mongodb://localhost:27017

>mongod --dbpath "C:\Program Files\MongoDB\Server\6.0\db"

>mongod --logpath "C:\Program Files\MongoDB\Server\6.0\log\log.log"

**What is Mongo Db ?**

[

{

"departureAirport": "MUC",

"arrivalAirport": "SFO",

"aircraft": "Airbus A380",

"distance": 12000,

"intercontinental": true

},

{

"departureAirport": "LHR",

"arrivalAirport": "TXL",

"aircraft": "Airbus A320",

"distance": 950,

"intercontinental": false

}

]

**CRUD Operations and Mongo DB**

**Read :**

find(filter, options);

db.FlightData.find().pretty()

**Cursor Object ->** find(): gives you back a cursor object and not all the data

it doesnt give an array of object because collection can be very big

data and it will take time so instead it gives back cursor object which is an

object with lots of metadata behind it that allows us to Cycle through results

db.PassengersData.find().toArray() :

.toArray will go ahead and exhaust the cursor.

it will go throught entire list fetch all the documents and not stop after first 20.

db.PassengersData.find().forEach( (PassengersData) => {printjson(PassengersData)})

.pretty() is method simply which exist on the cursor.

i.e it doesnt support .findOne(). which doesnt give a cursor bcz it only gives us one document

.find() give multiple documents therefor we dont get data immediately with the get cursor instead

db.FlightData.find({distance : {$gt: 1000}})

findAll(filter, options);

**Update :**

updateOne(filter, data, options)

db.FlightData.updateOne({"distance":950},{$set:{"marker":"delete"}})

db.FlightData.updateOne({ \_id : ObjectId("647df04f32ce2659b0d6ff7e")}, {$set : {delayed : true}})

updateMany(filter, data, options)

db.FlightData.updateMany({},{$set:{"marker":"delete"}})

update(filter, data, options)

update({\_id : ObjectId("nasdasd")}, {delayed: false})

it replace existing object with new object unlike others they just

update on parameter

replaceOne(filter, data, options)

replaceOne({\_id : ObjectId("nasdasd")},

{

"departureAirport": "MUC",

"arrivalAirport": "SFO",

"aircraft": "Airbus A380",

"distance": 12000,

"intercontinental": true

}

)

**Delete :**

deleteOne(filter, options)

db.FlightData.deleteOne({"departureAirport" :"TXL"})

deleteMany(filter, options)

db.FlightData.deleteMany({marker : "delete"})

db.products.deleteMany({})

**Create :**

insertOne(data, options);

insertMany(data, options);

db.FlightData.insertMany([

... {

... "departureAirport": "MUC",

... "arrivalAirport": "SFO",

... "aircraft": "Airbus A380",

... "distance": 12000,

... "intercontinental": true

... },

... {

... "departureAirport": "LHR",

... "arrivalAirport": "TXL",

... "aircraft": "Airbus A320",

... "distance": 950,

... "intercontinental": false

... }

... ]

... )

**What is Projection :**

MongoDB projection is a powerful tool that can be used to extract

only the fields you need from a document—not all fields.

db.collection\_name.find({},{<field> : <value>})

db.PassengersData.find({}, {name: 1}).pretty()

if you want to exclude id then

db.PassengersData.find({}, {name: 1, \_id: 0}).pretty()

**Embedded and Reference Documents :**

Embedded documents are an efficient and clean way to store related data, especially data that's regularly accessed together.

db.FlightData.updateMany({}, {$set: {status : {description : "on-time", lastUpdated: "1 Hour ago"}}})

db.PassengersData.updateMany({name : "Albert Twostone"}, {$set: {status : {hobbies : ["sports","cooking"]}})

db.PassengersData.find({name : "Albert Twostone"}).pretty()

db.PassengersData.findOne({name : "Albert Twostone"}).hobbies

db.PassengersData.find({hobbies : "sports"}).pretty()

db.FlightData.find({"status.description":"on-time"}).pretty()

db.FlightData.find({"status.details.responsible":"ABC"}).pretty()

[

{

\_id: ObjectId("648f1623ff81e4cd71c664a8"),

firstName: 'A',

lastName: 'X',

age: 29,

history: [

{ disease: 'cold', treament: { medicine: [ 'med1', 'med2' ] } }

]

},

{

\_id: ObjectId("648f1655ff81e4cd71c664a9"),

firstName: 'Amir',

lastName: 'Y',

age: 40,

history: [

{ disease: 'cough', treament: { medicine: [ 'med5', 'med6' ] } }

]

},

{

\_id: ObjectId("648f1674ff81e4cd71c664aa"),

firstName: 'C',

lastName: 'Z',

age: 19,

history: [

{ disease: 'cold', treament: { medicine: [ 'med1', 'med2' ] } },

{ disease: 'fever', treament: { medicine: [ 'med3', 'med4' ] } }

]

}

]

db.PatientData.updateOne({lastName:"Y"},{$set:{firstName:"Amir", age:40, history : [{disease :"cough", treament :{ medicine : ["med5", "med6"]}} ]}})

db.PatientData.find({age: {$gt : 25}}).pretty()

db.PatientData.deleteMany({"history.disease":"cold"})

Create:

db.patients.insertMany([{

firstName: "Jack",

lastName: "Daw",

age: 35,

history: [{

disease: "cold",

treatment: "Anti-Histamines"

}]

}, {

firstName: "John",

lastName: "Doe",

age: 27,

history: [{

disease: "type-1 diabetes",

treatment: "Insulin"

}]

}, {

firstName: "Jane",

lastName: "Doe",

age: 33,

history: [{

disease: "cold",

treatment: "Anti-Histamines"

}, {

disease: "typhoid",

treatment: "Broad-Spectrum Anti-Biotics"

}]

}])

Update:

db.patients.updateOne({

firstName: "John"

}, {

$set: {

firstName: "Jon",

age: 28,

history: [{

disease: "type-2 diabetes",

treatment: "Insulin"

}]

}

})

Read:

db.patients.find({

age: {

$gt: 30

}

})

Delete:

db.patients.deleteMany({

"history.disease": "cold"

})

Important: We will regularly start with a clean database server (i.e. all data was purged) in this course.

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().

db.companies.drop()

**Modules :**

Data Types :

db.stats()

typeof db.numbers.findOne().a

**Data Types & Limits**

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)

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")

});

By browsing the API docs for the driver you're using, you'll be able to identify the methods for building int32s, int64s etc.

var dsid = db.patients.findOne().diseaseSummary

db.diseaseSummaries.findOne({\_id:dsid})

db.orders.aggregate([{$lookup: {from: "customers", localField: "productId", foreignField : "\_id", as : "customerOrders"}}]).pretty()

**Validation :**

db.createCollection("posts", {validator : {$jsonSchema : { bsonType : "object", required : ["title", "text" , "creator" , "comments"], properties : { title : { bsonType : "string", description : "must be string and is required" }, text : { bsonType : "string", description : "must be string and is required" }, creator : { bsonType: "objectId", description : "must be an objectid and is required" }, comments : { bsonType : "array", description : "must be array and is required", items : { bsonType : "object", required : ["text", "author"] ,properties : { text : { bsonType : "string", description : "must be string and is required" }, author : { bsonType : "objectId", description : "must be objectid and is required" } } } }, } } }})

{ ok: 1 }

blog

Selection deleted

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'

}

}

}

}

}

}

}

});

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/

**ordered :**

db.hobbies.insertMany([{"":"","",""},{"":"","",""}],{ordered : false})

**WriteConcern :**

W : 1 ->

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 1}})

w : 0 ->

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 0}})

j : true ->

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 1, j : true}})

j : false ->

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 1, j : false}})

wtimeout ->

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 1, j : true, wtimeout : 200}})

db.PassengersData.insertOne({name : "Amir", age : 30},{writeConcern : {w : 1, j : true, wtimeout : 1}})

So let's play around with the write concern and for that I'll go back to my persons,

there we already have a couple of persons in the database of course and

all these writes succeeded and they normally will of course.

Now let's insert a new person with insert one and the write concern can also be set on all other write

operations like insert many, I will use insert one here with a name of Chrissy, age 41

and now I specify a second argument again where we previously set orders,

I will now set my write concern.

