MongoDb using Docker

docker run --name test-mongo -dit -p 27017:27017 --rm mongo:4.4.1

Docker run--name test-mongo, you can call it whatever you want, I just call it test mongo because I'm not creative. -dit, that's for detach interactive and tty that's what the dit stands for. -p this this is gonna be what port you're gonna be opening from your Docker container running on your computer to your local computer. So we're opening the port 27017, outside. So that's gonna be the ones gonna be available my host computer, and it's gonna be opening 27017 which is the default port for Mongo, interior on the inside

So you can kinda map those ports if you want them to be different. I don't, so that's why I'm saying that. The --rm just means if, when I'm done with it just remove cleanup automatically afterwards. And then the name of the container that will be pulling off of Docker Hub is Mongo.

And then the :4.4.1 we're pulling the version 4.4.1. So if you run this, you should just get something like this. This is the hash of what's running in side of docker. So if I type Docker PS right now, you can see I have one container running which is this one you can see these numbers match up.

docker exec -it test-mongo mongo

This will start the mongo in docker

Now we're inside of MongoDB. And now we're able to start writing quit all database clients, or all databases have these kind of clients, we can get into the command line and start doing queries. So let's look at one of these. If I do show dbs like that, you can see here I have three different databases. One called admin, one called configure and called local. This is what Mongo creates for you by default without you having to do anything. So Mongo has two kind of key concepts here. One is called a database and one is called a collection.

MongoDb without Docker

Just open the cmd and type mongo this will start the mongo server and you can start using mongoDb.

show dbs; it'll show you all the available database .

We're gonna be using a database called shop because we're gonna have multiple different collections inside of this database. So it's a group of collections. You'll find frequently your application will only ever use one database, that's okay. If you have a bigger application has a bunch of, this is users, and this is movies, and this is TV shows and you need to separate those out, you could have multiple databases which would then have multiple collections. It's all about organizing your data in a useful way. But the idea is that you wanna have like data together. So I'm gonna say use shop, like that. That's gonna be the name of my database

because it's gonna be a shop database. And you can see here now I've switched to DB adoption.
That didn't exist before, but again, Mongo is dynamic in nature, so it just rolls with it, rise like you want something called shop? Here you go. I don't actually have something like that, but you can have it now. So let's talk about collections. Collections are just basically like a grab bag of documents, right?
A document is gonna be like a record or if you're thinking about a spreadsheet, it's gonna be like a row, right? So one individual entry into your database is going to be a document and the table of those rows is going to be your collection. So yeah, it's just a grab bag of objects, you can think of it as basically JavaScript's objects.
To create a new database you have write use first then database name.
Ex:- use shop
switched to db shop
To insert the data:-

```
db.tvs.insertOne({tvName:"LG TV", company:"LG", price: 1500000})
{
	"acknowledged" : true,
	"insertedId" : ObjectId("62c3d5b503e7bd879353a382")
}
db.tvs.count()
1

db.stats()
{
	"db" : "shop",
	"collections" : 1,
	"views" : 0,
	"objects" : 1,
	"avgObjSize" : 71,
	"dataSize" : 71,
	"storageSize" : 20480,
	"indexes" : 1,
	"indexSize" : 20480,
	"totalSize" : 40960,
	"scaleFactor" : 1,
	"fsUsedSize" : 150764134400,
	"fsTotalSize" : 250048671744,
	"ok" : 1
}
```

So, we've now put one document into our collection. Let's get it out. So if I say db.tvs.findOne() , just like that. There's only one document in the database. So or in the collection, so it'll get that one right So I just gave it an empty query because I knew it's going to find something and that something is going to be literally the only one there.

```
db.tvs.findOne()
{
	"_id" : ObjectId("62c3d5b503e7bd879353a382"),
	"tvName" : "LG TV",
	"company" : "LG",
	"price" : 1500000
```

}

findOne is gonna it's gonna go into your database, it's gonna try and find the first thing that it can find that matches that particular query and then it will return that. Even if you have 100 things in there. It's just going to ignore them. If you want to find all of them Do you just do find, and this is going to give you an iterator.

```
db.tvs.findOne({company:"LG"})
{
	"_id" : ObjectId("62c3d5b503e7bd879353a382"),
	"tvName" : "LG TV",
	"company" : "LG",
	"price" : 1500000
}
```

it's going to give you an iterator to iterate all over all those. In this particular case, there's only one to iterate over so it doesn't do anything special about that. It just returns out to you. But notice here, this underscore ID thing, this is the one that MongoDB creates for you so that's it's hash of what its ideas.

```
db.tvs.find({company:"LG"})
{ "id" : ObjectId("62c3d5b503e7bd879353a382"), "tvName" : "LG TV", "company" : "LG", "price" : 1500000 }
```

```
db.tvs.insertMany(
  Array.from({ length: 10000 }).map((_, index) => ({
    name: [
      "10 inch tv",
      "20 inch tv ",
      "25 inch tv ",
       "30 inch tv ",
       "35 inch tv ",
       "40 inch tv ",
       "45 inch tv ",
```

```
        "50 inch tv ",
        "55 inch tv ",
        "60inch tv ",
        "65 inch tv ",
      ][index % 9],
      type: ["tv", "phone tv", "android tv", "black and white
tv"][index % 4],
      price: (index % 18) + 1,
      company: [
        "LG",
        "MI",
        "samsung",
        "vivo",
        "Dell",
        "HP",
        "jio",
      ][index % 7],
      index: index,
    }))
);
```

Basically this makes 10000 tv's objects in an array. Because we used modulo, everyone will get the same objects every time. Now run you're `db.tvs.count()` and see what you get. You should get back that you have 10,000 items in your database.

```
db.tvs.count()
10001
db.tvs.findOne()
{
      "_id" : ObjectId("62c3d5b503e7bd879353a382"),
      "tvName" : "LG TV",
      "company" : "LG",
      "price" : 1500000
}
```

So now I wonder if I do DB dot tv stuff. Find one You can see here so I think it goes by created date so I think it'll find the first item that matches it. And it will find the first one that was created. I think is the ordering on find one But I'm not sure it's ambiguous and you never want to rely on ambiguous behavior in a that that sounds like something that would crash your app like two months into it and you wouldn't know why.

```
  db.tvs.find()
{ "_id" : ObjectId("62c3d5b503e7bd879353a382"), "tvName" : "LG TV", "company" : "LG", "price" : 1500000
}
{ "_id" : ObjectId("62c5013247af947de18c8c1b"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"LG", "index" : 0 }
{ "_id" : ObjectId("62c5013247af947de18c8c1c"), "name" : "20 inch tv", "type" : "phone tv", "price" : 2,
"company" : "MI", "index" : 1 }
{ "_id" : ObjectId("62c5013247af947de18c8c1d"), "name" : "25 inch tv", "type" : "android tv", "price" : 3,
"company" : "samsung", "index" : 2 }
{ "_id" : ObjectId("62c5013247af947de18c8c1e"), "name" : "30 inch tv", "type" : "black and white tv", "price" :
4, "company" : "vivo", "index" : 3 }
{ "_id" : ObjectId("62c5013247af947de18c8c1f"), "name" : "35 inch tv", "type" : "tv", "price" : 5, "company" :
"Dell", "index" : 4 }
{ "_id" : ObjectId("62c5013247af947de18c8c20"), "name" : "40 inch tv", "type" : "phone tv", "price" : 6,
"company" : "HP", "index" : 5 }
{ "_id" : ObjectId("62c5013247af947de18c8c21"), "name" : "45 inch tv", "type" : "android tv", "price" : 7,
"company" : "jio", "index" : 6 }
{ "_id" : ObjectId("62c5013247af947de18c8c22"), "name" : "50 inch tv", "type" : "black and white tv", "price" :
8, "company" : "LG", "index" : 7 }
{ "_id" : ObjectId("62c5013247af947de18c8c23"), "name" : "55 inch tv", "type" : "tv", "price" : 9, "company" :
"MI", "index" : 8 }
{ "_id" : ObjectId("62c5013247af947de18c8c24"), "name" : "10 inch tv", "type" : "phone tv", "price" : 10,
"company" : "samsung", "index" : 9 }
{ "_id" : ObjectId("62c5013247af947de18c8c25"), "name" : "20 inch tv", "type" : "android tv", "price" : 11,
"company" : "vivo", "index" : 10 }
{ "_id" : ObjectId("62c5013247af947de18c8c26"), "name" : "25 inch tv", "type" : "black and white tv", "price" :
12, "company" : "Dell", "index" : 11 }
{ "_id" : ObjectId("62c5013247af947de18c8c27"), "name" : "30 inch tv", "type" : "tv", "price" : 13, "company"
: "HP", "index" : 12 }
{ "_id" : ObjectId("62c5013247af947de18c8c28"), "name" : "35 inch tv", "type" : "phone tv", "price" : 14,
"company" : "jio", "index" : 13 }
{ "_id" : ObjectId("62c5013247af947de18c8c29"), "name" : "40 inch tv", "type" : "android tv", "price" : 15,
"company" : "LG", "index" : 14 }
{ "_id" : ObjectId("62c5013247af947de18c8c2a"), "name" : "45 inch tv", "type" : "black and white tv", "price" :
16, "company" : "MI", "index" : 15 }
{ "_id" : ObjectId("62c5013247af947de18c8c2b"), "name" : "50 inch tv", "type" : "tv", "price" : 17, "company"
: "samsung", "index" : 16 }
{ "_id" : ObjectId("62c5013247af947de18c8c2c"), "name" : "55 inch tv", "type" : "phone tv", "price" : 18,
"company" : "vivo", "index" : 17 }
{ "_id" : ObjectId("62c5013247af947de18c8c2d"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "Dell", "index" : 18 }
```

```
Type "it" for more
db.tvs.findOne({index:1337})
{
	"_id" : ObjectId("62c5013247af947de18c9154"),
	"name" : "40 inch tv",
	"type" : "phone tv",
	"price" : 6,
	"company" : "LG",
	"index" : 1337
}
```

