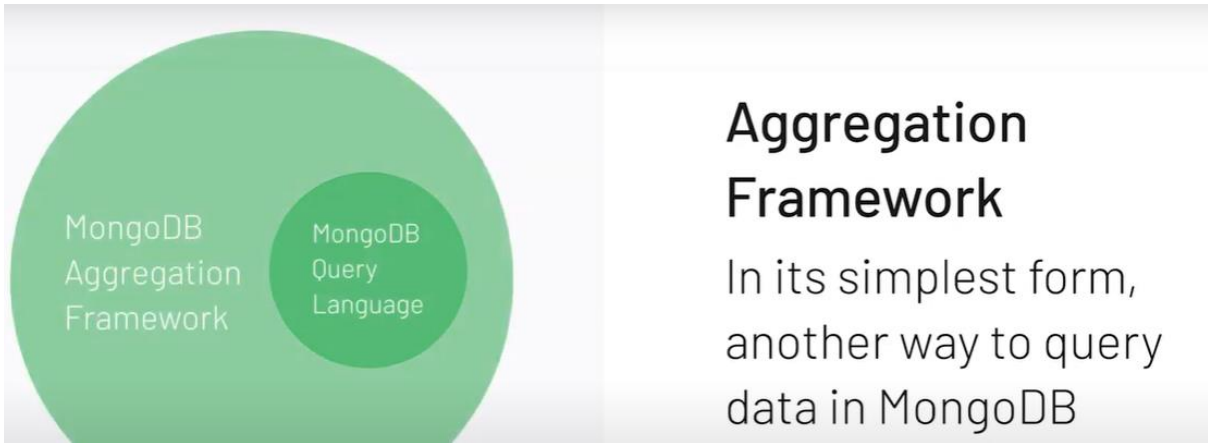
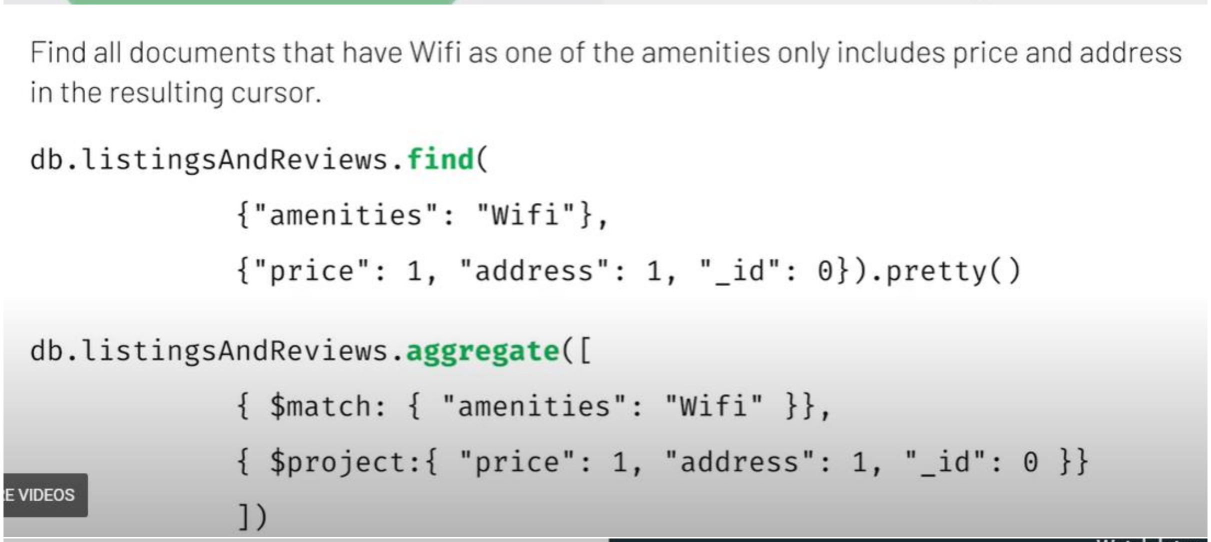
1. Aggregation Framework:



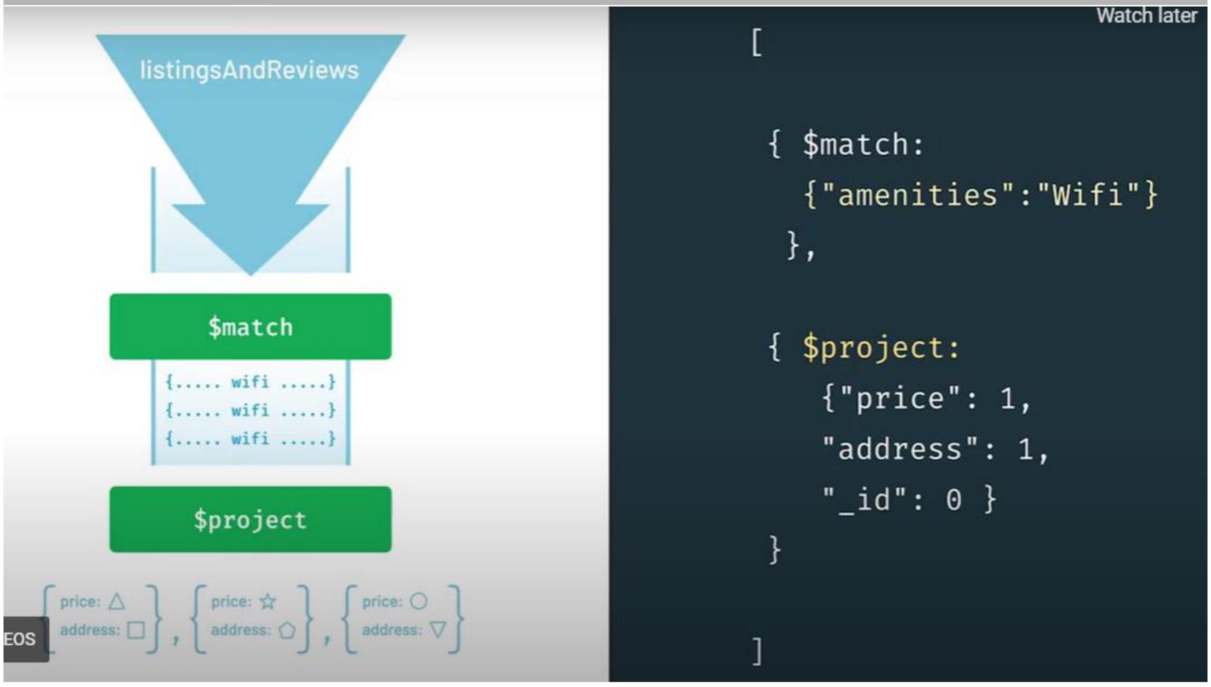
Aggregation Framework

In its simplest form, another way to query data in MongoDB

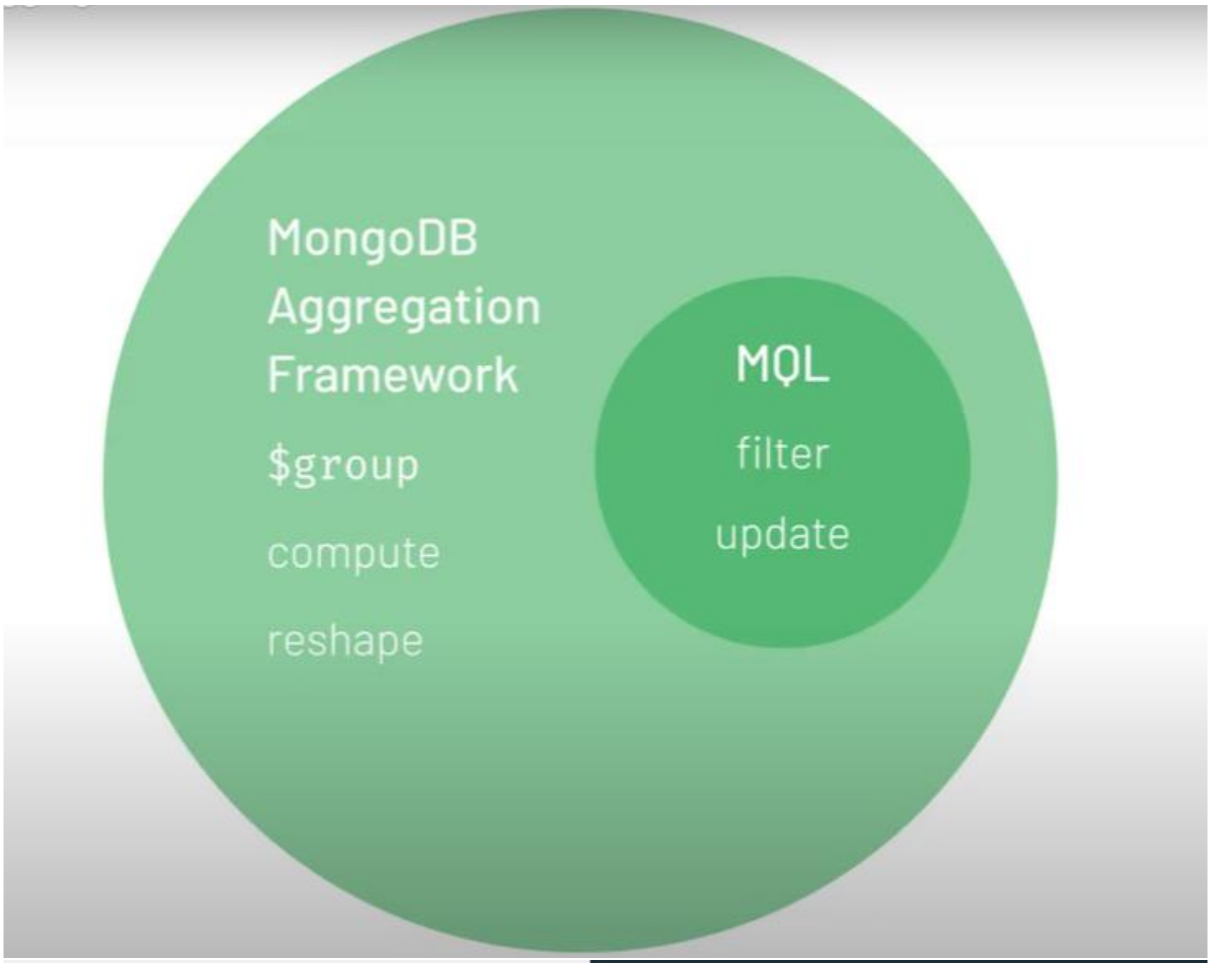


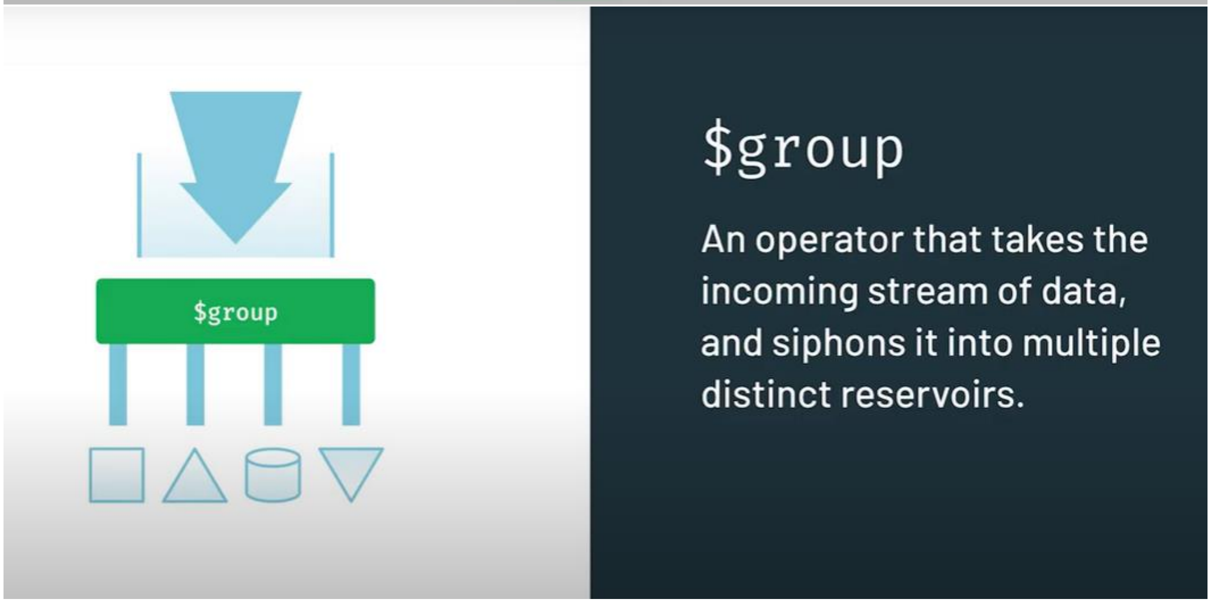
Find all documents that have Wifi as one of the amenities only includes price and address in the resulting cursor.

```
db.listingsAndReviews.find(
            {"amenities": "Wifi"},
            {"price": 1, "address": 1, "_id": 0}).pretty()

db.listingsAndReviews.aggregate([
            { $match: { "amenities": "Wifi" }},
            { $project:{ "price": 1, "address": 1, "_id": 0 }}
            ])
```