I do this by setting the write concern,

named like this,

the casing is important,

so the write concern key, I set it to a value which again is a document which has the shape you saw on

the slides.

W 1 is the default,

it simply means I need to be sure that the server acknowledge this,

you can set this to 0. If you do this, you get back acknowledged false

but if I find everything, you see that Chrissy was inserted.

So you get back a different result, also without an objectID

because it can't give you one, the server hasn't really registered this write yet, you just sent the request

and you immediately return,

you don't wait for a response of this request, so to say.

So the storage engine had no chance to store it in memory and generate that objectId and therefore, you get

back acknowledged false because you sent the request, you don't even know if it reached the server.

This is of course super fast because you don't have to wait for any response here, for any ID generation

but obviously, it tells you nothing about whether this succeeded or not. Could still be a valid option

for data where it's ok for you, if some data does not end up in a database,

so if you log some value every second about an application and you don't care if a couple of seconds

get lost, you could do that.

So that is w 0, the default with

Alex who is 36, the default is w 1 of course, this gives you the output you saw before, acknowledge

true

and the inserted ID.

Now let's go for Michael and let's play around with the journal,

the journal can be set to true,

the default is undefined or false,

so if I set it to false, I have the same result as before.

Now if I change it to McKayla and we set the journal to true now, the output for us does not change and

.

it also was super fast here because everything runs locally

and it's not like the journaling will take four hours but it will have been a little bit slower because

the entry will have been added to the journal and we waited for that journal editing to finish here.

So here, we have higher security because we can also be sure that it ended up in this to do list of the

storage engine which will eventually lead to the writes happen to database files.

So now we got this,

now let me add another person, Aliya 22 and let me also add a third option, w timeout. If I add this like this,

it succeeded. Now

if I repeat it with a very small value,

it also succeeded because this is super fast here but it is an option which you can set in case

you get shaky, a shaky connection or speed really matters and your connection is generally good but you

can't rule out that once in a year,

it's kind of shaky and you would then rather have that write fail and you recognize this in your client

application of course because you'll get back an error here too.

So you'll have that write fail and therefore you can try again later but you don't wait unnecessarily

long.

So this can absolutely be something you are fine with.

**Atomicity :**

Now there's one important concept I want to touch on and that is the atomicity of any write operation.

Let's say we have an insert one operation but this could be any write operation again,

now such operation

most of the time will of course succeed but it can fail,

there can be an error and here I'm talking about errors that happened whilst the document is inserted, so whilst

it's written to memory, whilst it's handled by a storage engine.

You could imagine that if you have a document with multiple fields, let's say also with a couple

of embedded documents, that some of these fields end up in the database,

so in the database files also whilst others are not included

because in between,

so whilst document was processed, the server had an issue.

So if you added a person with name, age and hobbies, maybe name and age was saved but not hobbies, mongodb

protects you against this,

it guarantees you an atomic transaction which means the transaction either succeeds as a whole or it fails

as a whole

and if it fails during the write, everything is rolled back for this document you inserted

and that is important.

It's on a per document level, that document means the top level document,

so it includes all embedded documents, all arrays so that is all included

but if you use insert many with multiple documents being inserted, then you don't get this,

so you only have support on a document level

but if you have multiple documents in one operation, like insert many, then only each document on its

own is guaranteed to either fail or succeed

but not insert many.

So if you have an issue during insert many, that just means that the documents on which it failed are

not added

and then the exact behavior depends on whether you used ordered or unordered inserts but the document

which already were inserted will not be rolled back

as I explained in the ordered and unordered insert lecture. We are actually able to control this on this

bulk insert or bulk update level too

and that will be related to a concept called transactions but that is something I will cover towards

the end of this course because it requires some additional knowledge about mongodb and how the

service work there that we don't have yet. But on a document level including all embedded documents, you

have this atomic operation guarantee.

information alert

**Import File :**

-d : for database

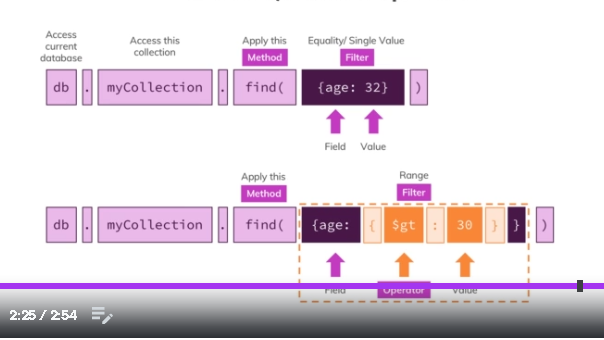
-c : for collection

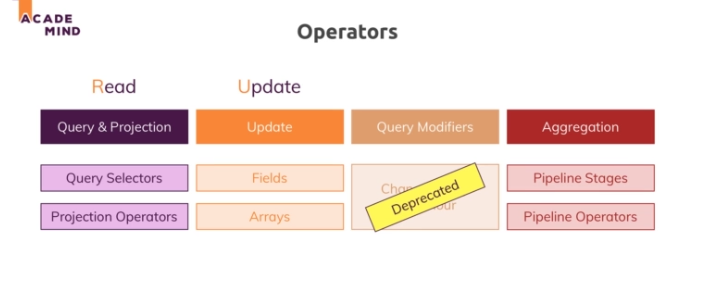
--jsonArray : to make mongo import aware for array format data

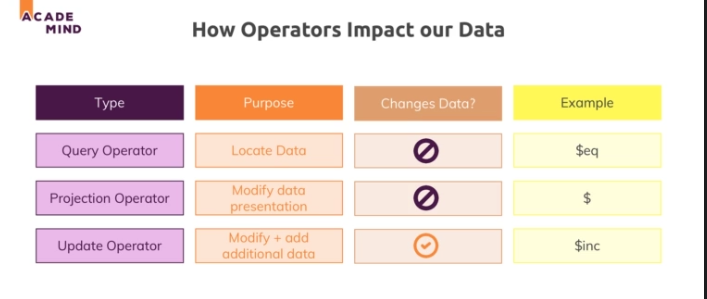
--drop : if exist then dropNow the last thing I'll add is --drop, this simply means if this collection should already exist, it will be dropped and then re-added otherwise it we'll append the data to the existing collection and that might also be what you want

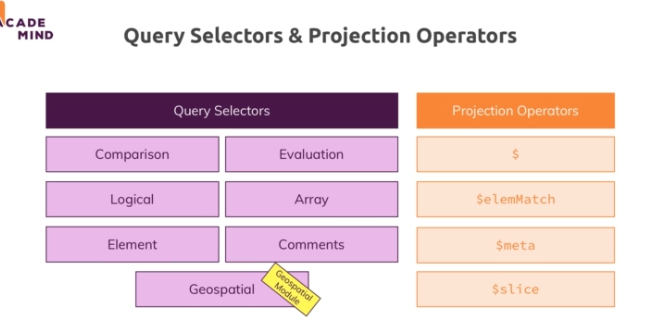
mongoimport tv-shows.json -d movie -c movieData --jsonArray – drop

Read Operations :









Db.movieData.find({runtime : { $lt : 40 }})

Db.movieData.find({runtime : { $eq : 40 }})

Db.movieData.find({runtime : { $ne : 40 }})

Db.movieData.find({runtime : { $in : [30, 42] }})

Db.movieData.find({runtime : { $nin : [30, 42] }})

Db.movieData.find({“rating.average” : { $gt : 70 }}) -> for object

Db.movieData.find({genres : “Drama”}) -> for array

Db.movieData.find({genres : [“Drama”]}) -> for exactly that array

db.movieData.find({"rating.average" : {$gt : 5}}).count()

db.movieData.find({"rating.average" : {$lt : 5}}).count()

db.movieData.find({ $or : [{"rating.average" : {$lt : 5}}, {"rating.average" : {$gt : 5}}]}).count()

db.movieData.find({ $nor : [{"rating.average" : {$lt : 5}}, {"rating.average" : {$gt : 5}}]}).count()

db.movieData.find({ $and : [{"rating.average" : {$lt : 5}}, {"rating.average" : {$gt : 5}}]}).count()

db.movieData.find({"rating.average" : {$gt : 9}, genres : "Drama" }).count()

db.movieData.find({ rating : { $not : { $eq : 60 } } })

db.movieData.find({ rating : { $ne : 60 } } )

db.userData.find({ age : {$exists : true }}.pretty()

db.userData.find({ age : {$exists : true, $gt : 30 }}.pretty()

db.userData.find({ age : {$exists : true, $gte : 30 }}.pretty()

db.userData.find({ phone : {$type : “string” }}.pretty()

db.userData.find({ phone : {$type : “number” }}.pretty()

db.userData.find({ phone : {$type : “double” }}.pretty()