So the first one here is find one we've still we saw this one already find one and we found LG TVBut let's do some other kind of interesting queries here. So let's say we wanted to find one of index, you know, 1337. This object here is kind of it's your query object, right?

```
db.tvs.findOne({type:"android tv",price:9})
{
	"_id" : ObjectId("62c5013247af947de18c8c35"),
	"name" : "55 inch tv",
	"type" : "android tv",
	"price" : 9,
	"company" : "HP",
	"index" : 26
}

db.tvs.count({company:"HP", price:9})
80
```

```
db.tvs.find({company:"HP"}).limit(40)
{ "_id" : ObjectId("62c5013247af947de18c8c20"), "name" : "40 inch tv", "type" : "phone tv", "price" : 6, "company" : "HP", "index" : 5 }
{ "_id" : ObjectId("62c5013247af947de18c8c27"), "name" : "30 inch tv", "type" : "tv", "price" : 13, "company" : "HP", "index" : 12 }
{ "_id" : ObjectId("62c5013247af947de18c8c2e"), "name" : "20 inch tv", "type" : "black and white tv", "price" : 2, "company" : "HP", "index" : 19 }
{ "_id" : ObjectId("62c5013247af947de18c8c35"), "name" : "55 inch tv", "type" : "android tv", "price" : 9, "company" : "HP", "index" : 26 }
{ "_id" : ObjectId("62c5013247af947de18c8c3c"), "name" : "45 inch tv", "type" : "phone tv", "price" : 16, "company" : "HP", "index" : 33 }
{ "_id" : ObjectId("62c5013247af947de18c8c43"), "name" : "35 inch tv", "type" : "tv", "price" : 5, "company" : "HP", "index" : 40 }
{ "_id" : ObjectId("62c5013247af947de18c8c4a"), "name" : "25 inch tv", "type" : "black and white tv", "price" : 12, "company" : "HP", "index" : 47 }
{ "_id" : ObjectId("62c5013247af947de18c8c51"), "name" : "10 inch tv", "type" : "android tv", "price" : 1, "company" : "HP", "index" : 54 }
{ "_id" : ObjectId("62c5013247af947de18c8c58"), "name" : "50 inch tv", "type" : "phone tv", "price" : 8, "company" : "HP", "index" : 61 }
{ "_id" : ObjectId("62c5013247af947de18c8c5f"), "name" : "40 inch tv", "type" : "tv", "price" : 15, "company" : "HP", "index" : 68 }
{ "_id" : ObjectId("62c5013247af947de18c8c66"), "name" : "30 inch tv", "type" : "black and white tv", "price" : 4, "company" : "HP", "index" : 75 }
```

{ "_id" : ObjectId("62c5013247af947de18c8c6d"), "name" : "20 inch tv", "type" : "android tv", "price" : 11, "company" : "HP", "index" : 82 }
{ "_id" : ObjectId("62c5013247af947de18c8c74"), "name" : "55 inch tv", "type" : "phone tv", "price" : 18, "company" : "HP", "index" : 89 }
{ "_id" : ObjectId("62c5013247af947de18c8c7b"), "name" : "45 inch tv", "type" : "tv", "price" : 7, "company" : "HP", "index" : 96 }
{ "_id" : ObjectId("62c5013247af947de18c8c82"), "name" : "35 inch tv", "type" : "black and white tv", "price" : 14, "company" : "HP", "index" : 103 }
{ "_id" : ObjectId("62c5013247af947de18c8c89"), "name" : "25 inch tv", "type" : "android tv", "price" : 3, "company" : "HP", "index" : 110 }
{ "_id" : ObjectId("62c5013247af947de18c8c90"), "name" : "10 inch tv", "type" : "phone tv", "price" : 10, "company" : "HP", "index" : 117 }
{ "_id" : ObjectId("62c5013247af947de18c8c97"), "name" : "50 inch tv", "type" : "tv", "price" : 17, "company" : "HP", "index" : 124 }
{ "_id" : ObjectId("62c5013247af947de18c8c9e"), "name" : "40 inch tv", "type" : "black and white tv", "price" : 6, "company" : "HP", "index" : 131 }
{ "_id" : ObjectId("62c5013247af947de18c8ca5"), "name" : "30 inch tv", "type" : "android tv", "price" : 13, "company" : "HP", "index" : 138 }
Type "it" for more
> it
{ "_id" : ObjectId("62c5013247af947de18c8cac"), "name" : "20 inch tv", "type" : "phone tv", "price" : 2, "company" : "HP", "index" : 145 }
{ "_id" : ObjectId("62c5013247af947de18c8cb3"), "name" : "55 inch tv", "type" : "tv", "price" : 9, "company" : "HP", "index" : 152 }
{ "_id" : ObjectId("62c5013247af947de18c8cba"), "name" : "45 inch tv", "type" : "black and white tv", "price" : 16, "company" : "HP", "index" : 159 }
{ "_id" : ObjectId("62c5013247af947de18c8cc1"), "name" : "35 inch tv", "type" : "android tv", "price" : 5, "company" : "HP", "index" : 166 }
{ "_id" : ObjectId("62c5013247af947de18c8cc8"), "name" : "25 inch tv", "type" : "phone tv", "price" : 12, "company" : "HP", "index" : 173 }
{ "_id" : ObjectId("62c5013247af947de18c8ccf"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" : "HP", "index" : 180 }
{ "_id" : ObjectId("62c5013247af947de18c8cd6"), "name" : "50 inch tv", "type" : "black and white tv", "price" : 8, "company" : "HP", "index" : 187 }
{ "_id" : ObjectId("62c5013247af947de18c8cdd"), "name" : "40 inch tv", "type" : "android tv", "price" : 15, "company" : "HP", "index" : 194 }
{ "_id" : ObjectId("62c5013247af947de18c8ce4"), "name" : "30 inch tv", "type" : "phone tv", "price" : 4, "company" : "HP", "index" : 201 }
{ "_id" : ObjectId("62c5013247af947de18c8ceb"), "name" : "20 inch tv", "type" : "tv", "price" : 11, "company" : "HP", "index" : 208 }
{ "_id" : ObjectId("62c5013247af947de18c8cf2"), "name" : "55 inch tv", "type" : "black and white tv", "price" : 18, "company" : "HP", "index" : 215 }
{ "_id" : ObjectId("62c5013247af947de18c8cf9"), "name" : "45 inch tv", "type" : "android tv", "price" : 7, "company" : "HP", "index" : 222 }
{ "_id" : ObjectId("62c5013247af947de18c8d00"), "name" : "35 inch tv", "type" : "phone tv", "price" : 14, "company" : "HP", "index" : 229 }
{ "_id" : ObjectId("62c5013247af947de18c8d07"), "name" : "25 inch tv", "type" : "tv", "price" : 3, "company" : "HP", "index" : 236 }
{ "_id" : ObjectId("62c5013247af947de18c8d0e"), "name" : "10 inch tv", "type" : "black and white tv", "price" : 10, "company" : "HP", "index" : 243 }
{ "_id" : ObjectId("62c5013247af947de18c8d15"), "name" : "50 inch tv", "type" : "android tv", "price" : 17, "company" : "HP", "index" : 250 }
{ "_id" : ObjectId("62c5013247af947de18c8d1c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 6, "company" : "HP", "index" : 257 }
{ "_id" : ObjectId("62c5013247af947de18c8d23"), "name" : "30 inch tv", "type" : "tv", "price" : 13, "company" : "HP", "index" : 264 }
{ "_id" : ObjectId("62c5013247af947de18c8d2a"), "name" : "20 inch tv", "type" : "black and white tv", "price" : 2, "company" : "HP", "index" : 271 }
{ "_id" : ObjectId("62c5013247af947de18c8d31"), "name" : "55 inch tv", "type" : "android tv", "price" : 9, "company" : "HP", "index" : 278 }