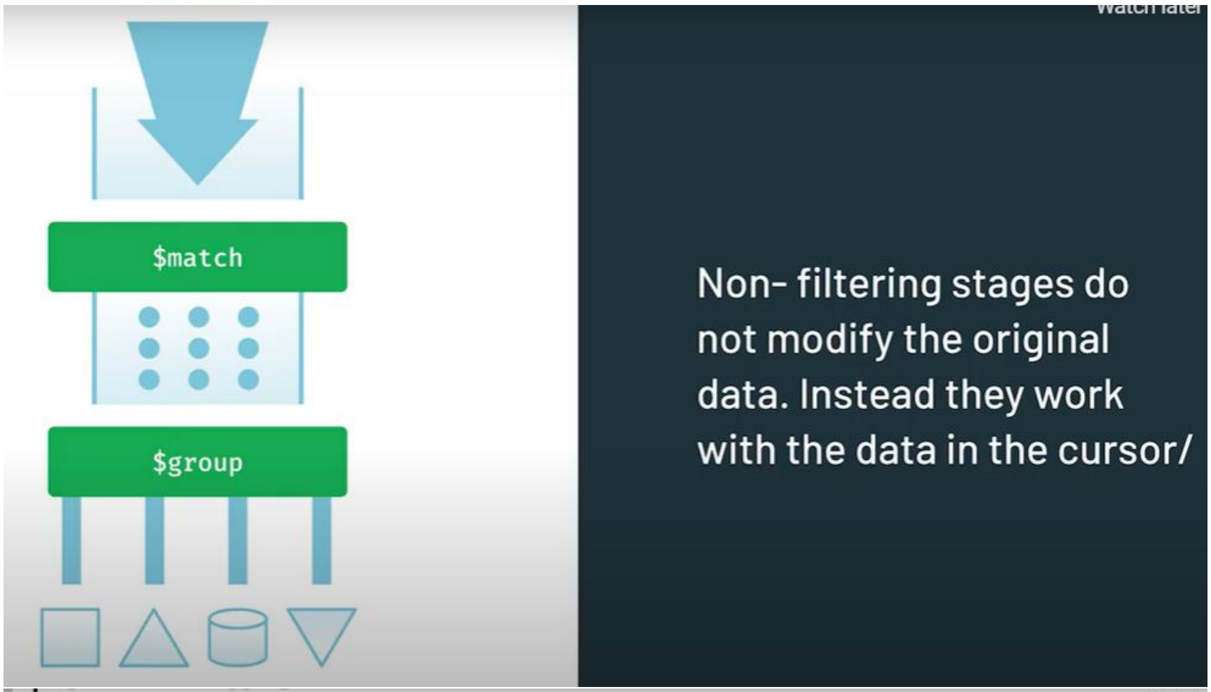
```
[

 { $match:
   {"amenities":"Wifi"}
  },

 { $project:
    {"price": 1,
    "address": 1,
    "_id": 0 }
 }

]
```



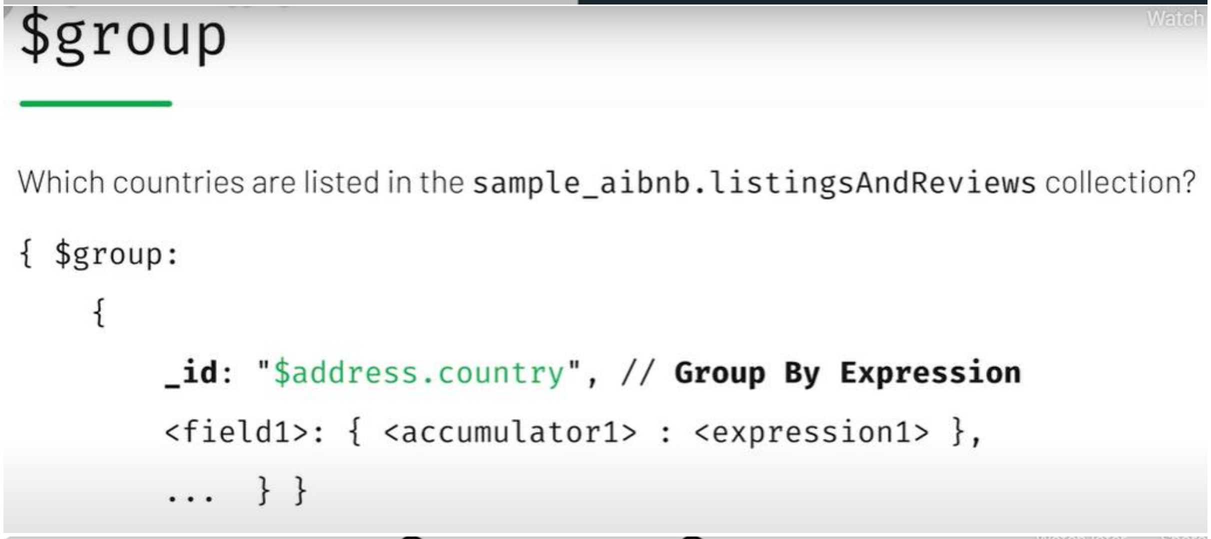


$group

An operator that takes the incoming stream of data, and siphons it into multiple distinct reservoirs.