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

db.salesData.find({$expr : {$gt : ["$volume", "$target"]}})

db.salesData.find({$expr : {$gt : [ "volume", "target" ]}}).pretty()

db.salesData.find({$expr : {$gt : [ { $cond : {if: {$gte : ["volume", 190]}, then : {$subtract : ["$volume", 30]}, else : "$volume" } }, "$target" ]}}).pretty()

db.user.find({hobbies : {$size: 3}})

db.moviestarts.find({genre : {$all : ["action", "thriller"]}})

db.user.find({$and : [{"hobbies.tittle": "Sports"}, {"hobbies.frequency": {$gte: 2}}]})

db.user.find({hobbies : {$elemMatch : {tittle : "Sports", frequency : {$gte : 3}}}})

**Cursor :**

Let's now briefly come back to that cursor thing which I already touched on earlier in this course, the

find method unlike the findOne method, yields us a cursor

if you remember.

Now why is that important and why does it do that? Well if we have our client communicate with the mongodb

server,

we potentially get back thousands or even millions of documents with find, especially if we have no

condition in there

but even with a condition, you easily have a condition that still meets like 1000 documents or more

depending on the scale of your app.

So you get back all these results and that is very inefficient because all these results have to be

fetched from the database,

they have to be sent over the wire and then they have to be loaded into memory in your client application.

So these are three things that are not optimal because chances are you will not need all thouand documents

at the same time and therefore, find gives you a cursor. A cursor is basically a pointer which has the

query you wrote stored and which can therefore quickly go to the database and say hey, give me the next

batch,

give me the next batch

and indeed you work with batches of data then. You fetch the data one by one, so one document by a time

and it really is only transferred over the wire when you request the next one.

Now in the shell, we get 20 by default because the shell automatically basically takes the cursor and gets

us the first 20 documents before we can get more.

If you write an application with a mongodb driver and we'll see that in the from driver to

shell module

at the end of the course, then you have to control that cursor manually and make sure you get back your

results. And that cursor approach is

great because it saves resources.

If you have a query that meets 1000 documents, but let's say you have a website where you only display

10 items, let's say 10 products you fetched at a time anyways, then there is no need to load all thousand

results that matched your query right at the start.

Instead you would only fetch the first 10,

display them on the screen and then go ahead and fetch the next 10

when the user navigated to the next page or anything like that. This is the idea behind a cursor,

now we'll find out how to work with a cursor

later as I mentioned when I talk about how you use the mongodb driver in a real application

but I can already show you some things in this module.

const dataCursor = db.movieData.find()

datacursor.next()

**dataCursor.forEach( doc => { printjson(doc)} )**

dataCursor.next()

dataCursor.hasNext()

db.movieData.find().next()

**1: asc**

**-1 : desc**

db.movieData.find({"rating.average" : {$gte : 7}}).sort({"rating.average":1}).pretty()

db.movieData.find({"rating.average" : {$gte : 7}}).sort({"rating.average":-1}).pretty()

db.movieData.find({"rating.average" : {$gte : 7}}).sort({"rating.average":-1, runtime : -1}).pretty()

db.movieData.find({"rating.average" : {$gte : 7}}).sort({"rating.average":1, runtime : -1}).skip(10).pretty()

db.movieData.find({"rating.average" : {$gte : 7}}).sort({"rating.average":1, runtime : -1}).skip(100).limit(10).pretty()

Projection :

db.movieData.find({}, {name : 1, genres : 1, runtime : 1, rating : 1, “schedule.time”:1, \_id : 0})

Projection in Array :

db.movieData.find({genres:"Drama"},{"genres.$":1})

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

db.movieData.find({genres:"Drama"},{genres : {$slice : 2}, name: 1 })

{$slice : [1, 2]} :

db.movieData.find({genres:"Drama"},{genres : {$slice : [1, 2]}, name: 1 })

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

db.UserData.updateOne({\_id : ObjectId("64a598d7c3329a81a548999d")}, {$set : {age : 50, phone: 9876543210}})

db.UserData.updateOne({name: "Manuel"}, {$inc : {age : 2}})

db.UserData.updateOne({name: "Manuel"}, {$inc : {age : 2}, $set: {isSporty : false}})

db.UserData.updateOne({name : "Chris"}, {$min : {age : 35}})

db.UserData.updateOne({name : "Chris"}, {$min : {age : 38}}) –didn’t modify

db.UserData.updateOne({name : "Chris"}, {$max : {age : 38}})

db.UserData.updateOne({name : "Chris"}, {$mul : {age : 1.1}})

db.UserData.updateMany({isSporty : true}, {$set : {ohone : null}})

db.UserData.updateMany({isSporty : true}, {$unset : {ohone : ""}})

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

db.UserData.updateOne({name : "Maria"},{$set: {age: 29, hobbies : [{title : "Good Food", frequency : 3}], isSporty: true}}, {upsert : true})

db.UserData.find({$and : [{"hobbies.title":"Sports"}, {"hobbies.frequency":{$gte: 3}}]})

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

--Array Matched Update

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

.$. 🡪

db.UserData.updateMany({"hobbies.frequency": {$gt:2}}, {$set: {"hobbies.$.goodFrequency": true}})

.$[]. 🡪

db.UserData.updateMany({totalAge: {$gt:30}}, {$set: {"hobbies.$[].frequency": -1}})

db.UserData.updateMany({"hobbies.frequency": {$gt:2}}, {$set: {"hobbies.$[el].goodFrequency": true}}, {arrayFilters : [{"el.frequency": {$gt: 2 }}]})

* Without replacing other doc you want to insert some values.

db.UserData.updateOne({name : "Maria"},{$push : {hobbies : { title : "Sports", frequency : 2 }}})

🡪

db.UserData.updateOne({name : "Maria"},{$push : {hobbies : { $each :[ {title : "Good Drinks", frequency : 5 } , { title : "Hiking", frequency : 2 } ] }}})

🡪

db.UserData.updateOne({name : "Maria"},{$push : {hobbies : { $each :[ {title : "Good Eats", frequency : 5 } , { title : "Bikes", frequency : 2 } ], $sort : {frequency : -1} }}})

* Remove
* db.UserData.updateOne({name : "Maria"}, {$pull : { hobbies : {title : "Good Drinks"} }})

db.UserData.updateOne({name : "Chris"}, {$pop : { hobbies : 1 }})

🡪db.UserData.updateOne({name : "Maria"},{$addToSet : {hobbies : { title : "Hiking", frequency : 2 }}})

**INDEX :**

So why would we use an index

and what is index to begin with? An index can speed up our find, update or delete queries,

So all the queries where we are looking for certain documents that should match some criteria. Consider

this find query, here I'm looking for all products where the seller is Max.

Now let's say this is my collection of documents, the products collection, now by default if I don't

have an index on seller set,

mongodb will go ahead and do a so-called collection scan,

now that simply means that mongodb to fulfill this query will go through the entire collection,

look at every single document and see if seller equals Max

and as you can imagine for very large collections with thousands or millions of documents, this can take

a while and we'll see this in practice with an example in a second.

So this is the default approach mongodb takes or the only approach it can take when you have no index

set up in order to retrieve maybe two documents out of your thousand documents.

Now you can create an index though, an index is not a replacement for a collection but an addition you

could say,

so you would create an index for the seller key of the products collection here and that index then

exists additionally to the collection and the index is essentially an ordered list of all the values

that are placed or stored in the seller key for all the documents.

So it's not an ordered list of the documents, just of the values for the field for which you created that

index

and it's not just an ordered list of the values, of course every value, every item in the index has a

pointer to the full document

it belongs to.