> no cursor

you can change if it gives you five records at a time or five documents or 100 documents at a time you can tell to you do all of them.

Let's say that You wanted to get a typed doc and you only wanted to get like the first 40 records. I can imagine you're implementing search for like a tv shop website.

You could say give me only the first, 40 records 40 documents in the collection here, limit 40 That's count. Yeah, sorry, find. So now if I do IT again, notice it's actually done now it's not actually asking me for. So you type it for more. Now it's done.

```
db.tvs.find({company:"MI"}).limit(40).toArray()
[
	{
		"_id" : ObjectId("62c5013247af947de18c8c1c"),
		"name" : "20 inch tv",
		"type" : "phone tv",
		"price" : 2,
		"company" : "MI",
		"index" : 1
	},
	{
		"_id" : ObjectId("62c5013247af947de18c8c23"),
		"name" : "55 inch tv",
		"type" : "tv",
		"price" : 9,
		"company" : "MI",
		"index" : 8
	},
	{
		"_id" : ObjectId("62c5013247af947de18c8c2a"),
		"name" : "45 inch tv",
		"type" : "black and white tv",
		"price" : 16,
		"company" : "MI",
		"index" : 15
	},
	{
		"_id" : ObjectId("62c5013247af947de18c8c31"),
		"name" : "35 inch tv",
		"type" : "android tv",
		"price" : 5,
		"company" : "MI",
		"index" : 22
	},
	{
		"_id" : ObjectId("62c5013247af947de18c8c38"),
		"name" : "25 inch tv",
		"type" : "phone tv",
		"price" : 12,
		"company" : "MI",
		"index" : 29
	},
	{
		"_id" : ObjectId("62c5013247af947de18c8c3f"),
		"name" : "10 inch tv",
		"type" : "tv",
		"price" : 1,
		"company" : "MI",
		"index" : 36
	},
	{
```

```
        "_id" : ObjectId("62c5013247af947de18c8c46"),
        "name" : "50 inch tv",
        "type" : "black and white tv",
        "price" : 8,
        "company" : "MI",
        "index" : 43
},
{
        "_id" : ObjectId("62c5013247af947de18c8c4d"),
        "name" : "40 inch tv",
        "type" : "android tv",
        "price" : 15,
        "company" : "MI",
        "index" : 50
},
{
        "_id" : ObjectId("62c5013247af947de18c8c54"),
        "name" : "30 inch tv",
        "type" : "phone tv",
        "price" : 4,
        "company" : "MI",
        "index" : 57
},
{
        "_id" : ObjectId("62c5013247af947de18c8c5b"),
        "name" : "20 inch tv",
        "type" : "tv",
        "price" : 11,
        "company" : "MI",
        "index" : 64
},
{
        "_id" : ObjectId("62c5013247af947de18c8c62"),
        "name" : "55 inch tv",
        "type" : "black and white tv",
        "price" : 18,
        "company" : "MI",
        "index" : 71
},
{
        "_id" : ObjectId("62c5013247af947de18c8c69"),
        "name" : "45 inch tv",
        "type" : "android tv",
        "price" : 7,
        "company" : "MI",
        "index" : 78
},
{
        "_id" : ObjectId("62c5013247af947de18c8c70"),
        "name" : "35 inch tv",
        "type" : "phone tv",
        "price" : 14,
        "company" : "MI",
        "index" : 85
},
{
        "_id" : ObjectId("62c5013247af947de18c8c77"),
        "name" : "25 inch tv",
        "type" : "tv",
        "price" : 3,
```

```
        "company" : "MI",
        "index" : 92
},
{
        "_id" : ObjectId("62c5013247af947de18c8c7e"),
        "name" : "10 inch tv",
        "type" : "black and white tv",
        "price" : 10,
        "company" : "MI",
        "index" : 99
},
{
        "_id" : ObjectId("62c5013247af947de18c8c85"),
        "name" : "50 inch tv",
        "type" : "android tv",
        "price" : 17,
        "company" : "MI",
        "index" : 106
},
{
        "_id" : ObjectId("62c5013247af947de18c8c8c"),
        "name" : "40 inch tv",
        "type" : "phone tv",
        "price" : 6,
        "company" : "MI",
        "index" : 113
},
{
        "_id" : ObjectId("62c5013247af947de18c8c93"),
        "name" : "30 inch tv",
        "type" : "tv",
        "price" : 13,
        "company" : "MI",
        "index" : 120
},
{
        "_id" : ObjectId("62c5013247af947de18c8c9a"),
        "name" : "20 inch tv",
        "type" : "black and white tv",
        "price" : 2,
        "company" : "MI",
        "index" : 127
},
{
        "_id" : ObjectId("62c5013247af947de18c8ca1"),
        "name" : "55 inch tv",
        "type" : "android tv",
        "price" : 9,
        "company" : "MI",
        "index" : 134
},
{
        "_id" : ObjectId("62c5013247af947de18c8ca8"),
        "name" : "45 inch tv",
        "type" : "phone tv",
        "price" : 16,
        "company" : "MI",
        "index" : 141
},
{
```

```
        "_id" : ObjectId("62c5013247af947de18c8caf"),
        "name" : "35 inch tv",
        "type" : "tv",
        "price" : 5,
        "company" : "MI",
        "index" : 148
},
{
        "_id" : ObjectId("62c5013247af947de18c8cb6"),
        "name" : "25 inch tv",
        "type" : "black and white tv",
        "price" : 12,
        "company" : "MI",
        "index" : 155
},
{
        "_id" : ObjectId("62c5013247af947de18c8cbd"),
        "name" : "10 inch tv",
        "type" : "android tv",
        "price" : 1,
        "company" : "MI",
        "index" : 162
},
{
        "_id" : ObjectId("62c5013247af947de18c8cc4"),
        "name" : "50 inch tv",
        "type" : "phone tv",
        "price" : 8,
        "company" : "MI",
        "index" : 169
},
{
        "_id" : ObjectId("62c5013247af947de18c8ccb"),
        "name" : "40 inch tv",
        "type" : "tv",
        "price" : 15,
        "company" : "MI",
        "index" : 176
},
{
        "_id" : ObjectId("62c5013247af947de18c8cd2"),
        "name" : "30 inch tv",
        "type" : "black and white tv",
        "price" : 4,
        "company" : "MI",
        "index" : 183
},
{
        "_id" : ObjectId("62c5013247af947de18c8cd9"),
        "name" : "20 inch tv",
        "type" : "android tv",
        "price" : 11,
        "company" : "MI",
        "index" : 190
},
{
        "_id" : ObjectId("62c5013247af947de18c8ce0"),
        "name" : "55 inch tv",
        "type" : "phone tv",
        "price" : 18,
```

```
        "company" : "MI",
        "index" : 197
},
{
        "_id" : ObjectId("62c5013247af947de18c8ce7"),
        "name" : "45 inch tv",
        "type" : "tv",
        "price" : 7,
        "company" : "MI",
        "index" : 204
},
{
        "_id" : ObjectId("62c5013247af947de18c8cee"),
        "name" : "35 inch tv",
        "type" : "black and white tv",
        "price" : 14,
        "company" : "MI",
        "index" : 211
},
{
        "_id" : ObjectId("62c5013247af947de18c8cf5"),
        "name" : "25 inch tv",
        "type" : "android tv",
        "price" : 3,
        "company" : "MI",
        "index" : 218
},
{
        "_id" : ObjectId("62c5013247af947de18c8cfc"),
        "name" : "10 inch tv",
        "type" : "phone tv",
        "price" : 10,
        "company" : "MI",
        "index" : 225
},
{
        "_id" : ObjectId("62c5013247af947de18c8d03"),
        "name" : "50 inch tv",
        "type" : "tv",
        "price" : 17,
        "company" : "MI",
        "index" : 232
},
{
        "_id" : ObjectId("62c5013247af947de18c8d0a"),
        "name" : "40 inch tv",
        "type" : "black and white tv",
        "price" : 6,
        "company" : "MI",
        "index" : 239
},
{
        "_id" : ObjectId("62c5013247af947de18c8d11"),
        "name" : "30 inch tv",
        "type" : "android tv",
        "price" : 13,
        "company" : "MI",
        "index" : 246
},
{
```

```
        "_id" : ObjectId("62c5013247af947de18c8d18"),
        "name" : "20 inch tv",
        "type" : "phone tv",
        "price" : 2,
        "company" : "MI",
        "index" : 253
    },
    {
        "_id" : ObjectId("62c5013247af947de18c8d1f"),
        "name" : "55 inch tv",
        "type" : "tv",
        "price" : 9,
        "company" : "MI",
        "index" : 260
    },
    {
        "_id" : ObjectId("62c5013247af947de18c8d26"),
        "name" : "45 inch tv",
        "type" : "black and white tv",
        "price" : 16,
        "company" : "MI",
        "index" : 267
    },
    {
        "_id" : ObjectId("62c5013247af947de18c8d2d"),
        "name" : "35 inch tv",
        "type" : "android tv",
        "price" : 5,
        "company" : "MI",
        "index" : 274
    }
]

db.tvs.count({type:"android tv", price:{$gt:12}}); //greater then
832
> db.tvs.count({type:"android tv",price:{$gte:12}}); greater then equal to
832
                                                    {$eq:12}}); equal to
                                                    {$ne:12}}); not equal to
                                                    {$lt:12}}); less than



db.tvs.count({type:"phone tv", $and: [{price:{$gte:4}},{price:{$lte: 8}}]})
834
```