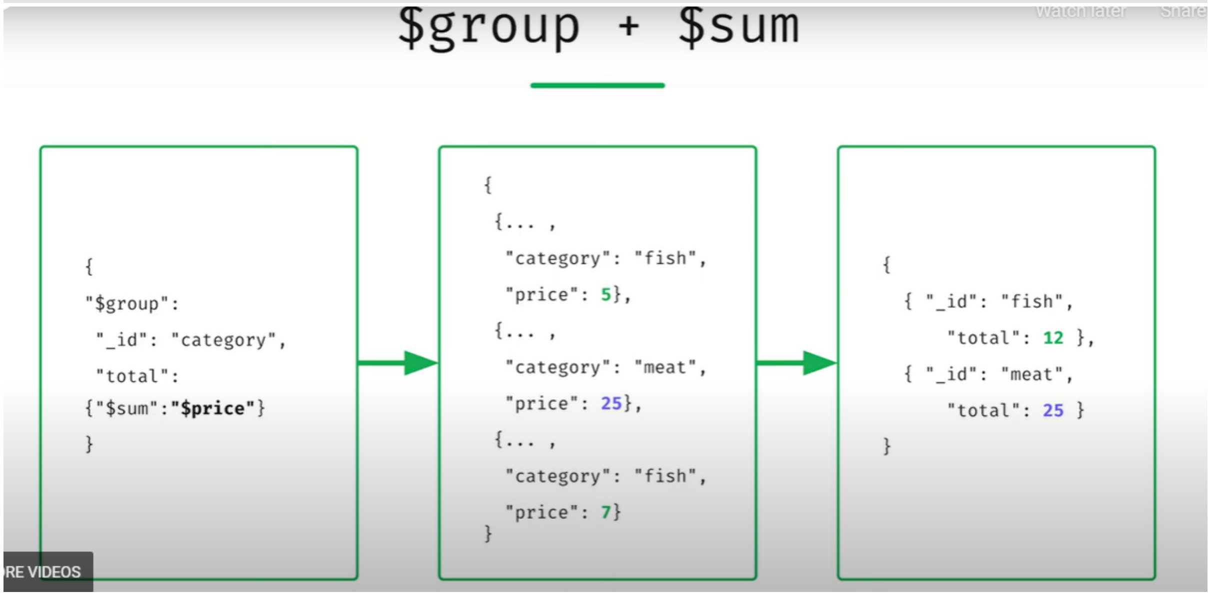
Non- filtering stages do not modify the original data. Instead they work with the data in the cursor/



$group

Which countries are listed in the `sample_aibnb.listingsAndReviews` collection?

```
{ $group:
    {
        _id: "$address.country", // Group By Expression
        <field1>: { <accumulator1> : <expression1> },
        ...  } }
```



$group + $sum

```
{
"$group":
 "_id": "category",
 "total":
{"$sum":"$price"}
}
```

```
{
 {... ,
  "category": "fish",
  "price": 5},
 {... ,
  "category": "meat",
  "price": 25},
 {... ,
  "category": "fish",
  "price": 7}
}
```

```
{
  { "_id": "fish",
       "total": 12 },
  { "_id": "meat",
       "total": 25 }
}
```

Switch to this database:

```
use sample_airbnb
```

Find all documents that have `Wifi` *as one of the amenities. Only include* `price` *and* `address` *in the resulting cursor.*

```
db.listingsAndReviews.find({ "amenities": "Wifi" },

                           { "price": 1, "address": 1, "_id":
0 }).pretty()
```

Using the aggregation framework find all documents that have `Wifi` *as one of the* `amenities``*. Only include* ``price` *and* `address` *in the resulting cursor.*

```
db.listingsAndReviews.aggregate([

                                  { "$match": { "amenities":
"Wifi" } },

                                  { "$project": { "price": 1,

                                                  "address":
1,

                                                  "_id": 0
}}]).pretty()
```

Find one document in the collection and only include the `address` *field in the resulting cursor.*

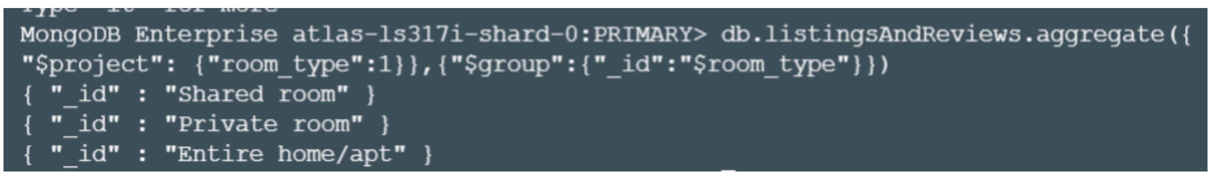
```
db.listingsAndReviews.findOne({ },{ "address": 1, "_id": 0 })
```

Project only the address field value for each document, then group all documents into one document per `address.country` *value.*

```
db.listingsAndReviews.aggregate([ { "$project": { "address":
1, "_id": 0 }},

                                  { "$group": { "_id":
"$address.country" }}])
```

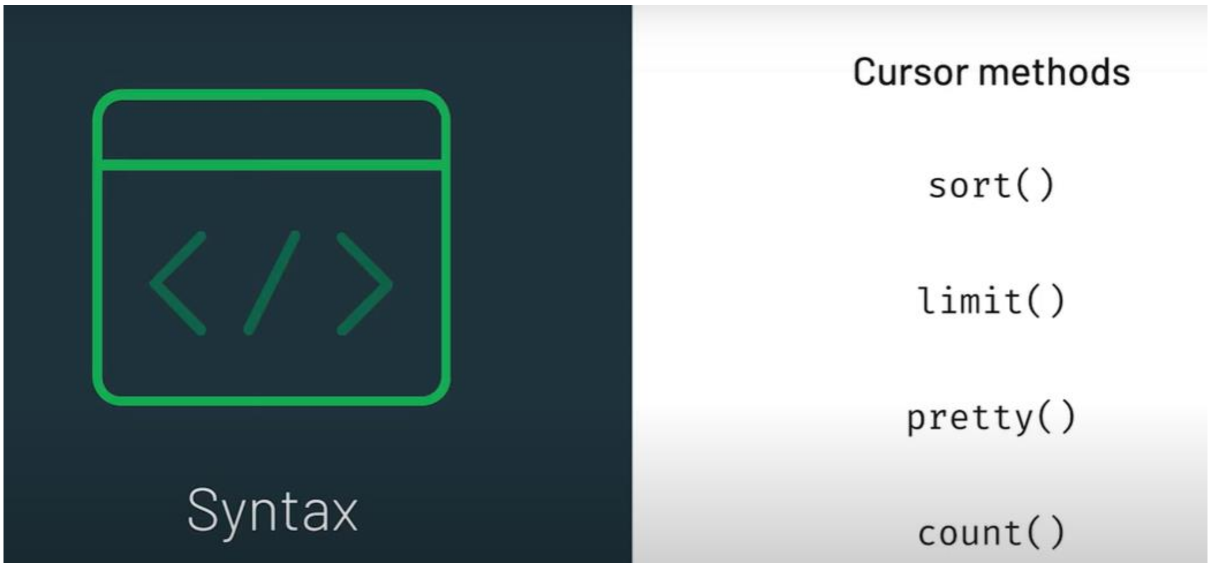
Project only the `address` *field value for each document, then group all documents into one document per* `address.country` *value, and count one for each document in each group.*

```
db.listingsAndReviews.aggregate([

                   { "$project": { "address": 1, "_id": 0 }},

                   { "$group": { "_id": "$address.country",

                                                 "count": {
"$sum": 1 } } }

                                ])
```



```
MongoDB Enterprise atlas-ls317i-shard-0:PRIMARY> db.listingsAndReviews.aggregate({
"$project": {"room_type":1}},{"$group":{"_id":"$room_type"}})
{ "_id" : "Shared room" }
{ "_id" : "Private room" }
{ "_id" : "Entire home/apt" }
```

```
use sample_training

db.zips.find().sort({ "pop": 1 }).limit(1)

db.zips.find({ "pop": 0 }).count()

db.zips.find().sort({ "pop": -1 }).limit(1)

db.zips.find().sort({ "pop": -1 }).limit(10)

db.zips.find().sort({ "pop": 1, "city": -1 })
```