Now this allows mongodb to do a so-called index scan to fulfill this query, which means it sees that

for seller, such an index exists and it therefore simply goes to that seller index and can quickly jump

to the right values because there, unlike for the collection,

it knows that the values are sorted by that key,

so it doesn't have to look at the first three records if it's only looking for records starting with

M or to be precise, records equal to Max.

So it can very efficiently go through that index and then find the matching products because of that ordering

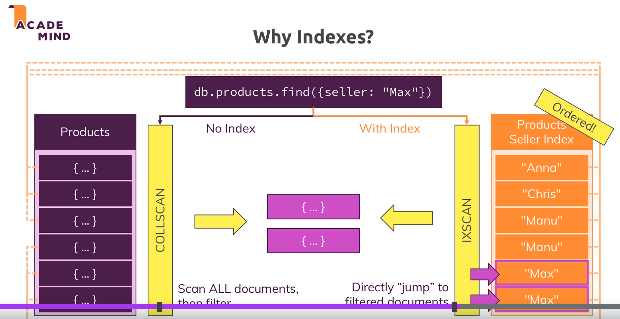
and because of that pointer, every element in this index has, so mongodb finds the value for this query

and then finds the related documents it can return this,

so it's this direct access that mongodb can use here and that speeds up your queries.

So this is the end how an index works and it also answers the question why you would use one because

creating such indexes can drastically speed up your queries.



However you also shouldn't overdo it with the indexes because you could of course think well ok I got

my product selection and every product has ID, name, age and hobby

and then I could simply create indexes for all fields and I will have the best performance ever, right?

Because no matter for what I look, I got an index for that.

Well this might speed up your find queries,

that is correct, indexes on all fields would speed up find queries because you can query for every field

efficiently but an index does not come for free,

you will pay some performance cost on inserts because that extra index that has to be maintained needs

to be updated with every insert,

makes sense right

because we have an ordered list of elements with pointers to the documents.

So if you add a new document, you also have to add a new element to the index and that might sound simple

and it won't take super long but if you've got 10 indexes for your documents in your collection

and you have to update all 10 indexes for every insert, then you might quickly run into issues because

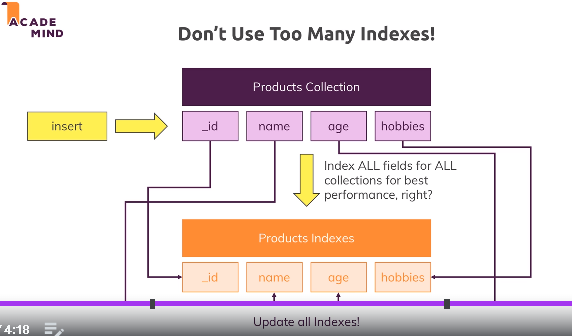
you'll have to do a lot of work for all these fields, for every insert and for every update too. Therefore,

indexes don't come for free and you really have to find out which indexes makes sense and which indexes

don't and this is exactly what I will also do in this module with you.

I will walk you through all the different index types that exist in mongodb and I will also show

you how you can measure whether an index makes sense or does not make sense.



Explain():

explain works for find, update, delete not for insert, so it basically works for the methods where you well narrow down documents, where you find documents

input :

db.PersonData.explain().find({"dob.age": {$gt : 60}})

output :

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.PersonData',

indexFilterSet: false,

parsedQuery: {

'dob.age': {

'$gt': 60

}

},

queryHash: '08155E45',

planCacheKey: '08155E45',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'COLLSCAN',

filter: {

'dob.age': {

'$gt': 60

}

},

direction: 'forward'

},

rejectedPlans: []

},

command: {

find: 'PersonData',

filter: {

'dob.age': {

'$gt': 60

}

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

Now here we get the detailed description of what mongodb did and how it derived our results,

mongodb thinks in so-called plans and plans are simply alternatives it considers for executing that query

and in the end it will find a winning plan

and I'll come back to how mongodb determines this later

and that winning plan is essentially what it did to get our results

and you see here, the winning plan was to do a full collection scan.

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

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.PersonData',

indexFilterSet: false,

parsedQuery: {

'dob.age': {

'$gt': 60

}

},

queryHash: '08155E45',

planCacheKey: '08155E45',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'COLLSCAN',

filter: {

'dob.age': {

'$gt': 60

}

},

direction: 'forward'

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 1222,

executionTimeMillis: 8,

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: 'PersonData',

filter: {

'dob.age': {

'$gt': 60

}

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

db.PersonData.createIndex({"dob.age" : 1})

Now an index is defined as a document here

and the first value,

the key here is the name of the field you want to create an index on

and in my case that is dob.age,

.

So you see you can create indexes on embedded fields just as you could use a normal field,

so you can use top level fields, you can use embedded fields, doesn't matter. Then the value is whether

mongodb should create that list of values in that age field in an ascending or descending order,

so it can sort by assigning or descending order.

If you add a one here, it'll be ascending order,

so lower values come first,

higher values towards the end,

if you add a -1 here, it's descending. What you choose here in the end doesn't matter too much

even if you do sort your results and you sort to the opposite direction, it will still be sped up because

mongodb can traverse that index in both directions,

so you can actually choose what you want here and I'll go for ascending.

Now this created an index, you see we had one before,

we'll see which index that was in a second,

now we have two.

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

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.PersonData',

indexFilterSet: false,

parsedQuery: {

'dob.age': {

'$gt': 60

}

},

queryHash: '08155E45',

planCacheKey: '3F4F0F60',

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, inf.0]'

]

}

}

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 1222,

executionTimeMillis: 87,

totalKeysExamined: 1222,

totalDocsExamined: 1222,

executionStages: {

stage: 'FETCH',

nReturned: 1222,

executionTimeMillisEstimate: 0,

works: 1223,

advanced: 1222,

needTime: 0,

needYield: 0,

saveState: 2,

restoreState: 2,

isEOF: 1,

docsExamined: 1222,

alreadyHasObj: 0,

inputStage: {

stage: 'IXSCAN',

nReturned: 1222,

executionTimeMillisEstimate: 0,

works: 1223,

advanced: 1222,

needTime: 0,

needYield: 0,

saveState: 2,

restoreState: 2,

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, inf.0]'

]

},

keysExamined: 1222,

seeks: 1,

dupsTested: 0,

dupsDropped: 0

}

}

},

command: {

find: 'PersonData',

filter: {

'dob.age': {

'$gt': 60

}

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

We also see that there are two execution stages now

and we also see that the first stage, the input stage was an index scan, in the other output, that would

be our winning plan essentially.

So it did not do a full collection scan but instead an index scan

and there you see that it returned 1222 documents or not documents to be precise

but keys in the index with their respective pointers at documents,

so the index scan does not return the documents but just the keys in the index and the pointers to the

documents.

It's the next stage,

the fetch stage which will then take these pointers returned from the index and reach out to the actual

collection and then fetch the real documents from there

and therefore in the end, we see here we had to only look at 1222 keys in our index to reach 1222

documents which are returned.

We also had to look at these documents because the index only has the pointer set to documents,

so the index just narrows down the set,

we still have to go through the collection and get the documents from there to return them in the end

but this sped up our query and this is how an index can help us.

Now before we dive deeper into different types of indexes, let me also show you something interesting

about this dataset which helps you understand indexes a bit better.

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.

drop

db.PersonData.dropIndex({"dob.age":1})

db.PersonData.explain("executionStats").find({"dob.age": {$gt : 20}})

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.PersonData',

indexFilterSet: false,

parsedQuery: {

'dob.age': {

'$gt': 20

}

},

queryHash: '08155E45',

planCacheKey: '08155E45',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'COLLSCAN',

filter: {

'dob.age': {

'$gt': 20

}

},

direction: 'forward'

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 5000,

executionTimeMillis: 13,

totalKeysExamined: 0,

totalDocsExamined: 5000,

executionStages: {

stage: 'COLLSCAN',

filter: {

'dob.age': {

'$gt': 20

}

},

nReturned: 5000,

executionTimeMillisEstimate: 1,

works: 5002,

advanced: 5000,

needTime: 1,

needYield: 0,

saveState: 5,

restoreState: 5,

isEOF: 1,

direction: 'forward',

docsExamined: 5000

}

},

command: {

find: 'PersonData',

filter: {

'dob.age': {

'$gt': 20

}

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

Now something interesting happens if we get rid of that index which we can do by reaching out to our

database, to our collection and then running drop index and now here we specify the exact same document we

used for creating it,

so dob.age and then one and now you see that index was removed

and now let's run the same query again or explain it,

now that we have no index. What you'll see is that this now actually is faster, we get execution time of 6 milli

seconds.