Let's talk about logical operators. So we just talked about query operators with like non equals and equals and greater than and greater than or equal to. So we're gonna be talking now about, let's say you needed two things to be true at the same time. So let's say you wanted to combine two logical operators and say like, I want to find tvs between the price of four and eight so it would be both of those statements to be true, the way they would do that is with a logical operator.

```
db.tvs.find({type:"android tv"}).sort({price:-1}).limit(5)
{ "_id" : ObjectId("62c5013247af947de18c8c85"), "name" : "50 inch tv", "type" : "android tv", "price" : 17,
"company" : "MI", "index" : 106 }
{ "_id" : ObjectId("62c5013247af947de18c8c61"), "name" : "50 inch tv", "type" : "android tv", "price" : 17,
"company" : "LG", "index" : 70 }
{ "_id" : ObjectId("62c5013247af947de18c8ccd"), "name" : "50 inch tv", "type" : "android tv", "price" : 17,
"company" : "vivo", "index" : 178 }
```

```
{ "_id" : ObjectId("62c5013247af947de18c8ca9"), "name" : "50 inch tv", "type" : "android tv", "price" : 17, "company" : "samsung", "index" : 142 }
{ "_id" : ObjectId("62c5013247af947de18c8c3d"), "name" : "50 inch tv", "type" : "android tv", "price" : 17, "company" : "jio", "index" : 34 }
```

So we are gonna sort by price, and you can either sort by ascending which would be one, or descending which would be negative one. Okay, and then we'll say limit five. And you can see there, we have five android tv here that are all 17. So that's what that dot sort does there for you that allows you to sort by ascending or descending, you can also sort by two things at once.

Projections:-

```
db.tvs.find({type:"tv"},{name:1}).limit(5)
{ "_id" : ObjectId("62c5013247af947de18c8c1b"), "name" : "10 inch tv" }
{ "_id" : ObjectId("62c5013247af947de18c8c1f"), "name" : "35 inch tv" }
{ "_id" : ObjectId("62c5013247af947de18c8c23"), "name" : "55 inch tv" }
{ "_id" : ObjectId("62c5013247af947de18c8c27"), "name" : "30 inch tv" }
{ "_id" : ObjectId("62c5013247af947de18c8c2b"), "name" : "50 inch tv" }
```

```
db.tvs.find({type:"tv"},{name:1,_id:0}).limit(5)
{ "name" : "10 inch tv" }
{ "name" : "35 inch tv" }
{ "name" : "55 inch tv" }
{ "name" : "30 inch tv" }
{ "name" : "50 inch tv" }
```

If I want to not include that, you can actually come in here and specifically disallow that. So if I put 0 there or false, both of those would work. it's not include. So there you go. That way, we actually even disinclude the _ID as well.

```
db.tvs.find({type:"tv"},{_id:0}).limit(5)
{ "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" : "LG", "index" : 0 }
{ "name" : "35 inch tv", "type" : "tv", "price" : 5, "company" : "Dell", "index" : 4 }
{ "name" : "55 inch tv", "type" : "tv", "price" : 9, "company" : "MI", "index" : 8 }
{ "name" : "30 inch tv", "type" : "tv", "price" : 13, "company" : "HP", "index" : 12 }
{ "name" : "50 inch tv", "type" : "tv", "price" : 17, "company" : "samsung", "index" : 16 }
```

I'm gonna do ID: false. So all of those work as well. And another thing that you can do as well, you can actually just disinclude things. So if I do ID false like that, it'll include everything else.
So that's kind of takes a disallow list kind of approach to projection as well that is like, specifically don't include these things, include everything else. So let's say you're querying a user database that has their password hash in it, you could specifically just disinclude that if you don't need that for whatever you're doing
So that's a lot of what's about querying MongoDB. So at this point, you're probably pretty well equipped to just go out there and start reading from MongoDB. The nice thing about learning these databases is 85% of the use case can be learned in about, I don't know how long we've been talking about now, or something like that.

