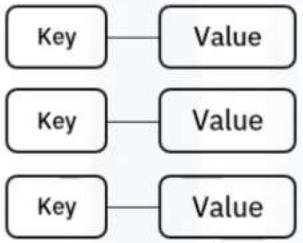
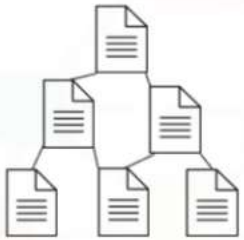
Subtitle

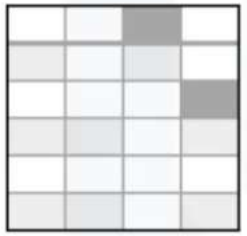
NoSQL Database Categories



Key-Value



Document



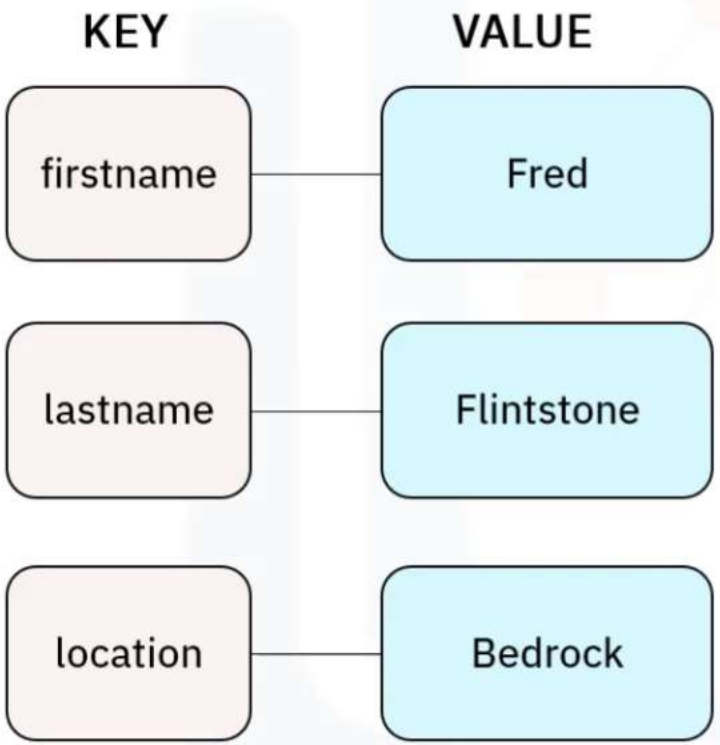
Column

Graph

Each category has:

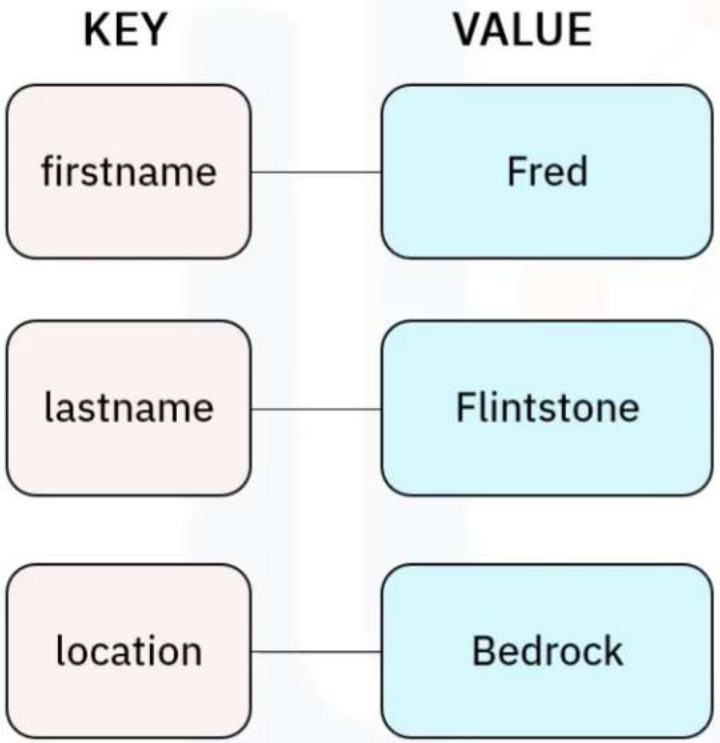
- Unique characteristics
- Architecture
- Use cases