Now the reason why this is faster is that we save the step of going for the index,

if you have a query that will return a large portion or the majority of your documents, an index can

actually be slower because you then just have an extra step to go through your almost entire

index list and then you have to go to the collection and get all these documents,

so you then just have an extra step because if you do a full collection scan, it can be faster

and it certainly is if you return all elements but even for the majority it would be faster because

with a full collection scan, you already have all the documents in memory and then an index doesn't offer you

any more because that just is an extra step.

Instead here we got all the documents in memory,

we would have needed to go to the documents anyways to fetch them from the pointers the index gives

us,

so now we already have them and since we need most of them, this is now faster.

So if you have queries that regularly return the majority or all of your documents, an index will not

really help you there,

it might even slow down the execution

and that is important to keep in mind as a first restriction that you need to know when planning your

queries.

If you have a dataset where your queries typically only return fractions, like 10 or 20 percent or lower

than that of the documents, then indexes will almost certainly always speed it up.

If you've got a lot of queries that give you back all the documents or close to all the documents,

indexes can't do that much work for you and logically, that makes sense because the idea of index is to

quickly let you get to a narrow subset of your document list and not to the majority of that.

db.PersonData.createIndex({gender : 1})

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

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.PersonData',

indexFilterSet: false,

parsedQuery: {

gender: {

'$eq': 'male'

}

},

queryHash: '025F03D3',

planCacheKey: '4B09AA47',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'FETCH',

inputStage: {

stage: 'IXSCAN',

keyPattern: {

gender: 1

},

indexName: 'gender\_1',

isMultiKey: false,

multiKeyPaths: {

gender: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

gender: [

'["male", "male"]'

]

}

}

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 2435,

executionTimeMillis: 31,

totalKeysExamined: 2435,

totalDocsExamined: 2435,

executionStages: {

stage: 'FETCH',

nReturned: 2435,

executionTimeMillisEstimate: 22,

works: 2436,

advanced: 2435,

needTime: 0,

needYield: 0,

saveState: 3,

restoreState: 3,

isEOF: 1,

docsExamined: 2435,

alreadyHasObj: 0,

inputStage: {

stage: 'IXSCAN',

nReturned: 2435,

executionTimeMillisEstimate: 22,

works: 2436,

advanced: 2435,

needTime: 0,

needYield: 0,

saveState: 3,

restoreState: 3,

isEOF: 1,

keyPattern: {

gender: 1

},

indexName: 'gender\_1',

isMultiKey: false,

multiKeyPaths: {

gender: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

gender: [

'["male", "male"]'

]

},

keysExamined: 2435,

seeks: 1,

dupsTested: 0,

dupsDropped: 0

}

}

},

command: {

find: 'PersonData',

filter: {

gender: 'male'

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

db.PersonData.createIndex({gender : 1, "dob.age" : 1})

db.PersonData.explain("executionStats").find({gender : "male", "dob.age" : {$gt : 35}})

db.PersonData.createIndex({"dob.age" : 1, gender : 1})

sort index :

db.PersonData.explain().find({"dob.age": 35}).sort({gender : 1})

db.PersonData.getIndexes()

db.PersonData.createIndex({email : 1}, {unique : true})

So unique indexes can help you as a developer

ensure data consistency and help you avoid duplicate data for fields that you need to have unique,

so the id, \_id is unique by default but you have other use cases like maybe that email here

too

and the unique index is a great way for you to not just speed up your find queries but also to guarantee

that you have unique values for that given field in that collection.

db.PersonData.dropIndex({"dob.age": 1, gender: 1})

Partial Index :

Having an index on the date of birth age field then

might make sense

but the problem of course is that you have a lot of values in your index that you never actually query

for. Now

your index will still be efficient but it will be unnecessarily big

and of course an index

also eats up size on your disk,

additionally the bigger the index is, well the more performance certain queries will take nonetheless.

So if you know that certain values will not be looked at or only very very rarely and you would be fine

using a collection scan if that happens, you can actually create a partial index where you only add the

values you're regularly going to look at,

db.PersonData.createIndex({"dob.age" : 1}, {partialFilterExpress : {"dob.age": {$gt :60}}})



but in the second argument to the create index function here, unique true, this also works.

But now let me insert some new document, db users insert one and there, I'll insert Anna

also without an email. And now I get an error, I get a duplicate key error because that non-existing e-mail

for which I have an index is treated as a duplicate key because now I have a no value, so no value stored

twice.

So that is an interesting behavior but some behavior you just have to be aware of, mongodb treats

nonexisting values still as values in your index, so as a not there value

you could say, as a null value and therefore if you have two documents with no value for an indexed field

and that index is unique, you will get this error.

****

true, so this filter simply says I only want to add elements into my index where the email field exists

and this will avoid the case of having a clash with unique. If I run this, it works,

if I now try to insert Anna without an email,

this also works.

**Time to Live :**

Now the last interesting index option I want to show to you is a time to live index and that's a really

cool kind of index that can be very helpful for a lot of applications where you have self-destroying

data,

let's say sessions of users where you want to clear their data after some duration or anything like

that.

Let's create a new sessions collection here, you can of course come up with any example you want and

in there, I'll insert one document and that one document will receive some data

which is some random text

and more importantly, it will get a createdAt key and you can name this however you want, doesn't have to

be named createdAt. Now

this will receive a new date and it will receive new date like this which will just be the current timestamp

or the current date,

so if I look into sessions with find pretty, well then I see this ISO date was created by mongodb and

this is the current date and time.

Now let me add a time to live index here, for that,

I'll go to my sessions and create an index

and now I will create that index on the createdAt field.

Now first of all, you could of course create a normal ascending index here too and you can order dates

just like you can text and numbers but now I'll get rid of this because I want to add this index a bit differently.

Instead of adding it as I did before in ascending order,

I was still add it like this but I will add an additional argument here to configure this index and

there, I will add the expire after seconds field.

This is a special feature mongodb offers and that only works on date indexes or on date fields, on other

fields it will just be ignored,

you could add it but it will be ignored

and there I could say every element should be removed after 10 seconds.

Now if I hit enter here and I look into my collection, you'll see this value is still there because it

was there before the index was added and the index does not delete elements in hindsight

but if I insert a new element here, you see it's now in the database

but let's now wait for 10 seconds because that is the duration

I did specify and thereafter you see,

actually both were deleted.

The reason for that is

adding a new element basically triggered mongodb to now re-evaluate the entire collection,

so also the already existing documents and see whether this field which is indexed fulfills this criteria

of only being valid for 10 seconds

and therefore then it also reconsidered the existing documents,

it just doesn't do that right

when you add the index but it does do it whenever you add a new element.

So this can be very useful because it allows you to maintain a collection of documents which destroy

themselves after a certain time span and for a lot of applications,

this can be useful things like for example as I just said, session data for a user of your website

or maybe in an online shop where you want to clear the cart after one day,

so whenever you have a use case where data should clean up itself, you don't need to write a complex

script for that,

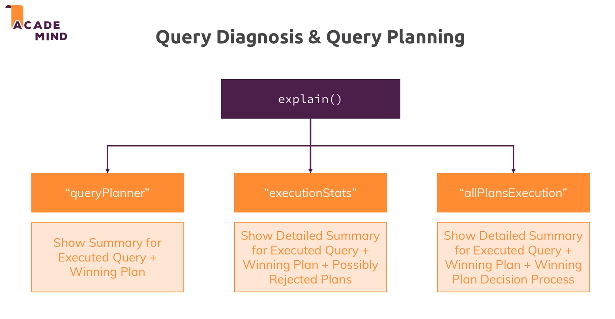
you can use a time to live index with that expiry after seconds addition we added in our index.