Updating MongoDB

```
db.tvs.updateOne(
... {type:"phone tv", name:"40 inch tv", company:"MI"},
... {$set: {owner: "Samir Akhtar"}}
... );
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.tvs.find({owner:"Samir Akhtar"});
{ "_id" : ObjectId("62c5013247af947de18c8c8c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 6, "company" : "MI", "index" : 113, "owner" : "Samir Akhtar" }
```

we're gonna update one. So what I'm gonna do now is, I'm gonna say db.tvs.updateOne.
So this is gonna work just like find one, where it's going to find that the first one that it encounters, I think it's by create date. It's going to update that one, it's not gonna update anything else. So here, I'm gonna say, update one. And the first thing that you're gonna give it is a query object.
This is the exact same object that you provide to find or find one or find many, right? Or there's no find many, it's just find. So I'm gonna say, find one that's a phontv, that's named 40 inch tv.
This is the exact same object that you provide to find or find one or find many, right? Or there's no find many, it's just find. So I'm gonna say, find one that's a dog, that's named Luna. And you don't have to put quotes there, those quotes are optional, cuz it's JavaScript, right?
Name, 40 inch tv , company,MI. So we're gonna find my 40 inch tv, okay? And then the second object that you can provide to, is you're basically gonna tell MongoDB, how do you want to update this? We wanna set her owner to me, samir akhtar, dollar signs set. So this is going to be basically an object that's gonna do like an object merge, or what is object at assign, if you're familiar with JavaScript, that's how this is gonna work.
So I'm gonna provide this in another object, and I'm gonna give it an owner of samir akhtar, Okay? So let's just kind of break this down, update one, you're gonna provide it a query object and an update object. The first one is going to find the record that you're trying to, or the document that you're trying to update.
The second one is going to tell it how to update it. So if I run this it'll give you acknowledged true, this is like, this was successful, this actually did happen. It matched one thing which is good, cuz that's what I asked for, update one and I modified one thing. So if I ran this again, you would notice that it said, it matched one, but it didn't modify anything, because that was already the way that it was, right?
So now, if I change this from, if I say db.tvs.find, and I searched by owner, samir akhtar. You notice, it's only gonna return one. So a couple things that I've demonstrated here, one, we just showed you how to do update one, right? So that's gonna be updating this one to be samir akhtar.

[00:04:50]
But also notice that, no other tv object or tv document in our collection has an owner. I just made that one up on the flier, that piece of schema, right? So that's what's really cool about MongoDB, is like, if you wanna change the schema of your collection, just do it.
There's no pomp and circumstance to do anything about this. You just go, and you can change things on the fly. It's one of the superpowers of MongoDB, it's also one of its downfalls, cuz it's really easy to mess this up, right? Because it's not going to challenge you, the guard rails are of, you can do whatever you want.
So it's kind of a double edged sword there.

```
db.tvs.updateMany(
... {type:"phone tv"},
... {$inc: {price:1}}
... );
{ "acknowledged" : true, "matchedCount" : 2500, "modifiedCount" : 2500 }
> db.tvs.find({owner:"Samir Akhtar"});
{ "_id" : ObjectId("62c5013247af947de18c8c8c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 113, "owner" : "Samir Akhtar" }
```

So the first thing is we're gonna do is, we're going to do an updateMany. We're gonna give it a query object.
So we're gonna find every phone tv type, phone tv, and then we're gonna give it a query object. And we're gonna say, when to use a special operator for updates. And we're going increment all of their price by one, right? Because some of them will be af price: 1, some will be price: 2, some will be price 3, and we wanted to just add 1 to all of those.
So we say, $inc for increment And then we're gonna find price, and we're going to want to increase it by 1. You can increase it by 2, right? If you wanted to So now, if we run that, you can see that we matched 2500 documents in the collection, and we've modified all of those.
So actually, if it's find again that dog, you can see here, Luna's age was age eight here, and now, you can see phone tv price is price 7, because we incremented all of their price by 1, make sense? There's a bunch of those.

There's, I put a link in here as well, you could just take a look here.

Name	Description
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.

Name	Description
`$inc`	Increments the value of the field by the specified amount.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$set`	Sets the value of a field in a document.
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
`$unset`	Removes the specified field from a document.

Array

Operators

Name	Description
`$`	Acts as a placeholder to update the first element that matches the query condition.
`$[]`	Acts as a placeholder to update all elements in an array for the documents that match the query condition.
`$[<identifier>]`	Acts as a placeholder to update all elements that match the `arrayFilters` condition for the documents that match the query condition.
`$addToSet`	Adds elements to an array only if they do not already exist in the set.
`$pop`	Removes the first or last item of an array.
`$pull`	Removes all array elements that match a specified query.
`$push`	Adds an item to an array.
`$pullAll`	Removes all matching values from an array.

Modifiers

Name	Description
`$each`	Modifies the `$push` and `$addToSet` operators to append multiple items for array updates.
`$position`	Modifies the `$push` operator to specify the position in the array to add elements.
`$slice`	Modifies the `$push` operator to limit the size of updated arrays.
`$sort`	Modifies the `$push` operator to reorder documents stored in an array.

Bitwise

Name	Description
`$bit`	Performs bitwise `AND`, `OR`, and `XOR` updates of integer values.

```
db.tvs.updateOne(
... ... {
... ... type: "mobile",
... ... name: "rdmi",
... ... company:"MI",
... ... },
... ... {
... ... $set: {
... ... type: "mobile",
...     name:"redmi",
...     company:"MI",
...     price: 500000,
...     index: 10000,
...     owner: "samiran",
... },
... },
... {upsert: true,
... }
... );
{
        "acknowledged" : true,
        "matchedCount" : 0,
        "modifiedCount" : 0,
        "upsertedId" : ObjectId("62c86c5eccb4517c21a49c45")
}
> db.tvs.findOne({name:"redmi"});
{
        "_id" : ObjectId("62c86c5eccb4517c21a49c45"),
        "company" : "MI",
        "name" : "redmi",
        "type" : "mobile",
        "index" : 10000,
        "owner" : "samiran",
        "price" : 500000
}
```

So let's talk about upsert, which is a ridiculous word, but it is one that you'll hear a lot in talking about databases. Upsert is the idea, is like, if you find something, update it.

If it's not there, insert it, does that make sense? So it's kind of this atomic transaction, or this atomic query that allows you to change something if it exists. If not, then go ahead and update it. So let's go ahead and run one of those really quick. So we're gonna say, DB.tvs.updateOne.

And let's say we wanna put in here, Samiran tv. The tvs name is redmi,

So type tv name, redmi. You'll notice that we don't have a tv named redmi in our database right now. And the company is a MI, which I had to go look up, So that's the thing that we're gonna query for, that query will not match anything on a database.

So what we're gonna do here is, we're gonna give it a set object. So dollar signs set, and because it's an upsert, we wanna provide it the whole object, because if nothing matches that tv, then it's going to take that set object and make that the new object.

So you have to provide it with everything else. Also will be lacking that in the new object. So type mobile name, redmi. company, MI, price, 500000. Index, we'll just give it a random index. So I'm gonna give it 20,000 and the owner is Samiran

so that's our set object. Then the last thing is, we're gonna give it one more parameter to this update one, this is like a configuration ones.

We can provide different sorts of configurations to this query. We're just gonna give it an upsert:true. That basically says, if you don't find it, insert it.

So we're querying for this, it's not gonna find it. We're gonna try and set it to this. So if this did exist, this would get updated, because it's not going to exist, it's going to get inserted. So you can see here, match 0, modified 0, upserted ID. This means it actually did get inserted into our collection.

So now, if we just said DB.tvs.findpets, find name, redmi, we would find, Find one.

Delete Documents

```
db.tvs.deleteMany({type:"tv", company:"LG"});
{ "acknowledged" : true, "deletedCount" : 358 }
```

db.tvs.deleteMany. Let's delete all tv of company LG , so you just give it a query object type tv and you give it a company, LG, I mean, you're doing a delete many so you wanna be very careful of what you're gonna do with the delete many because you could delete things that you don't want to.

But in the end, you can see here this ended up deleting 358 documents from our collection.

```
db.tvs.findOneAndDelete({name:"40 inch tv", type:"phone tv"});
{
	"_id" : ObjectId("62c5013247af947de18c8c20"),
	"name" : "40 inch tv",
	"type" : "phone tv",
	"price" : 7,
	"company" : "HP",
	"index" : 5
}
```

Let's say name 40 inch tv, type phone tv.So what this does is that's different from a delete. Notice when we did a delete many here, it just says I deleted a bunch of stuff, it's gone with find one and delete what's it's gonna do, it's actually going to find it, return to that thing, and then also delete it right.

So in this particular case, I got the Phyto object back from MongoDB, but it also deleted this object. So this object is now deleted in our database. So that's what find one and updates gonna do it's gonna return to you that object or the the document before you update it so you can actually see it before you update it.

Sometimes that's useful, sometimes you need to see it before you update it or any want that to be the same thing you wanna find it and updated at the same time. So yeah, that's find 1 and update, find 1 and replace, and find one and delete the name of those three different methods.