Syntax

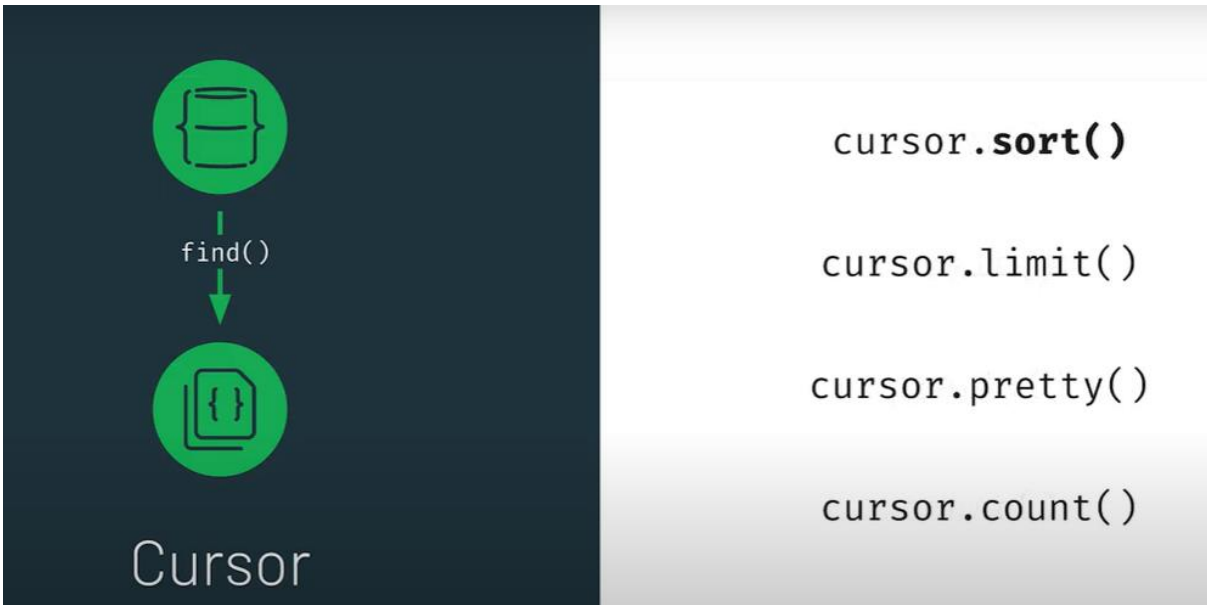
Cursor methods

sort()

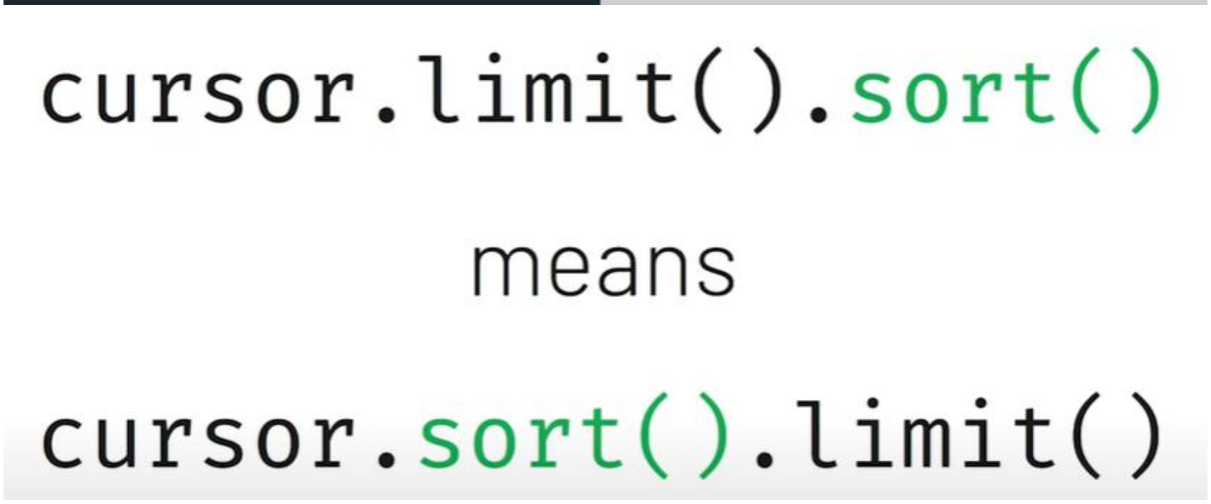
limit()

pretty()

count()



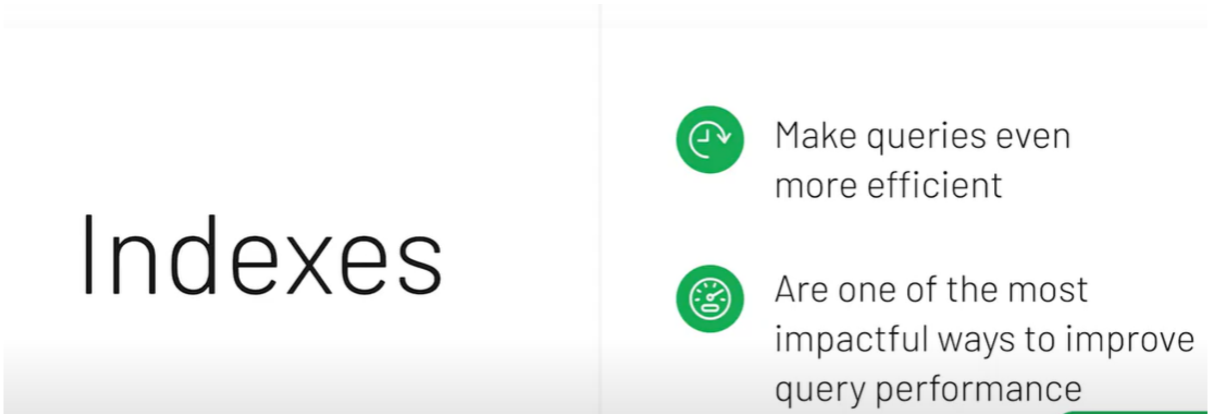
```
cursor.sort()
cursor.limit()
cursor.pretty()
cursor.count()
```