Important to know here by the way,

you can only use that on single field indexes,

it does not work on compound indexes and as I mentioned, it works on date objects.

db.session.insertOne({data : "Hello World", createdAt : new Date()})

****

Now we had a detailed look at what indexes do and how you can create your own indexes and I can only

encourage you to play around with that to get a better feeling for your different options and how indexes

work

but in order to play around and to understand if an index is worth the effort, you need to know how to

diagnose your queries and I already did show you the explain method for this.

The important part here is that you can execute it like this or pass query planner as an argument to

get that default minimal output where it simply tells you the winning plan and not much else.

Or you use execution stats, what

we did a couple of times in this module already to see detailed summary outputs and see information

about the winning plan and possibly rejected plan and also some information about how long it took.

There also is the all plans execution option which also show shows detailed summaries and which also

gives you more information about how the winning plan was chosen.

Now we'll have a look at all three of them throughout this module again. For determining whether a query is efficient,

it's obviously interesting to look at the milliseconds process time and also compare this to a solution

where you don't use an index, so that you'll also have a look whether index scan really beats a collection

scan which it typically does though

but I did already show you some use cases in cases where you fetch almost everything where the index

scan can be slower

and another important measure is that you compare the number of keys in the index, that is what it means

in the output are examined, how many documents that are examined and how many documents that are returned

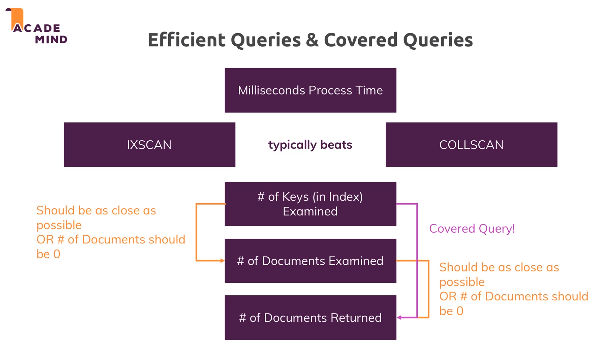
and the keys and document should be close together and documents or documents, examine the documents returned

should be closed or documents should be zero so that it looked at zero documents

and when would this be the case? In a so-called covered query

and now what is a covered query?

Well let me show it to you.

****

db.customers.explain("executionStats").find({name : "Max"})

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.customers',

indexFilterSet: false,

parsedQuery: {

name: {

'$eq': 'Max'

}

},

queryHash: '64908032',

planCacheKey: 'A6C0273F',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'FETCH',

inputStage: {

stage: 'IXSCAN',

keyPattern: {

name: 1

},

indexName: 'name\_1',

isMultiKey: false,

multiKeyPaths: {

name: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

name: [

'["Max", "Max"]'

]

}

}

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 1,

executionTimeMillis: 10,

totalKeysExamined: 1,

totalDocsExamined: 1,

executionStages: {

stage: 'FETCH',

nReturned: 1,

executionTimeMillisEstimate: 10,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

docsExamined: 1,

alreadyHasObj: 0,

inputStage: {

stage: 'IXSCAN',

nReturned: 1,

executionTimeMillisEstimate: 10,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

keyPattern: {

name: 1

},

indexName: 'name\_1',

isMultiKey: false,

multiKeyPaths: {

name: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

name: [

'["Max", "Max"]'

]

},

keysExamined: 1,

seeks: 1,

dupsTested: 0,

dupsDropped: 0

}

}

},

command: {

find: 'customers',

filter: {

name: 'Max'

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

**Covered Query :**

With projection in find()

db.customers.explain("executionStats").find({name : "Max"}, {\_id : 0, name: 1})

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.customers',

indexFilterSet: false,

parsedQuery: {

name: {

'$eq': 'Max'

}

},

queryHash: '4B1D5B79',

planCacheKey: 'A40C523D',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'PROJECTION\_COVERED',

transformBy: {

\_id: 0,

name: 1

},

inputStage: {

stage: 'IXSCAN',

keyPattern: {

name: 1

},

indexName: 'name\_1',

isMultiKey: false,

multiKeyPaths: {

name: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

name: [

'["Max", "Max"]'

]

}

}

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 1,

executionTimeMillis: 0,

totalKeysExamined: 1,

totalDocsExamined: 0,

executionStages: {

stage: 'PROJECTION\_COVERED',

nReturned: 1,

executionTimeMillisEstimate: 0,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

transformBy: {

\_id: 0,

name: 1

},

inputStage: {

stage: 'IXSCAN',

nReturned: 1,

executionTimeMillisEstimate: 0,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

keyPattern: {

name: 1

},

indexName: 'name\_1',

isMultiKey: false,

multiKeyPaths: {

name: []

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

name: [

'["Max", "Max"]'

]

},

keysExamined: 1,

seeks: 1,

dupsTested: 0,

dupsDropped: 0

}

}

},

command: {

find: 'customers',

filter: {

name: 'Max'

},

projection: {

\_id: 0,

name: 1

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

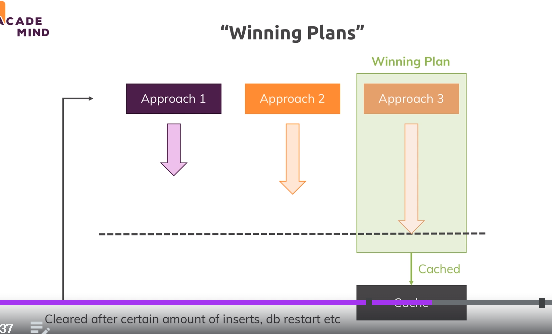
}

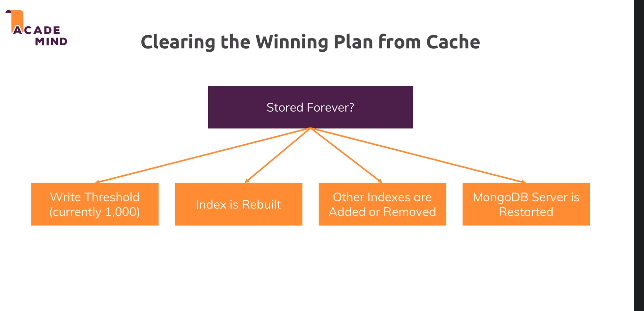
db.customers.getIndexes()

**Compound Index :**

db.customers.createIndex({age : 1, name : 1})

db.customers.explain().find({name : "Max", age : 30})

****

****

Now what we can see is that we now also have a rejected plan and the rejected plan was to use an

index scan just on the name index, that is the single field name index I created.

Of course it made sense for mongodb to also consider this because we have that name index,

we had a query that included a search for name,

so it made sense to consider that name index

but of course since we have another index which simply fits this query better, this compound index,

mongodb chose to use this one as a winning plan and rejected this one.

Now this is interesting to know,

the question of course is how exactly does mongodb figure out which plan is better? For this, mongodb

uses an approach where it simply

first of all looks for indexes that could help you with the query at hand,

so in our last lecture and example, we had the name single field index and age and name as a compound index.

Since our query, our find method included a look for name, mongodb automatically derived that

both the name single field index and the age name compound index could be helpful.

So it came up with two approaches, for other scenarios you might of course have more approaches,

so let's say we have three approaches, mongodb then simply let's those approaches race against each

other but not for the full dataset

but it sets a certain winning condition, right now that should be one hundred documents.

So it looks who's the first to find 100 documents and of course one of the approaches will be the fastest

to reach that threshold, mongodb will then basically use that approach for the real query

and of course that would be cumbersome if it would have to do this for every find method, for every

query you send to the database because that obviously costs a little bit of performance.

Therefore mongodb caches

this winning plan for this exact query,

so for exactly the query you send with the fields you were looking for and the values for the fields

you were looking for,

so it caches this plan as the winner plan for this kind of query. And for future queries that are looking

exactly equal, it uses this winning plan, for future queries that look different, that use different values

or different keys,

it will of course race again and find a winning plan for that type of query.

Now of course this cache is not there forever,

it is cleared after a certain amount of inserts or a db restart.

To be precise, instead of being stored forever,

this winning plan is removed from cache after you wrote a certain amount of documents to that collection,

currently there should be one thousand,

so after you added that many documents, mongodb says I don't know if the current winning plan will still

win because the collection changed a lot,

so I should reconsider.