Indexes in MongoDb

```
db.tvs.find({name:"10 inch tv"})
{ "_id" : ObjectId("62c5013247af947de18c8c24"), "name" : "10 inch tv", "type" : "phone tv", "price" : 11,
"company" : "samsung", "index" : 9 }
{ "_id" : ObjectId("62c5013247af947de18c8c2d"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "Dell", "index" : 18 }
{ "_id" : ObjectId("62c5013247af947de18c8c36"), "name" : "10 inch tv", "type" : "black and white tv", "price" :
10, "company" : "jio", "index" : 27 }
{ "_id" : ObjectId("62c5013247af947de18c8c3f"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"MI", "index" : 36 }
{ "_id" : ObjectId("62c5013247af947de18c8c48"), "name" : "10 inch tv", "type" : "phone tv", "price" : 11,
"company" : "vivo", "index" : 45 }
{ "_id" : ObjectId("62c5013247af947de18c8c51"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "HP", "index" : 54 }
{ "_id" : ObjectId("62c5013247af947de18c8c5a"), "name" : "10 inch tv", "type" : "black and white tv", "price" :
10, "company" : "LG", "index" : 63 }
{ "_id" : ObjectId("62c5013247af947de18c8c63"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"samsung", "index" : 72 }
{ "_id" : ObjectId("62c5013247af947de18c8c6c"), "name" : "10 inch tv", "type" : "phone tv", "price" : 11,
"company" : "Dell", "index" : 81 }
{ "_id" : ObjectId("62c5013247af947de18c8c75"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "jio", "index" : 90 }
{ "_id" : ObjectId("62c5013247af947de18c8c7e"), "name" : "10 inch tv", "type" : "black and white tv", "price" :
10, "company" : "MI", "index" : 99 }
{ "_id" : ObjectId("62c5013247af947de18c8c87"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"vivo", "index" : 108 }
{ "_id" : ObjectId("62c5013247af947de18c8c90"), "name" : "10 inch tv", "type" : "phone tv", "price" : 11,
"company" : "HP", "index" : 117 }
{ "_id" : ObjectId("62c5013247af947de18c8c99"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "LG", "index" : 126 }
{ "_id" : ObjectId("62c5013247af947de18c8ca2"), "name" : "10 inch tv", "type" : "black and white tv", "price" :
10, "company" : "samsung", "index" : 135 }
{ "_id" : ObjectId("62c5013247af947de18c8cab"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"Dell", "index" : 144 }
{ "_id" : ObjectId("62c5013247af947de18c8cb4"), "name" : "10 inch tv", "type" : "phone tv", "price" : 11,
"company" : "jio", "index" : 153 }
{ "_id" : ObjectId("62c5013247af947de18c8cbd"), "name" : "10 inch tv", "type" : "android tv", "price" : 1,
"company" : "MI", "index" : 162 }
{ "_id" : ObjectId("62c5013247af947de18c8cc6"), "name" : "10 inch tv", "type" : "black and white tv", "price" :
10, "company" : "vivo", "index" : 171 }
{ "_id" : ObjectId("62c5013247af947de18c8ccf"), "name" : "10 inch tv", "type" : "tv", "price" : 1, "company" :
"HP", "index" : 180 }
Type "it" for more

> db.tvs.find({name:"10 inch tv"}).explain("executionStats")
{
        "explainVersion" : "1",
        "queryPlanner" : {
                "namespace" : "shop.tvs",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "name" : {
                                "$eq" : "10 inch tv"
                        }
```

```
        },
        "maxIndexedOrSolutionsReached" : false,
        "maxIndexedAndSolutionsReached" : false,
        "maxScansToExplodeReached" : false,
        "winningPlan" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "name" : {
                                "$eq" : "10 inch tv"
                        }
                },
                "direction" : "forward"
        },
        "rejectedPlans" : [ ]
},
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1072,
        "executionTimeMillis" : 115,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 9643,
        "executionStages" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "name" : {
                                "$eq" : "10 inch tv"
                        }
                },
                "nReturned" : 1072,
                "executionTimeMillisEstimate" : 2,
                "works" : 9645,
                "advanced" : 1072,
                "needTime" : 8572,
                "needYield" : 0,
                "saveState" : 10,
                "restoreState" : 10,
                "isEOF" : 1,
                "direction" : "forward",
                "docsExamined" : 9643
        }
},
"command" : {
        "find" : "tvs",
        "filter" : {
                "name" : "10 inch tv"
        },
        "$db" : "shop"
},
"serverInfo" : {
        "host" : "DESKTOP-CC8A0TA",
        "port" : 27017,
        "version" : "5.0.9",
        "gitVersion" : "6f7dae919422dcd7f4892c10ff20cdc721ad00e6"
},
"serverParameters" : {
        "internalQueryFacetBufferSizeBytes" : 104857600,
        "internalQueryFacetMaxOutputDocSizeBytes" : 104857600,
        "internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,
        "internalDocumentSourceGroupMaxMemoryBytes" : 104857600,
        "internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,
```

```
            "internalQueryProhibitBlockingMergeOnMongoS" : 0,
            "internalQueryMaxAddToSetBytes" : 104857600,
            "internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600
        },
        "ok" : 1
}
```

So this is actually gonna pass back to from Mongo and say, here's how I'm going to find that. So, it does what's called a execution or a query planner, you'll hear those terms thrown around quite a bit. So, the first thing it does is say, all right, here's the query that I'm looking for.

I'm looking for name, where it's equal to 10 inch tv. And then it has a winning plan. Sometimes the database will try multiple different strategies to try and figure out okay, I can search this way, or I can search this way, or I can consider this way. In this case, there's only one way to find it.

It's gonna do what's called a call scan. And this is your worst nightmare right here call scan is like, worst case scenario. I'm going to look at literally every item in this or every document in this collection to find all of your pet's name to find out, this is not what you wanna see.

If this is a query that you run a lot, you need to index this, that you don't see Colescott if you run this like once a day, that's fine, right? Once a day, you can take a performance hit. It doesn't really matter, but if you're running this 10 times a second, you, need to index this yesterday.

Okay, so that this is the part that I want to be looking at here is the winning plan. And you can see here, the call scan is exactly what you're trying to avoid. So here are the execution stats. This is actually gonna go through and tell you what it did, how long it took.

And you can see here, this is, again, we have 9,644 items in the collection and it looked at all 9,644 items in the collection. This is not what you wanna see. So, how can we fix that? Well, we're gonna do that by creating an index. So the first thing we're gonna do is, we're gonna say db.tvs.create index name colon 1.

And what we're doing is we're creating a simple index on name. And that's it. Now, something else I probably need to throw out there. Don't just log on to your server this is actually a fairly heavy procedure to do. So don't just do this in production. Like this can cause downtime.

This can be very expensive. Again, we're doing it for 9,600 and some odd items in a collection. Not that big a deal but 4 million, this could take down your entire app.

```
db.tvs.count({name:"10 inch tv"})
1072
> db.tvs.getIndexes()
[ { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ]
```

db.tvs.get indexes and you can see here we have one on name you will always have one an ID is always indexed just by default. And yeah, that's how you do indexes. So you can do compound indexes, so let's say you're frequently querying by both maybe age and pet at the same time.