Key-Value NoSQL Database Architecture



- Least complex
- Represented as hashmap
- Ideal for basic CRUD operations
- Scale well
- Shard easily

Key-Value NoSQL Database Architecture



- Not intended for complex queries
- Atomic for single key operations only
- Value blobs are opaque to database
  - Less flexible data indexing and querying

Key-Value NoSQL Database Use Cases

Suitable Use Cases

- For quick basic CRUD operations on non-interconnected data
  - E.g. Storing and retrieving session information for web applications
- Storing in-app user profiles and preferences
- Shopping cart data for online stores

Key-Value NoSQL Database Use Cases

Unsuitable Use Cases

- For data that is interconnected with many-to-many relationships
  - Social networks
  - Recommendation engines
- When high-level consistency is required for multi-operation transactions with multiple keys
  - Need a database that provides ACID transactions
- When apps runs queries based on value vs key
  - Consider using ‘Document’ category of NoSQL database

Key-Value NoSQL Database Examples

Summary

In this video, you learned that:

- The four main categories of NoSQL database are Key-Value, Document, Column, and Graph
- The Key-Value database architecture is the least complex; data is stored with a key and corresponding value blob and is represented by a hashmap
- The primary use cases for Key-Value databases are for quick CRUD operations; for example, storing and retrieving session information, storing in-app user profiles, and storing shopping cart data

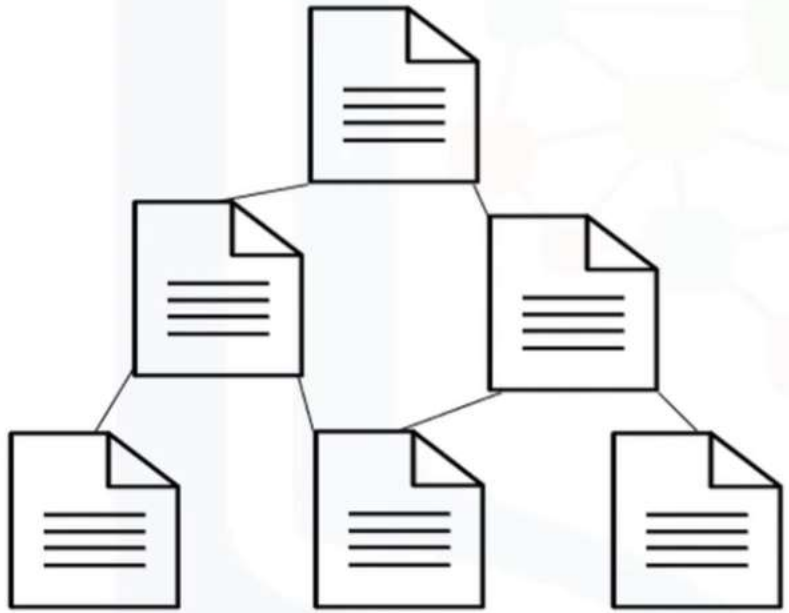
Document-Based NoSQL Databases

Objectives

After watching this video, you will be able to:

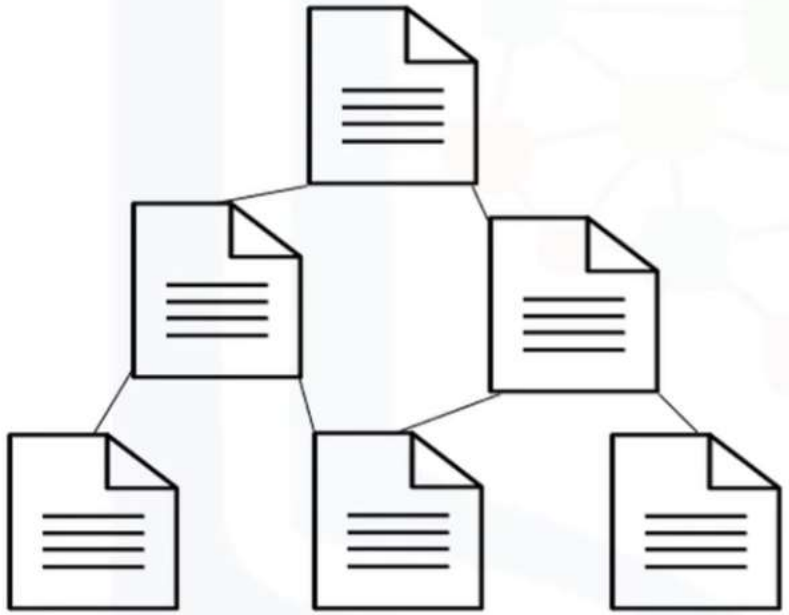
- Describe the architecture of the Document-Based category of NoSQL database
- Explain the primary use cases for the Document-Based NoSQL database category

Document-Based NoSQL Database Architecture



- Values are visible and can be queried
- Each piece of data is considered a document
  - Typically JSON or XML
- Each document offers a flexible schema
  - No two documents need to contain the same information

Document-Based NoSQL Database Architecture



- Content of document databases can be indexed and queried
  - Key and value range lookups and search
  - Analytical queries with MapReduce
- Horizontally scalable
- Allow sharding across multiple nodes
- Typically only guarantee atomic operations on single documents

Document-Based NoSQL Database Use Cases

Suitable Use Cases

- Event logging for apps and processes – each event instance is represented by a new document
- Online blogs – each user, post, comment, like, or action is represented by a document
- Operational datasets and metadata for web and mobile apps – designed with Internet in mind (JSON, RESTful APIs, unstructured data)

Document-Based NoSQL Database Use Cases

Unsuitable Use Cases

- When you require ACID transactions
  - Document databases can't handle transactions that operate over multiple documents
  - Relational database would be a better choice
- If your data is in an aggregate-oriented design
  - If data naturally falls into a normalized tabular model
  - Relational database would be a better choice

Document-Based NoSQL Database Examples

IBM Cloudant®

mongoDB

CouchDB

terrastore

OrientDB®

Couchbase

RAVENDB

Summary

In this video, you learned that:

- Document-based NoSQL databases use documents to make values visible and able to be queried
- Each piece of data is considered a document (JSON/XML)
- Each document offers a flexible schema
- The primary use cases for document-based NoSQL databases are event logging for apps/processes, online blogging, operational datasets or metadata for web and mobile apps

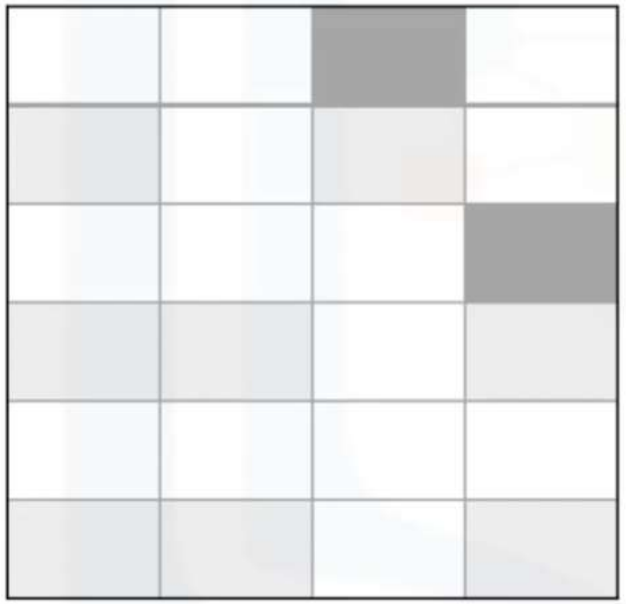
Column-Based NoSQL Databases

Objectives

After watching this video, you will be able to:

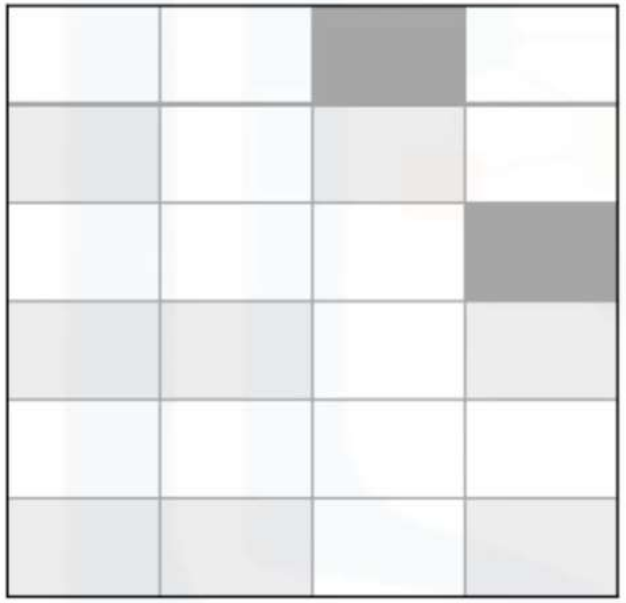
- Describe the architecture of the Column-Based category of NoSQL database
- Explain the primary use cases for the Column-Based NoSQL database category

Column-Based NoSQL Database Architecture



- Spawned from Google's 'Bigtable'
- a.k.a. Bigtable clones or Columnar or Wide-Column databases
- Store data in columns or groups of columns

Column-Based NoSQL Database Architecture



- Column ‘families’ are several rows, with unique keys, belonging to one or more columns
  - Grouped in families as often accessed together
- Rows in a column family are not required to share the same columns
  - Can share all, a subset, or none
  - Columns can be added to any number of rows, or not

Column-Based NoSQL Database Use Cases

Suitable Use Cases

- Great for large amounts of sparse data
- Column databases can handle being deployed across clusters of nodes
- Column databases can be used for event logging and blogs
- Counters are a unique use case for column databases
- Columns can have a TTL parameter, making them useful for data with an expiration value

Column-Based NoSQL Database Use Cases

Unsuitable Use Cases

- For traditional ACID transactions
  - Reads and writes are only atomic at the row level
- In early development, query patterns may change and require numerous changes to column-based databases
  - Can be costly and can slow down the production timeline

Column-Based NoSQL Database Examples

Summary

In this video, you learned that:

- Column-based databases were spawned from the architecture of Google's Bigtable storage system
- Column-based databases store data in columns or groups of columns
- Column 'families' are several rows, with unique keys, belonging to one or more columns
- The primary use cases for Column-based databases are event logging, blogs, counters, and data with expiration values

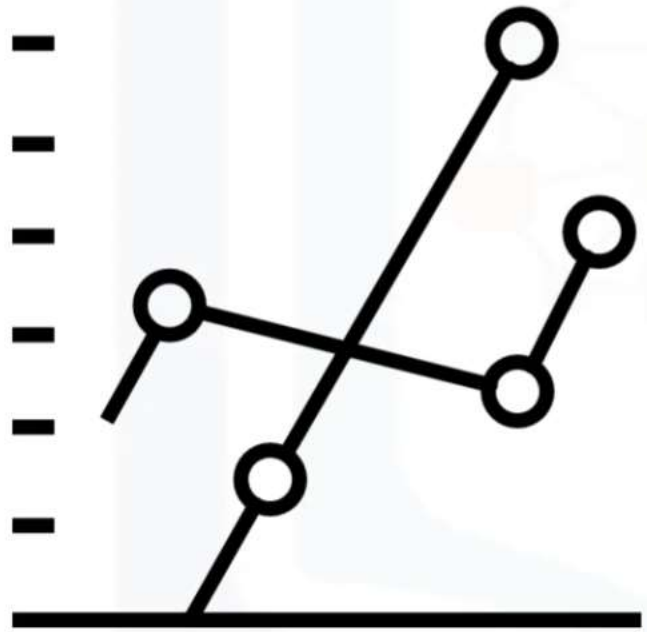
Graph NoSQL Databases

Objectives

After watching this video, you will be able to:

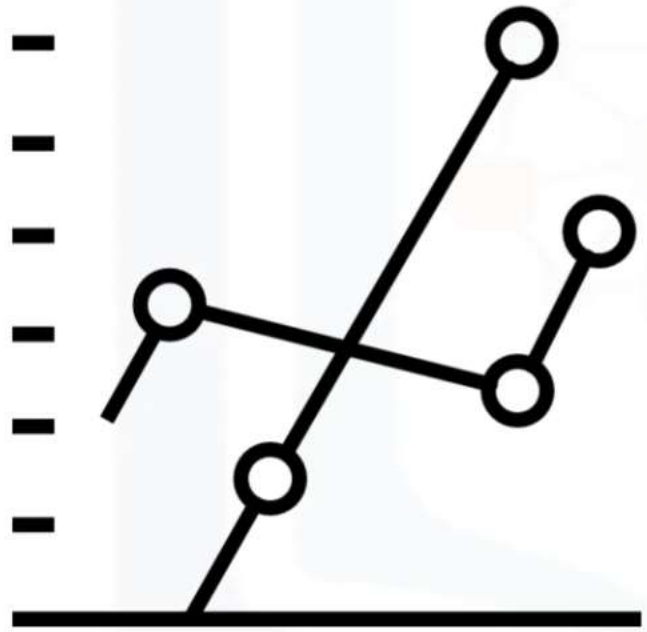
- Describe the architecture of the Graph category of NoSQL database
- Explain the primary use cases for the Graph NoSQL database category

Graph NoSQL Database Architecture



- Graph databases store information in entities (or nodes), and relationships (or edges
- Graph databases are impressive when your data set resembles a graph-like data structure

Graph NoSQL Database Architecture



- Graph databases do not shard well
  - Traversing a graph with nodes split across multiple servers can become difficult and hurt performance
- Graph databases are ACID transaction compliant
  - Unlike other NoSQL databases discussed

Graph NoSQL Database Use Cases

Suitable Use Cases

- For highly connected and related data
- Social networking
- Routing, spatial, and map apps
- Recommendation engines

Graph NoSQL Database Use Cases

Unsuitable Use Cases

- When looking for advantages offered by other NoSQL database categories
- When an application needs to scale horizontally
  - You will quickly reach the limitations associated with these types of data stores
- When trying to update all or a subset of nodes with a given parameter
  - These types of operations can prove to be difficult and non-trivial

Graph NoSQL Database Examples

Summary

In this video, you learned that:

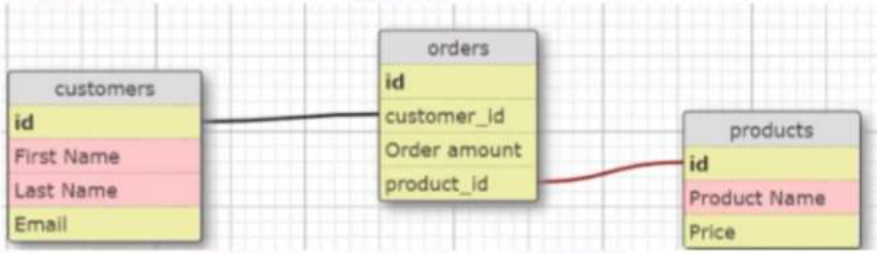
- Graph databases store information in entities and relationships
- Graph databases are impressive when your data set resembles a graph-like data structure
- Graph databases don’t shard well but are ACID transaction compliant
- The primary use cases for Graph databases are for highly connected and related data, for social networking sites, for routing, spatial and map applications, and for recommendation engines

Learning Objectives

- Define MongoDB
- Identify the key benefits of using MongoDB
- List and describe the most common use cases for MongoDB
- Use a mongo shell to connect to your MongoDB database and perform basic CRUD operations
- Explain why we need indexes and describe how MongoDB stores them
- Define the Aggregation Framework in MongoDB
- Explain what replication is
- Define MongoClient and perform basic CRUD operations using Python

What is MongoDB Database?