`cursor.limit().sort()`

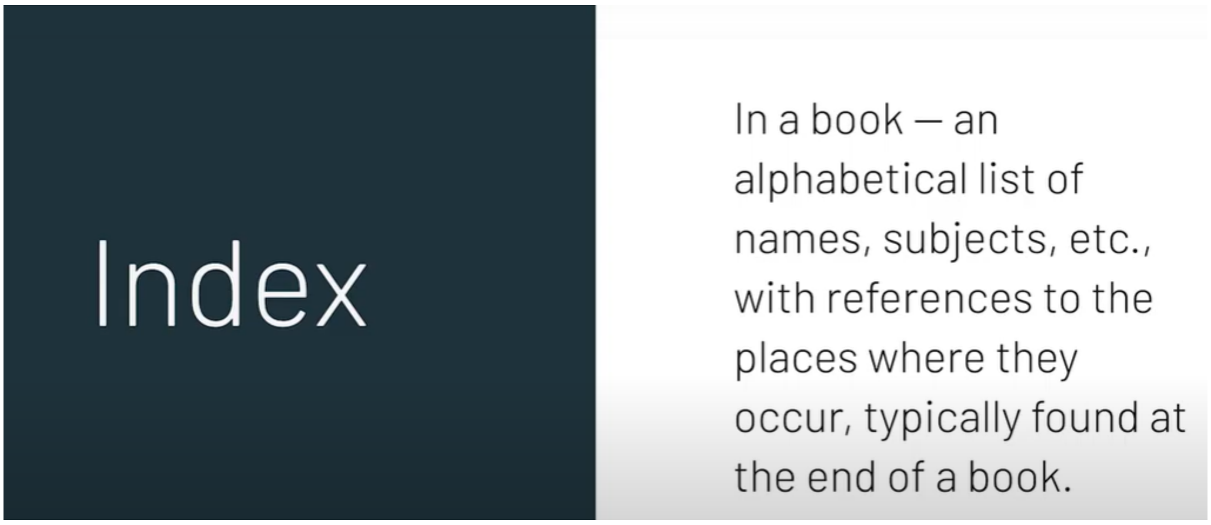
means

`cursor.sort().limit()`



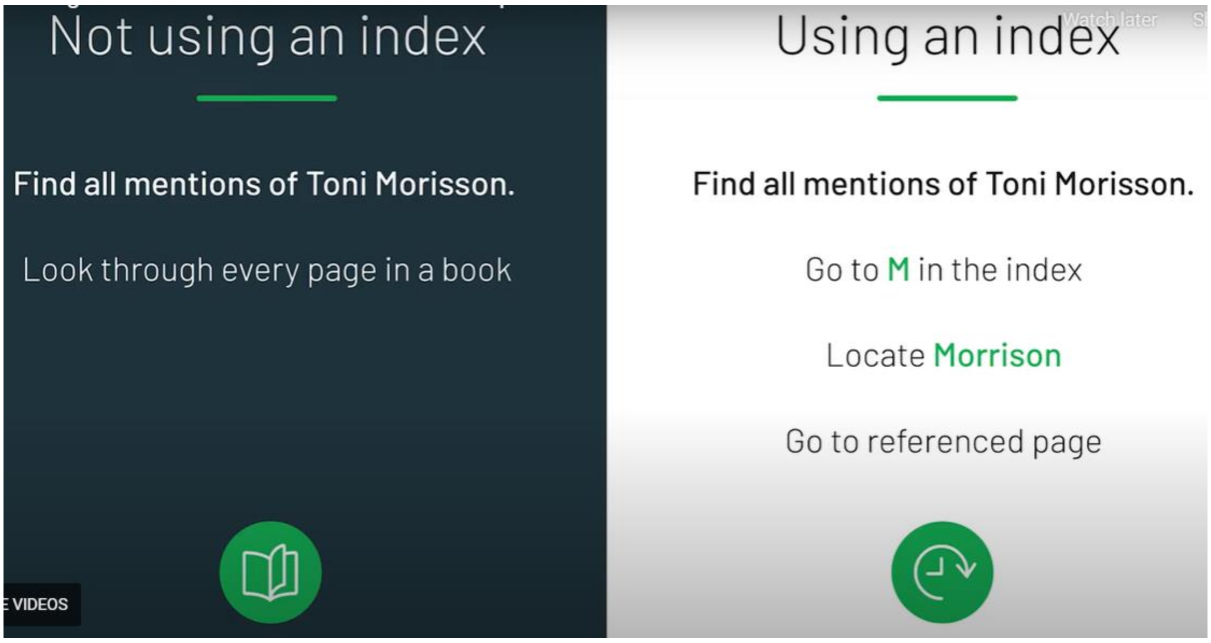
Indexes

- Make queries even more efficient
- Are one of the most impactful ways to improve query performance



Index

In a book — an alphabetical list of names, subjects, etc., with references to the places where they occur, typically found at the end of a book.



Not using an index

Find all mentions of Toni Morisson.