You can actually make a compound index that uses both of those things. You kinda just have to understand the shape of your data that it needs to be in. And how you're gonna query it. Normally I, kind of wait a little bit to figure out what indexes I wanna do and kind of see how I'm using the database and how the code is written.

```
> db.tvs.find({index:10000})
{ "_id" : ObjectId("62c86c5eccb4517c21a49c45"), "company" : "MI", "name" : "redmi", "type" : "mobile",
"index" : 10000, "owner" : "samiran", "price" : 500000 }
> db.tvs.createIndex({index:1}, {unique: 1})
{
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "createdCollectionAutomatically" : false,
        "ok" : 1
}
> db.tvs.find({index:10000})
{ "_id" : ObjectId("62c86c5eccb4517c21a49c45"), "company" : "MI", "name" : "redmi", "type" : "mobile",
"index" : 10000, "owner" : "samiran", "price" : 500000 }
> db.tvs.insertOne({index:10000})
WriteError({
        "index" : 0,
        "code" : 11000,
        "errmsg" : "E11000 duplicate key error collection: shop.tvs index: index_1 dup key: { index: 10000.0 }",
        "op" : {
                "_id" : ObjectId("62c92de50479502c3963bc97"),
                "index" : 10000
        }
}) :
WriteError({
        "index" : 0,
        "code" : 11000,
        "errmsg" : "E11000 duplicate key error collection: shop.tvs index: index_1 dup key: { index: 10000.0 }",
        "op" : {
                "_id" : ObjectId("62c92de50479502c3963bc97"),
                "index" : 10000
        }
})
WriteError@src/mongo/shell/bulk_api.js:465:48
mergeBatchResults@src/mongo/shell/bulk_api.js:871:49
executeBatch@src/mongo/shell/bulk_api.js:940:13
Bulk/this.execute@src/mongo/shell/bulk_api.js:1182:21
DBCollection.prototype.insertOne@src/mongo/shell/crud_api.js:264:9
@(shell):1:1
```

Let's talk about unique indexes. So right now, in our database, you can have in fact, here's a good example. Cause I think I did this earlier. Do these index find, index 10,000? No, okay. So I actually only have one item of index one 10,000, but this index should be unique, right?

No one should have the same index in this particular collection. So we want that to be unique. Or maybe you can imagine, like maybe you're doing a users collection and you want the username to be unique, right? So you put a unique constraint on that particular field. You can do that with unique indexes

So what we're gonna do is we're gonna say db.tvs.create index. And I'm going to index, that's funny index referring to this particular field in our collection, right? It's not nothing special, but the word index. And then all you have to do is you add another thing here and say, make sure it's unique.

Okay, so I went and created a new index, it's on index. So now, if I say db.let's say 1, that query will go much faster, cuz it's not gonna scan all of them. It actually goes directly to that particular thing, but the other interesting thing is I say, db.tvs.insert(1) I try to insert a index 10,000 like that.

It's gonna give me an error it's gonna say right there duplicate error key, right? So it's actually not going to let me insert duplicate keys here. That's what that unique constraint allows me to do. Whereas before it would have just happily accepted that right another useful thing about this is now index is quick is indexed right the the index is indexed.

That's just kind of a necessary byproduct of doing a unique constraint is it does have to index that.

Text-Search-Indexes

```
> db.tvs.createIndex({type:"text", company:"text", name:"text"});
{
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "createdCollectionAutomatically" : false,
        "ok" : 1

db.tvs.find({ $text: { $search: "phone tv MI 40 inch tv"}}).sort({score:{$meta:"textScore"}});
{ "_id" : ObjectId("62c5013247af947de18c985c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 3137 }
{ "_id" : ObjectId("62c5013247af947de18c9c4c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 4145 }
{ "_id" : ObjectId("62c5013247af947de18ca624"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 6665 }
{ "_id" : ObjectId("62c5013247af947de18cad08"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 8429 }
{ "_id" : ObjectId("62c5013247af947de18caa14"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 7673 }
{ "_id" : ObjectId("62c5013247af947de18cb1f4"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 9689 }
{ "_id" : ObjectId("62c5013247af947de18c9e44"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 4649 }
{ "_id" : ObjectId("62c5013247af947de18c9d48"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 4397 }
{ "_id" : ObjectId("62c5013247af947de18c907c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 1121 }
{ "_id" : ObjectId("62c5013247af947de18ca330"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 5909 }
{ "_id" : ObjectId("62c5013247af947de18c9958"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 3389 }
{ "_id" : ObjectId("62c5013247af947de18cac0c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 8177 }
{ "_id" : ObjectId("62c5013247af947de18c9f40"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 4901 }
{ "_id" : ObjectId("62c5013247af947de18c9274"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 1625 }
{ "_id" : ObjectId("62c5013247af947de18ca720"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 6917 }
{ "_id" : ObjectId("62c5013247af947de18c9178"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 1373 }
{ "_id" : ObjectId("62c5013247af947de18caf00"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 8933 }
{ "_id" : ObjectId("62c5013247af947de18caffc"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 9185 }
{ "_id" : ObjectId("62c5013247af947de18c9664"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7,
"company" : "MI", "index" : 2633 }
```

{ "_id" : ObjectId("62c5013247af947de18ca03c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 5153 }
Type "it" for more

we're gonna say db.tvs.createIndex again. But we're going to do this with type, text, company, text, name, text. And what's this is gonna do is its gonna create a new index containing all of these things, so now I can just say, hey, tvs, search for 40 inch tv

So by default MongoDB does not sort these for you. It does actually create what's called a text score. So you actually go have to go back and say, give me this and also sort by the one that matches most closely to my search terms. So you can do that by, same thing, but you have to give it a sort.

And that sort is by score, And score can be called whatever you want, it just has to be called something. But you have to give it this meta, $meta and you're gonna search by text score. I think that's right. Yep, okay. So now I searched again by that but if you look here at the top.

Now it's named 40 inch tv type phone tv company MI. And so it's actually more closely trying to figure out which of these matches my search terms the best. That makes sense?

```
db.tvs.find({ $text: { $search: "phone tv MI 40 inch tv"}},{score:{$meta:
"textScore"}}).sort({score:{$meta:"textScore"}});
```


{ "_id" : ObjectId("62c5013247af947de18c8d88"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 365, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18cb2f0"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 9941, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c9760"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 2885, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18cab10"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 7925, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c9a54"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 3641, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca138"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 5405, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c9568"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 2381, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c9370"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 1877, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c8f80"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 869, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18cb0f8"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 9437, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18cae04"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 8681, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca42c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 6161, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c9b50"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 3893, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c8c8c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 113, "owner" : "Samir Akhtar", "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca234"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 5657, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca81c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 7169, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca918"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 7421, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18ca03c"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 5153, "score" : 4.6 }
{ "_id" : ObjectId("62c5013247af947de18c8e84"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 617, "score" : 4.6 }

{ "_id" : ObjectId("62c5013247af947de18ca528"), "name" : "40 inch tv", "type" : "phone tv", "price" : 7, "company" : "MI", "index" : 6413, "score" : 4.6 }
Type "it" for more

So if you actually wanna see that, you can actually pull that score out so you can see the text score yourself.

You just have to give it a projection or not. You can just do it that way. So I'm going to say score and just put the same thing here. $meta: "textScore". It's through the projection. So you can see here, the first thing we're giving it is the search object.

The second thing, we have to project that score into the response, right, using the projections that we talked about earlier. And you can see here now it's giving these all a score of 4.6. I don't actually know what those scores mean, but you can see all these match it, but let's say like if we sort by text score.

Yeah, if it didn't match it as well, the text score would be different, and you can kinda sort by that. Yeah, that's how that works. Anyway, this is super useful if you have some sort of text field freeform text search. You can use these text scores to kind of pull things out.

Another nice thing about these texts score are these a full text search. It actually is language aware so it will drop things like the, and and, and it, right like those kind of stop words is what they're called. It's not just in English, it does it for English but it also has that ability for Dutch and German and Japanese and Chinese all that kind of stuff as well.

Or you can just have it do neutral, where it won't do any stop words whatsoever. All you have to do is you have to pass an extra flag to that Full Text Search to say, here's the language that I want you to do the full text search for.

You can even do it for multiple languages at a time. It actually works. If you want to do English and Spanish that you can do it that way as well. There's a lot more to full text search, there's a lot more superpowers that it has. This is just kind of the most basic way to do it.

So I left the link there in the course notes for you to go ahead and to go and explore various different ways to use full text search.

Is the search case sensitive?
No, pretty sure it's not, but let's give it a shot. Nope, it is not case sensitive.

I wanna say that there is a way to make it case sensitive, but I haven't tried. So the question is, is what happens if we insert another document? Do we have to reindex it? And the answer is no, it will continually index new things that you put into the database.

When you update it, it'll reindex itself. It automatically keeps its own up to indexes up to date.

