical OR returns all documents that match the conditions of either clause.   |  |  | | --- | --- | | [$exists](https://www.mongodb.com/docs/manual/reference/operator/query/exists/#mongodb-query-op.-exists) | Matches documents that have the specified field. | | [$type](https://www.mongodb.com/docs/manual/reference/operator/query/type/#mongodb-query-op.-type) | Selects documents if a field is of the specified type. | | [$text](https://www.mongodb.com/docs/manual/reference/operator/query/text/#mongodb-query-op.-text) | Performs text search. | | [$where](https://www.mongodb.com/docs/manual/reference/operator/query/where/#mongodb-query-op.-where) | Matches documents that satisfy a JavaScript expression. | |

Example:-

db.movies.find({runtime: {$lt :30}}).count();

Only thing is everything is an object for find method we should pass an Object, here key is string and value should also be an object

**db.movies.find({runtime: {$eq :30}}).count();**

**62**

**db.movies.find({runtime :30}).count(); both above statements are same , even if u don’t gove eq** operator those are same

### Pretty

U can call this function only on a cursor ex:-

db.find().pretty() – works because find returns a cursor

db.findOne().pretty(); - gives exception because findOne doesnot return a cursor

db.insertOne().pretty(); gives exception because insertOne method is not returning any data hence cursor will not be returned

cursor will be returned only in case when you are returning huge data

### Projections

Means selecting only few columns ,if database is having 100-200 columns if u don’t want all of them to be fetched then fetch only some columns, need means 1 ,If u don’t require that column give 0

Db.flights.find(filter,options);

Db.flights.find({},{name:1, \_id: 0 }); // since I don’t want to filter I didn’t gave any condition

By default \_id column will be fetched if u don’t want that to be fetched, hence I gave 0 wantedly

Replaceone

While replacing it will maintain \_id alone and it will replace entire all fields with this field

Db.flights.replaceOne(condition,data);

db.flights.replaceOne(data);

//practice here

shutdownServer

db.shutdownServer(); //means shutdown the mongo db server

Update

If u have partial modifications then prefer this

1. updateOne(filter,data,options)
2. replaceOne(filter,data,options)

Its Always recommended to use updateOne(), updateMany()

shop> db.flights.update({\_id:'i1'},{engine:'powerful'});

the problem was , since we are not using **$set** : {engine:'powerful'} here $set is missing so it was overriding all fields,means all fields are gone only this field will be set alone , whereas in updateOne(),updateMany() the $set symbol keyword is mandatory

as here the entire record is getting overridden seems this could have got deprecated.

DeprecationWarning: Collection.update() is deprecated. Use updateOne, updateMany, or bulkWrite.

MongoInvalidArgumentError: Update document requires atomic operators

UpdateOne

updateOne(filter,data,options)

1st argument is filter-where condition

db.flights.updateOne( {distance:12000}, {$set: { engine : ”double powered” } } );

db.flights.find();

**Before update**

There is no field called **designedBy**

{ \_id: 'Air1', flightName: 'AirIndia', capcacity: 250, manufacturedBy: 'CTS' },

**After running the query**

db.flights.updateOne({\_id:"Air1"}, {$set:{designedBy:"Accenture"}});

note:- if the field “designedBy “is present for that document it will update the key value else it will create that field key value pair for that document

[ { \_id: 'Air1', flightName: 'AirIndia', capcacity: 250, manufacturedBy: 'CTS', designedBy: Accenture' },

UpdateMany

1. updateMany(filter,data,options)

To select all documents without any condition then use empty flower braces {}

db.flights.updateMany({}, {$set: {wheels:4} } );

Here since first option is having empty braces {} means no condition it will select all records and it will update all records if there is a field called wheels in a document it will update else it will add that field to that document

{ acknowledged: true, insertedId: null, matchedCount: 3, modifiedCount: 3, upsertedCount: 0 }

In above already in that collection only 3 records are there before running that query, now it selected all docs it updated all docs

After running the query it will look like this

{ \_id: 'i1', flightName: 'Indigo', capcacity: 200, manufacturedBy: 'Tata', **wheels: 4** },

{ \_id: 'Air1', flightName: 'AirIndia', capcacity: 250, manufacturedBy: 'CTS', designedBy: Accenture',

**wheels: 4** },

{ \_id: 'king1', flightName: 'KingFisher-1', capcacity: 50, manufacturedBy: 'IQA', **wheels: 4** }

If u want a new field to all existing records then refer below

1st param is {} because we want to select all records

db.flights.updateMany({},{$set :{details:null}});

so for all documents new fields called details will be added with null value

Datatypes

**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.

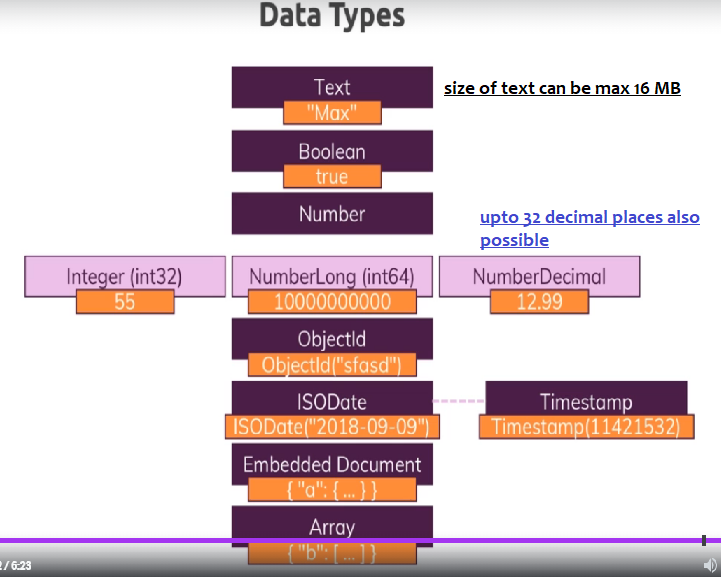
**NumberInt** creates a **int32** value => NumberInt(55)

**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.



### Misc functions

##### 

This stats method can be called on database and on individual collection too

Db.stats(); entire database stats

Db.Flights.stats();

db.laptop.stats(); // here laptop is collection name

result is

{ ns: 'shop.laptop', size: 59, count: 1, avgObjSize: 59, numOrphanDocs: 0, storageSize: 4096,

freeStorageSize: 0, capped: false, wiredTiger: { metadata: { formatVersion: 1 },