Look through every page in a book

Using an index

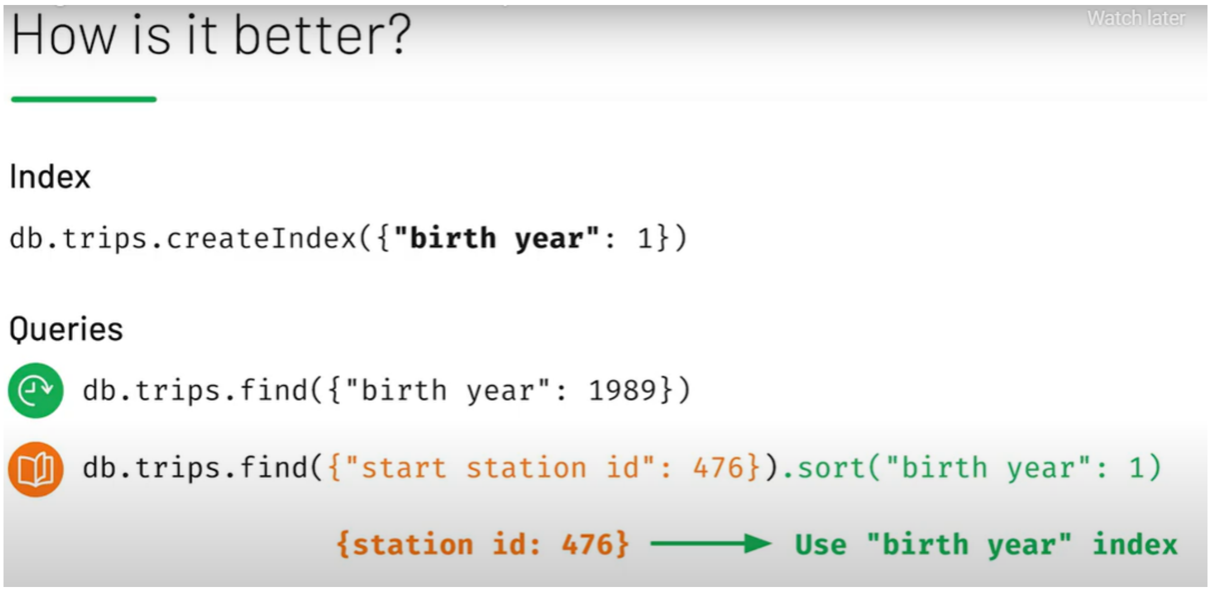
Find all mentions of Toni Morisson.

Go to M in the index

Locate Morrison

Go to referenced page

In a database – special data structure that stores a small portion of the collection's data set in an easy to traverse form.(Index)



How is it better?

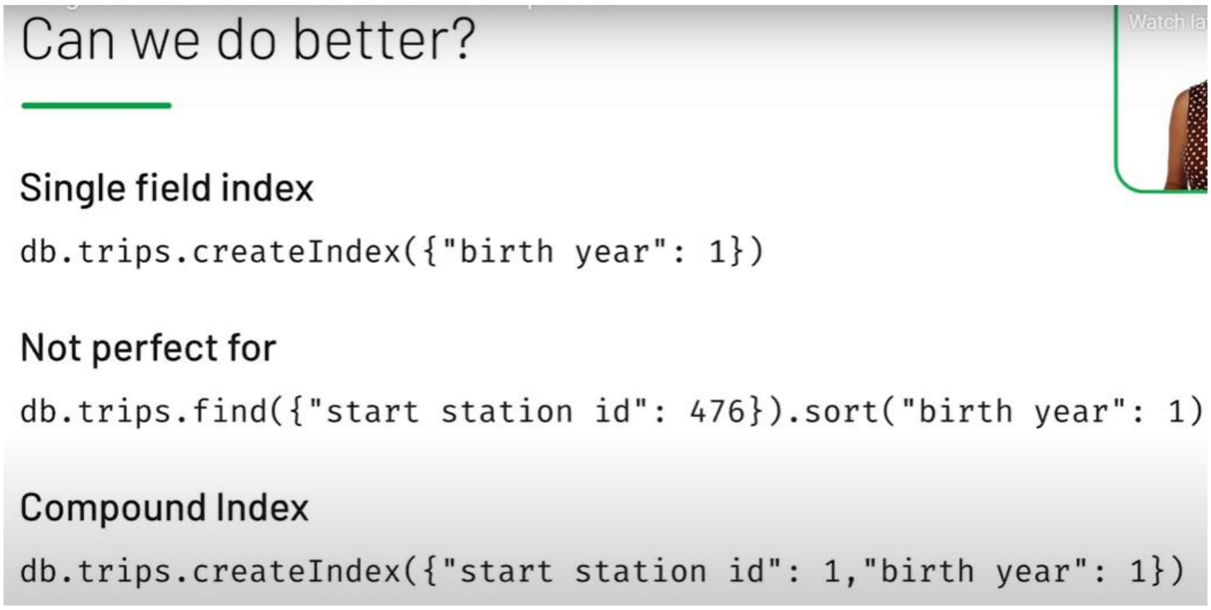
Index

```
db.trips.createIndex({"birth year": 1})
```

Queries

```
db.trips.find({"birth year": 1989})
db.trips.find({"start station id": 476}).sort("birth year": 1)
```

{station id: 476} ⟶ **Use "birth year" index**



Can we do better?

Single field index

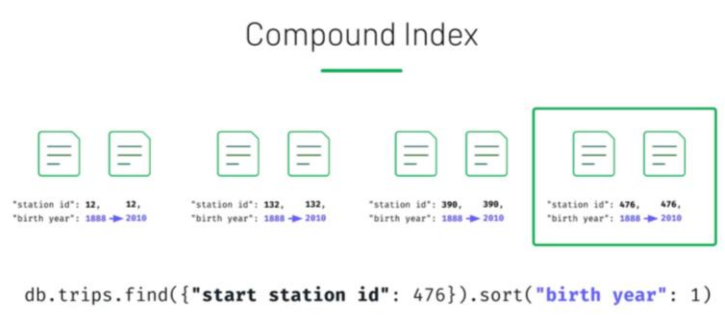
```
db.trips.createIndex({"birth year": 1})
```

Not perfect for

```
db.trips.find({"start station id": 476}).sort("birth year": 1)
```

Compound Index

```
db.trips.createIndex({"start station id": 1,"birth year": 1})
```



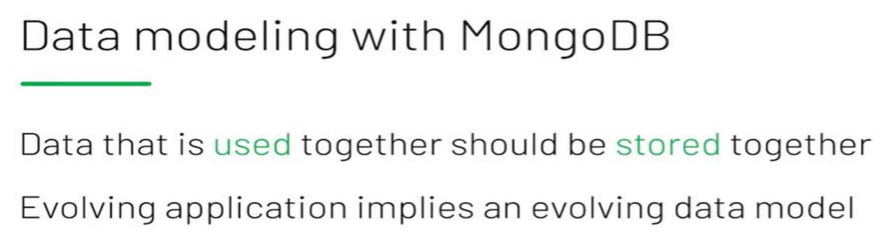
```
db.trips.find({"start station id": 476}).sort("birth year": 1)
```

```
use sample_training

db.trips.find({ "birth year": 1989 })
db.trips.find({ "start station id": 476 }).sort( { "birth
year": 1 } )

db.trips.createIndex({ "birth year": 1 })

db.trips.createIndex({ "start station id": 476, "birth
year": 1 })
```

Making decisions about the shape and structure of data is called data modelling.