Aggregation

MongoDB has a fun feature called aggregation. There's two ways of doing it, aggregation pipelines and map-reduce. Map-reduce is exactly what you'd expect if you're from a functional programming background: you provide MongoDB a map function that it will run every item in the array and then a reduce function to aggregate your collection into a smaller set of data.

MongoDB also released a newer feature of aggregation pipelines that tend to perform better and can also be easier to maintain. With these you provide a configuration object to the `aggregation` pipeline

```
db.tvs.aggregate([
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [0, 3, 9, 15],
      default: "very old",
      output: {
        count: { $sum: 1 },
      },
    },
  },
]);
```

- With the aggregation pipelines, you provide a step of things to do. In this case we only have one step, bucket tvs into 0-2 years old, 3-8 years old, 9-15 years, and "very old" (which is the default bucket.)
- With the output you're defining what you want to pass to the next step. In this case we just want to sum them up by adding 1 to the count each time we see a tv that matches a bucket.

This is all pets. We want just dogs. Let's add another stage.

```
db.tvs.aggregate([
  {
    $match: {
      type: "phone tv",
    },
  },
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [0, 3, 9, 15],
      default: "very old",
      output: {
        count: { $sum: 1 },
      },
    },
  },
```

```
]);
```

Using the $match stage of the aggregation, we can exclude every tvs that isn't a phone tv.

Last one, what if we wanted to sort the results by which group had the most tvs?

```
db.tvs.aggregate([
  {
    $match: {
      type: "phone tv",
    },
  },
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [0, 3, 9, 15],
      default: "very old",
      output: {
        count: { $sum: 1 },
      },
    },
  },
  {
    $sort: {
      count: -1,
    },
  },
]);
```

As you can see, you just add more stages to the aggregation until you gather the insights you're looking for.

This is definitely one of the most fun parts about MongoDB. We used to use MongoDB's aggregation features to catch fraudsters in our classifieds app!

```
db.tvs.aggregate([  { $bucket: { groupBy: "$price", boundaries: [0, 3, 9, 15], default: "16+", output :{ count: {$sum: 1} } } } ])
{ "_id" : 0, "count" : 794 }
{ "_id" : 3, "count" : 3215 }
{ "_id" : 9, "count" : 3213 }
{ "_id" : "16+", "count" : 2421 }

db.tvs.aggregate([ {$match:{type:"phone tv",},},  { $bucket: { groupBy: "$price", boundaries: [0, 3, 9, 15], default: "16+", output :{ count: {$sum: 1} } } } ])
{ "_id" : 3, "count" : 833 }
{ "_id" : 9, "count" : 833 }
{ "_id" : "16+", "count" : 833 }
```

We're gonna say DB.tvs.aggregate. And then it does this in stages, right? So you can give it multiple different stages of like, do this first, then do this, then do this.

So we're going to give it one stage first of bucketing. So the bucket allows you to break your we're going to break our tvs collection down into buckets of by price. So with the bucket here, we're going to say. GroupBy, Dollar sign price so the dollar sign is just saying that's the name of the field that I want you to look for.

So dollar sign price is going to be grouped by their price. Okay then the boundaries. This is going to be defining the kind of boundaries of where we want to put our phone tv. So the first one is going to be down to zero, right? So, anything between zero and two, I don't know if I'm correct, but I'm asserting anything between zero and two.

```
db.tvs.aggregate([ {$match:{type:"android tv",},},  { $bucket: { groupBy: "$price", boundaries: [0, 3, 9, 15],
default: "16+", output :{ count: {$sum: 1}, }, },}, { $sort: { count: -1, },}, ]);
{ "_id" : 3, "count" : 834 }
{ "_id" : 9, "count" : 833 }
{ "_id" : "16+", "count" : 555 }
{ "_id" : 0, "count" : 278 }
```

Now we're just saying sort by count descending, which basically says we're going to have the one with the most first second most second, so on and so forth. So you can see here we have the most high rate tvs, second high rate tvs, third high rate tvs and the least amount of tvs.

This is just one very, very simple use case of the aggregation pipeline. This is a infinitely flexible, you can do lots of really amazing things with the aggregation pipeline. I just wanted to kinda like get your head around it a little bit. So you can think like, hey, I need to derive this insight out of my data.

There are a lot of stages. So I showed you a bucket I showed you he can do count. Match. I showed you match. I showed you sort there's projectors redact, replace with sample.

Also, like there's a lot of different stuff you can do with the aggregation. Pipeline. How do you query buckets once you've gone through the aggregation pipeline. What you get back is an array of objects and you can treat those as you would basically documents. Otherwise in your yourself.

But it's not terribly difficult you just end up acquiring them like odd like normal objects. There's nothing special you need to treat them. In fact, you can see here I did literally query them right here. With the word I just said, hey, sort this and count by negative one.

You could bucket these again by saying here's all the ones that have more than 800. And here's all the ones that have less than 800. It's all just objects to MongoDB. That's a good question. Are the buckets saved or indexed or anything like that? Not by default, by default, it's just gonna print out what you ask it to print out.

It does have the ability to actually write to another collection. So let's say you're doing like daily stats on how many people visited your website, right and you were keeping all of your sessions in a database. One that's not really a good use case for MongoDB. But you could potentially hit the software exists to do that.

And then at the end of the day, you wanted to aggregate that into like, how many Firefox users visited and how many. Chrome users and edge users and all that kind of stuff. You could do that with the aggregation pipeline. And then you could have that right to another, like daily starts collection and then you could read from that collection.

Every day, right? So by default, it doesn't save anywhere. But you could write that to a another collection. Obviously, that's not gonna be indexed by default either, but there's no reason you couldn't index that as well.

I don't actually really know precisely the algorithm that goes into the one thing that I do know is it actually does create does contain a time stamp in it. So it is based on the time when was written, you can actually pull the time stamp out of it.

You'd have to go look it up if you just search for like MongoID. Time stamp, it'll contain the creation time in it. So you can actually, you don't actually if you're looking at creation times, just know that the object idea automatically contains that whether or not you created or not.

That's the only interesting thing I can remember about it. And the other thing is it's always guaranteed unique. But the randomization is through the time. Even if two objects are inserted relatively close to each other in time. They will never have the same ID this this object ID here 100% of the time will always be totally unique to that object.

Write a Node.js app with MongoDB

Let's quickly write up a quick Node.js project to help you transfer your skills from the command line to the coding world.

Make a new directory. In that directory run:

```
npm init -y
npm i express mongodb@3.6.2 express@4.17.1
mkdir static
touch static/index.html server.js
code . # or open this folder in VS Code or whatever editor you want
```

Let's make a dumb front end that just makes search queries against the backend. In static/index.html put:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>MongoDB Sample</title>
  </head>
  <body>
    <input type="text" placeholder="Search" id="search" />
    <button id="btn">Search</button>
    <pre>
        <code id="code"></code>
    </pre>

    <script>
      const btn = document.getElementById("btn");
      const code = document.getElementById("code");
      const search = document.getElementById("search");

      btn.addEventListener("click", async () => {
        code.innerText = "loading";

        const res = await fetch(
          "/get?search=" + encodeURIComponent(search.value)
        );
```

```
        const json = await res.json();

        code.innerText = "\n" + JSON.stringify(json, null, 4);
      });
    </script>
  </body>
</html>
```

In server.js, put this:

```
const express = require("express");
const { MongoClient } = require("mongodb");

const connectionString =
  process.env.MONGODB_CONNECTION_STRING ||
"mongodb://localhost:27017";

async function init() {
  const client = new MongoClient(connectionString, {
    useUnifiedTopology: true,
  });
  await client.connect();

  const app = express();

  app.get("/get", async (req, res) => {
    const db = await client.db("shop");
    const collection = db.collection("tvs");

    const pets = await collection
      .find(
        {
          $text: { $search: req.query.search },
        },
        { _id: 0 }
      )
```

```
      .sort({ score: { $meta: "textScore" } })
      .limit(10)
      .toArray();

    res.json({ status: "ok", tvs}).end();
  });

  const PORT = process.env.PORT || 3000;
  app.use(express.static("./static"));
  app.listen(PORT);

  console.log(`running on http://localhost:${PORT}`);
}
init();
```