- A document and a NoSQL database



- Where data is structured in non-relational way

Order document

What are documents?

Associative arrays like JSON objects or Python dictionaries

For example, a student document

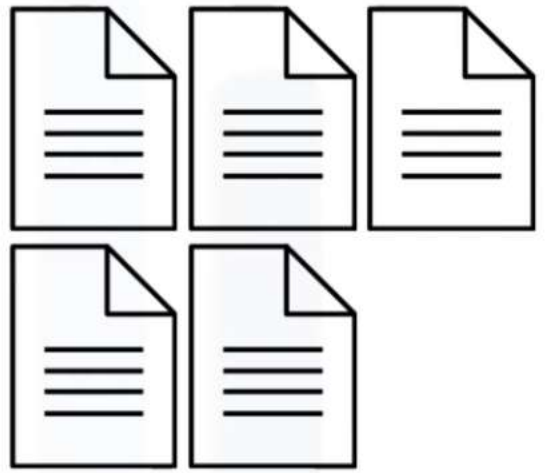
```
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@email.com",
  "studentId": 20217484
}
```

What is a Collection?

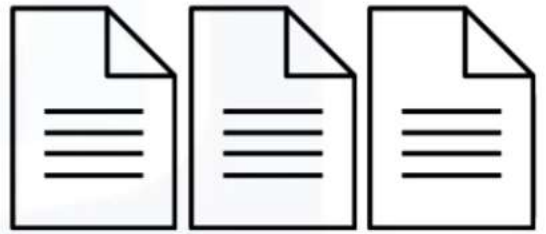
A group of stored documents

For example, all student records in Students section (**collection**)

and staff records in Employees section (**collection**)



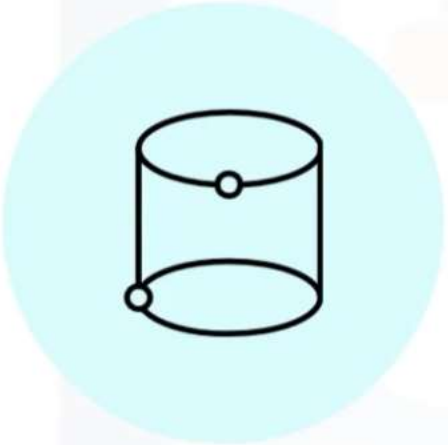
Students



Employees

What is a Database?

Database stores Collections



Students and Employees collections stored in a database called CampusManagementDB

Documents in detail - 1/2

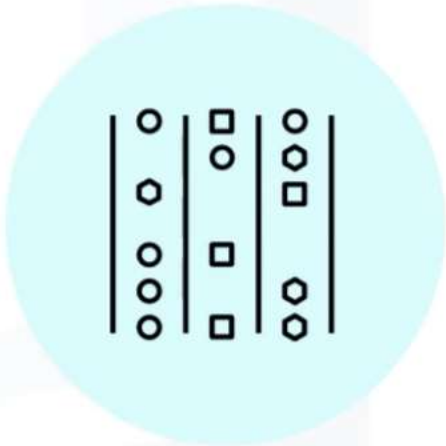
Fields in the document below:
firstName, lastName, email and studentId

```
{
  "firstName": "John",
  "lastName":  "Doe",
  "email":     "john.doe@email.com",
  "studentId": 20217484
}
```

Documents in detail - 2/2

MongoDB supports various data types:

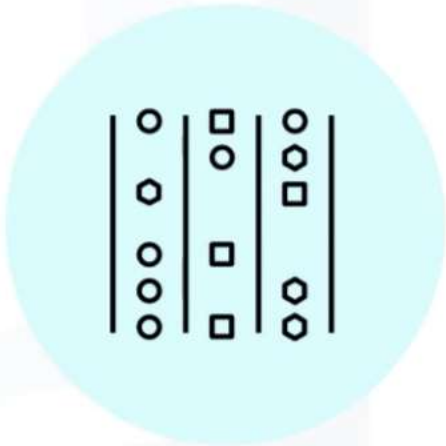
```
{
  "name": "John",
  "dateOfBirth": ISODate("2000-01-01T14:45:00.000Z"),
  "studentId": 20217484,
  "enrolled": true,
  "balance": 20.01,
  "address": {
    "city": "Stonefield",
    "country": "UK"
  },
  "interests": ["football", "skiing", "travelling"]
}
```



Documents in detail - 2/2

MongoDB supports various data types:

```
{
  "name": "John",
  "dateOfBirth": ISODate("2000-01-01T14:45:00.000Z"),
  "studentId": 20217484,
  "enrolled": true,
  "balance": 20.01,
  "address": {
    "city": "Stonefield",
    "country": "UK"
  },
  "interests": ["football", "skiing", "travelling"]
}
```



Why use MongoDB?

- Model data as you read/write, not the other way
  - Traditional relational databases: create schema first, then create tables
  - To store another field, you have to alter tables

```
{
  "orderId": "AXU-1311",
  "customer":  {
         "name": "John Doe",
         "email": "john.doe@email.com"
         }
}
```

Why use MongoDB?

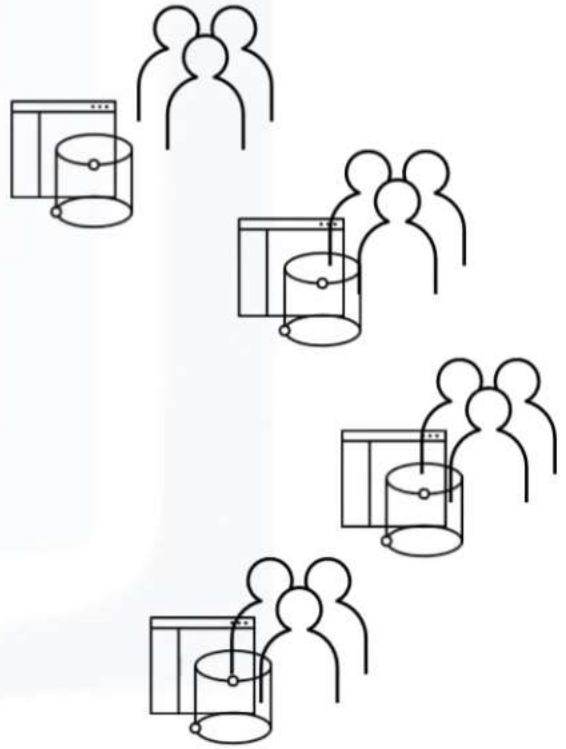
- Model data as you read/write, not the other way
  - Traditional relational databases: create schema first, then create tables
  - To store another field, you have to alter tables

```
SELECT * FROM Orders INNER JOIN Customers...


ALTER TABLE Customers...
```