The same happens if you rebuild the index, so if you drop it and recreate it for example,

it also gets deleted if you add other indexes because that new index could be better,

so the cache for indexes or winning plans I should say gets cleared when you add new indexes or when

you restart the mongodb server,

then this also gets reset.

So this is how mongodb derives

the winning plan and how it stores it

and it's interesting for you as a developer to regularly check your queries, your find or update or delete

queries and see what mongodb actually does,

if it uses indexes efficiently, if maybe a new index should be added, something you can do on your own

if you own the database or you pass that information to your db administrator

or if you maybe need to adjust the query. Maybe you're always fetching data that you don't really need

and you could get a covered query if you just would project the data you need

which happens to be the data stored in the index.

That is why you as a developer need to know how indexes work because either you need to create them

on your own in your next project on which you work alone or because you can optimize your queries or

tell your administrator to optimize the indexes.

Now with that information gained, let's have a look at the last level of verbosity the explain method offers

and that is that we can add all plans execution as a value here.

Now what this will do is it will give us a bunch of output because here, we get detailed statistics for all

plans,

also the rejected plans. So there we can see in detail how would an index scan on our compound index

perform but then also what would happen if we had that index scan on the name index,

how long would that take,

how many documents would it there consider documents and how many keys would it consider,

how long would it take,

here we have zero because it's so super fast but for larger collections, you would of course see a difference.

And with that, you can get detailed analytics on the different indexes and queries and possible ways

of running your query and therefore you should have all the tools you need to optimize your queries

and your indexes.

db.customers.explain("allPlansExecution").find({name : "Max", age : 30})

**{**

**explainVersion: '1',**

**queryPlanner: {**

**namespace: 'user.customers',**

**indexFilterSet: false,**

**parsedQuery: {**

**'$and': [**

**{**

**age: {**

**'$eq': 30**

**}**

**},**

**{**

**name: {**

**'$eq': 'Max'**

**}**

**}**

**]**

**},**

**queryHash: 'A0EEAF22',**

**planCacheKey: 'A1B2B9E5',**

**maxIndexedOrSolutionsReached: false,**

**maxIndexedAndSolutionsReached: false,**

**maxScansToExplodeReached: false,**

**winningPlan: {**

**stage: 'FETCH',**

**inputStage: {**

**stage: 'IXSCAN',**

**keyPattern: {**

**age: 1,**

**name: 1**

**},**

**indexName: 'age\_1\_name\_1',**

**isMultiKey: false,**

**multiKeyPaths: {**

**age: [],**

**name: []**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**age: [**

**'[30, 30]'**

**],**

**name: [**

**'["Max", "Max"]'**

**]**

**}**

**}**

**},**

**rejectedPlans: [**

**{**

**stage: 'FETCH',**

**filter: {**

**age: {**

**'$eq': 30**

**}**

**},**

**inputStage: {**

**stage: 'IXSCAN',**

**keyPattern: {**

**name: 1**

**},**

**indexName: 'name\_1',**

**isMultiKey: false,**

**multiKeyPaths: {**

**name: []**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**name: [**

**'["Max", "Max"]'**

**]**

**}**

**}**

**}**

**]**

**},**

**executionStats: {**

**executionSuccess: true,**

**nReturned: 0,**

**executionTimeMillis: 0,**

**totalKeysExamined: 0,**

**totalDocsExamined: 0,**

**executionStages: {**

**stage: 'FETCH',**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**works: 2,**

**advanced: 0,**

**needTime: 0,**

**needYield: 0,**

**saveState: 0,**

**restoreState: 0,**

**isEOF: 1,**

**docsExamined: 0,**

**alreadyHasObj: 0,**

**inputStage: {**

**stage: 'IXSCAN',**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**works: 1,**

**advanced: 0,**

**needTime: 0,**

**needYield: 0,**

**saveState: 0,**

**restoreState: 0,**

**isEOF: 1,**

**keyPattern: {**

**age: 1,**

**name: 1**

**},**

**indexName: 'age\_1\_name\_1',**

**isMultiKey: false,**

**multiKeyPaths: {**

**age: [],**

**name: []**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**age: [**

**'[30, 30]'**

**],**

**name: [**

**'["Max", "Max"]'**

**]**

**},**

**keysExamined: 0,**

**seeks: 1,**

**dupsTested: 0,**

**dupsDropped: 0**

**}**

**},**

**allPlansExecution: [**

**{**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**totalKeysExamined: 0,**

**totalDocsExamined: 0,**

**score: 2.0002,**

**executionStages: {**

**stage: 'FETCH',**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**works: 1,**

**advanced: 0,**

**needTime: 0,**

**needYield: 0,**

**saveState: 0,**

**restoreState: 0,**

**isEOF: 1,**

**docsExamined: 0,**

**alreadyHasObj: 0,**

**inputStage: {**

**stage: 'IXSCAN',**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**works: 1,**

**advanced: 0,**

**needTime: 0,**

**needYield: 0,**

**saveState: 0,**

**restoreState: 0,**

**isEOF: 1,**

**keyPattern: {**

**age: 1,**

**name: 1**

**},**

**indexName: 'age\_1\_name\_1',**

**isMultiKey: false,**

**multiKeyPaths: {**

**age: [],**

**name: []**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**age: [**

**'[30, 30]'**

**],**

**name: [**

**'["Max", "Max"]'**

**]**

**},**

**keysExamined: 0,**

**seeks: 1,**

**dupsTested: 0,**

**dupsDropped: 0**

**}**

**}**

**},**

**{**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**totalKeysExamined: 1,**

**totalDocsExamined: 1,**

**score: 1.0002,**

**executionStages: {**

**stage: 'FETCH',**

**filter: {**

**age: {**

**'$eq': 30**

**}**

**},**

**nReturned: 0,**

**executionTimeMillisEstimate: 0,**

**works: 1,**

**advanced: 0,**

**needTime: 1,**

**needYield: 0,**

**saveState: 1,**

**restoreState: 0,**

**isEOF: 0,**

**docsExamined: 1,**

**alreadyHasObj: 0,**

**inputStage: {**

**stage: 'IXSCAN',**

**nReturned: 1,**

**executionTimeMillisEstimate: 0,**

**works: 1,**

**advanced: 1,**

**needTime: 0,**

**needYield: 0,**

**saveState: 1,**

**restoreState: 0,**

**isEOF: 0,**

**keyPattern: {**

**name: 1**

**},**

**indexName: 'name\_1',**

**isMultiKey: false,**

**multiKeyPaths: {**

**name: []**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**name: [**

**'["Max", "Max"]'**

**]**

**},**

**keysExamined: 1,**

**seeks: 1,**

**dupsTested: 0,**

**dupsDropped: 0**

**}**

**}**

**}**

**]**

**},**

**command: {**

**find: 'customers',**

**filter: {**

**name: 'Max',**

**age: 30**

**},**

**'$db': 'user'**

**},**

**serverInfo: {**

**host: 'Uchiha\_Madara',**

**port: 27017,**

**version: '6.0.6',**

**gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'**

**},**

**serverParameters: {**

**internalQueryFacetBufferSizeBytes: 104857600,**

**internalQueryFacetMaxOutputDocSizeBytes: 104857600,**

**internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,**

**internalDocumentSourceGroupMaxMemoryBytes: 104857600,**

**internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,**

**internalQueryProhibitBlockingMergeOnMongoS: 0,**

**internalQueryMaxAddToSetBytes: 104857600,**

**internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600**

**},**

**ok: 1**

**}**

**Mutli Index :**

db.contacts.explain("executionStats").find({hobbies : "Sports"})