Data modeling - a way to organize fields in a document to support your application performance and querying capabilities.

Rule: data is stored in the way that it is used

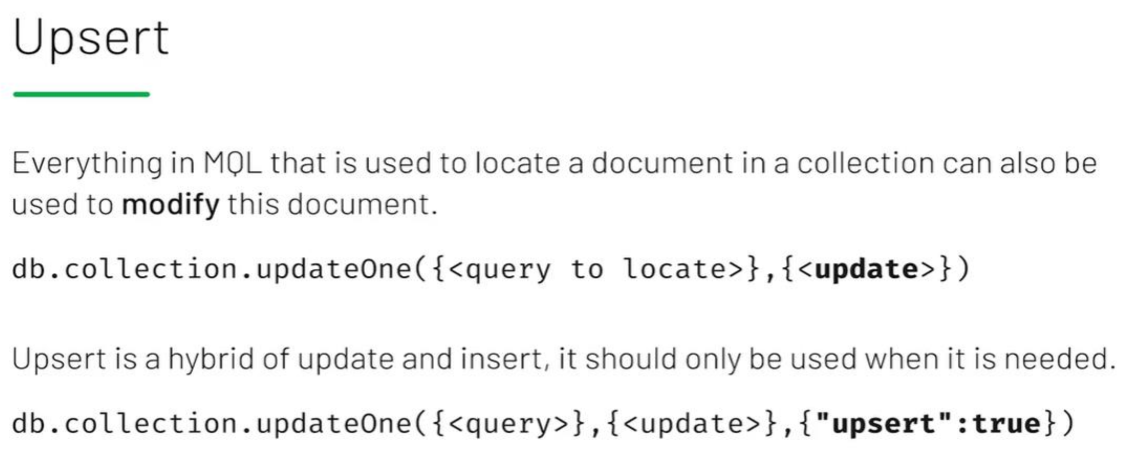


Data modeling with MongoDB

Data that is used together should be stored together

Evolving application implies an evolving data model

Upsert:



Upsert

Everything in MQL that is used to locate a document in a collection can also be used to **modify** this document.

```
db.collection.updateOne({<query to locate>},{<update>})
```

Upsert is a hybrid of update and insert, it should only be used when it is needed.

```
db.collection.updateOne({<query>},{<update>},{"upsert":true})
```

If `upsert` is `true`

YES

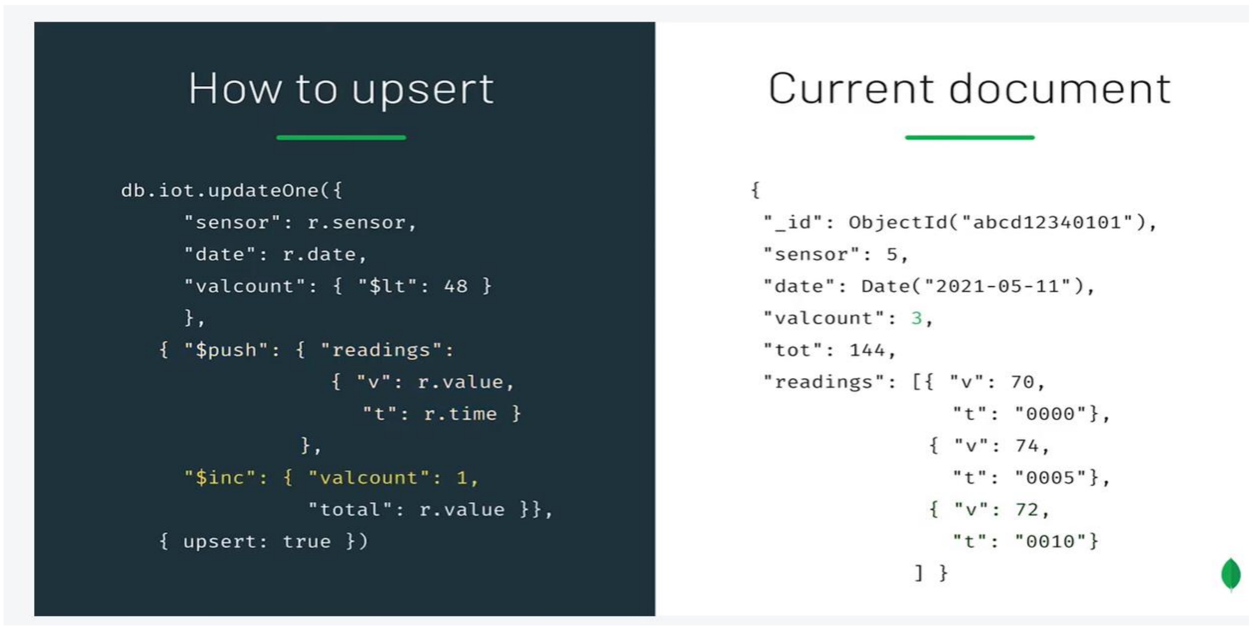
Is there a match?

Update the matched document

NO

Is there a match?

Insert a new document



How to upsert

```
db.iot.updateOne({
     "sensor": r.sensor,
     "date": r.date,
     "valcount": { "$lt": 48 }
     },
   { "$push": { "readings":
                { "v": r.value,
                  "t": r.time }
              },
     "$inc": { "valcount": 1,
               "total": r.value }},
   { upsert: true })
```

Current document

```
{
 "_id": ObjectId("abcd12340101"),
 "sensor": 5,
 "date": Date("2021-05-11"),
 "valcount": 3,
 "tot": 144,
 "readings": [{ "v": 70,
                "t": "0000"},
              { "v": 74,
                "t": "0005"},
              { "v": 72,
                "t": "0010"}
             ] }
```

`upsert:true`

Be mindful

Is `{<update>}` enough to create a new document?

Will the document have the same or similar form to other documents in the collection?

Summary

`upsert : true`

Conditional updates

`upsert : false`

Update an existing document

Insert a brand new document

```
db.iot.updateOne({ "sensor": r.sensor, "date": r.date,

                   "valcount": { "$lt": 48 } },

                         { "$push": { "readings": { "v":
r.value, "t": r.time } },

                        "$inc": { "valcount": 1, "total":
r.value } },

                 { "upsert": true })
```