**{**

**explainVersion: '1',**

**queryPlanner: {**

**namespace: 'user.contacts',**

**indexFilterSet: false,**

**parsedQuery: {**

**hobbies: {**

**'$eq': 'Sports'**

**}**

**},**

**queryHash: '8463837D',**

**planCacheKey: 'CE1580AB',**

**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: [**

**'["Sports", "Sports"]'**

**]**

**}**

**}**

**},**

**rejectedPlans: []**

**},**

**executionStats: {**

**executionSuccess: true,**

**nReturned: 1,**

**executionTimeMillis: 41,**

**totalKeysExamined: 1,**

**totalDocsExamined: 1,**

**executionStages: {**

**stage: 'FETCH',**

**nReturned: 1,**

**executionTimeMillisEstimate: 31,**

**works: 2,**

**advanced: 1,**

**needTime: 0,**

**needYield: 0,**

**saveState: 1,**

**restoreState: 1,**

**isEOF: 1,**

**docsExamined: 1,**

**alreadyHasObj: 0,**

**inputStage: {**

**stage: 'IXSCAN',**

**nReturned: 1,**

**executionTimeMillisEstimate: 31,**

**works: 2,**

**advanced: 1,**

**needTime: 0,**

**needYield: 0,**

**saveState: 1,**

**restoreState: 1,**

**isEOF: 1,**

**keyPattern: {**

**hobbies: 1**

**},**

**indexName: 'hobbies\_1',**

**isMultiKey: true,**

**multiKeyPaths: {**

**hobbies: [**

**'hobbies'**

**]**

**},**

**isUnique: false,**

**isSparse: false,**

**isPartial: false,**

**indexVersion: 2,**

**direction: 'forward',**

**indexBounds: {**

**hobbies: [**

**'["Sports", "Sports"]'**

**]**

**},**

**keysExamined: 1,**

**seeks: 1,**

**dupsTested: 1,**

**dupsDropped: 0**

**}**

**}**

**},**

**command: {**

**find: 'contacts',**

**filter: {**

**hobbies: 'Sports'**

**},**

**'$db': 'user'**

**},**

**serverInfo: {**

**host: 'Uchiha\_Madara',**

**port: 27017,**

**version: '6.0.6',**

**gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'**

**},**

**serverParameters: {**

**internalQueryFacetBufferSizeBytes: 104857600,**

**internalQueryFacetMaxOutputDocSizeBytes: 104857600,**

**internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,**

**internalDocumentSourceGroupMaxMemoryBytes: 104857600,**

**internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,**

**internalQueryProhibitBlockingMergeOnMongoS: 0,**

**internalQueryMaxAddToSetBytes: 104857600,**

**internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600**

**},**

**ok: 1**

**}**

**Array base indexing :**

db.contacts.explain("executionStats").find({"address.street" : "Secondary Street"})

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.contacts',

indexFilterSet: false,

parsedQuery: {

'address.street': {

'$eq': 'Secondary Street'

}

},

queryHash: '911BAE14',

planCacheKey: '911BAE14',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'COLLSCAN',

filter: {

'address.street': {

'$eq': 'Secondary Street'

}

},

direction: 'forward'

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 0,

executionTimeMillis: 0,

totalKeysExamined: 0,

totalDocsExamined: 1,

executionStages: {

stage: 'COLLSCAN',

filter: {

'address.street': {

'$eq': 'Secondary Street'

}

},

nReturned: 0,

executionTimeMillisEstimate: 0,

works: 3,

advanced: 0,

needTime: 2,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

direction: 'forward',

docsExamined: 1

}

},

command: {

find: 'contacts',

filter: {

'address.street': 'Secondary Street'

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

db.contacts.explain("executionStats").find({addresses : {street : "Secondary Street"}})

{

explainVersion: '1',

queryPlanner: {

namespace: 'user.contacts',

indexFilterSet: false,

parsedQuery: {

addresses: {

'$eq': {

street: 'Secondary Street'

}

}

},

queryHash: '355C72B9',

planCacheKey: '4B18A14A',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'FETCH',

inputStage: {

stage: 'IXSCAN',

keyPattern: {

addresses: 1

},

indexName: 'addresses\_1',

isMultiKey: true,

multiKeyPaths: {

addresses: [

'addresses'

]

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

addresses: [

'[{ street: "Secondary Street" }, { street: "Secondary Street" }]'

]

}

}

},

rejectedPlans: []

},

executionStats: {

executionSuccess: true,

nReturned: 1,

executionTimeMillis: 10,

totalKeysExamined: 1,

totalDocsExamined: 1,

executionStages: {

stage: 'FETCH',

nReturned: 1,

executionTimeMillisEstimate: 10,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

docsExamined: 1,

alreadyHasObj: 0,

inputStage: {

stage: 'IXSCAN',

nReturned: 1,

executionTimeMillisEstimate: 10,

works: 2,

advanced: 1,

needTime: 0,

needYield: 0,

saveState: 0,

restoreState: 0,

isEOF: 1,

keyPattern: {

addresses: 1

},

indexName: 'addresses\_1',

isMultiKey: true,

multiKeyPaths: {

addresses: [

'addresses'

]

},

isUnique: false,

isSparse: false,

isPartial: false,

indexVersion: 2,

direction: 'forward',

indexBounds: {

addresses: [

'[{ street: "Secondary Street" }, { street: "Secondary Street" }]'

]

},

keysExamined: 1,

seeks: 1,

dupsTested: 1,

dupsDropped: 0

}

}

},

command: {

find: 'contacts',

filter: {

addresses: {

street: 'Secondary Street'

}

},

'$db': 'user'

},

serverInfo: {

host: 'Uchiha\_Madara',

port: 27017,

version: '6.0.6',

gitVersion: '26b4851a412cc8b9b4a18cdb6cd0f9f642e06aa7'

},

serverParameters: {

internalQueryFacetBufferSizeBytes: 104857600,

internalQueryFacetMaxOutputDocSizeBytes: 104857600,

internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,

internalDocumentSourceGroupMaxMemoryBytes: 104857600,

internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,

internalQueryProhibitBlockingMergeOnMongoS: 0,

internalQueryMaxAddToSetBytes: 104857600,

internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600

},

ok: 1

}

**TEXT INDEX :**

db.products.createIndex({description : "text"})

db.products.find({$text : {$search : "awesome"}})

db.products.find({$text : {$search : "\"awesome book\""}})

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

**Text Index Sorting :**

db.products.find({$text : {$search : "awesome t-shirt"}}, {score : {$meta : "textScore"}})

db.products.find({$text : {$search : "awesome t-shirt"}}, {score : {$meta : "textScore"}}).sort({score : {$meta : "textScore"}})

DROP TEXT INDEX : db.products.dropIndex("description\_text")

Combined Text Indes : db.products.createIndex({title : "text", description : "text"}) : title\_text\_description\_text

db.products.find({$text : {$search : "ship"}})

excluding second word :

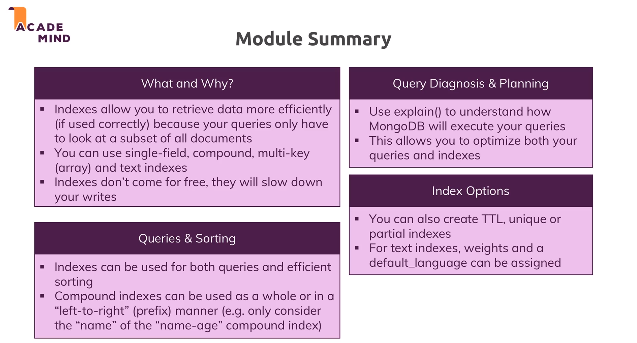
db.products.find({$text : {$search : "awesome -t-shirt"}})

db.products.dropIndex("title\_text\_description\_text")

change language :

db.products.createIndex({title : "text", description : "text"}, {default\_language : "english"})

147 : video rewatch

****

Helpful Articles/ Docs:

* More on partialFilterExpressions: <https://docs.mongodb.com/manual/core/index-partial/>
* Supported default\_languages: <https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages>
* How to use different languages in the same index: <https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/#create-a-text-index-for-a-collection-in-multiple-languages>

**GEO:**

db.places.insertOne({ name : "Jeju-Si", location : {type : "Point", coordinates : [ 126.5271242 , 33.5049796 ]}})

Finding Geometry :

db.places.find({location : {$near : { $geometry : { type:"Point", coordinates : [126.6682778, 33.4956446] } }}})

Geo Index :

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

db.places.find({location : {$near : { $geometry : { type:"Point", coordinates : [126.6682778, 33.4956446] }, $maxDistance : 9000, $minDistance : 6000 }}})