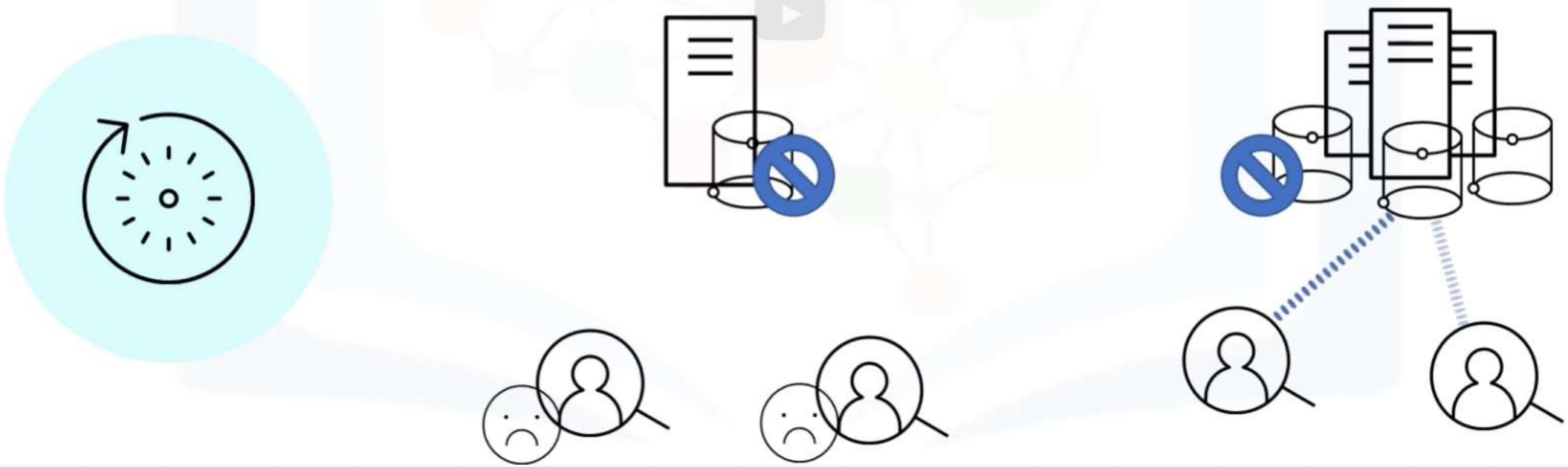
Why use MongoDB?

- Model data as you read/write, not the other way



Why use MongoDB?

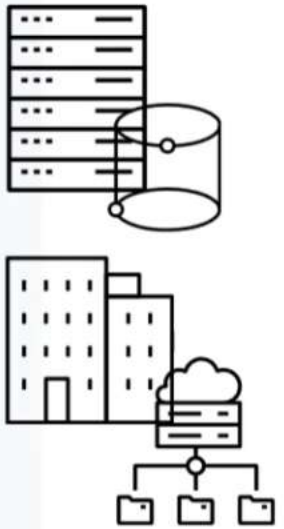
- Model data as you read/write, not the other way
- Bring structured/unstructured data
- High availability



Where to use MongoDB

MongoDB is a popular choice of database for

- Large and unstructured
- Complex
- Flexible
- Highly scalable applications
- Self-managed, hybrid, or cloud hosted



https://www.ibm.com/cloud/databases-for-mongodb

Summary

In this video, you learned that:

- MongoDB is a document and a NoSQL database
- MongoDB supports various data types
- Documents provide a flexible way to store data
- MongoDB documents of similar type are grouped into collections
- MongoDB models data as you read/write, brings structured or unstructured data, and provides high availability
- MongoDB can be used for a variety of purposes

Advantages of MongoDB

Flexibility with Schema

```
{
  "street": "10 High St",
  "city": "London",
  "postcode": "W1 1SU"
}
```

```
{
  "street": "8717 West St",
  "city": "New York",
  "zip": "10940"
}
```

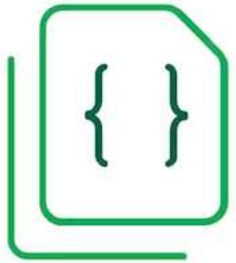
CURD


MongoDB m001 BSON vs JSON v5
Watch later
Share
mongoDB.
How Does MongoDB Store Data?
MORE VIDEOS
0:06 / 3:02

About documents

Representation

How are documents represented in memory?



Correct syntax

What is the correct syntax for documents?

JSON

JSON

JavaScript Standard Object Notation

JSON format

Start and end with curly braces `{}`

Separate each `key` and `value` with a colon `:`

Separate each `key:value` pair with a comma `,`

`"keys"` must be surrounded by quotation marks `""`

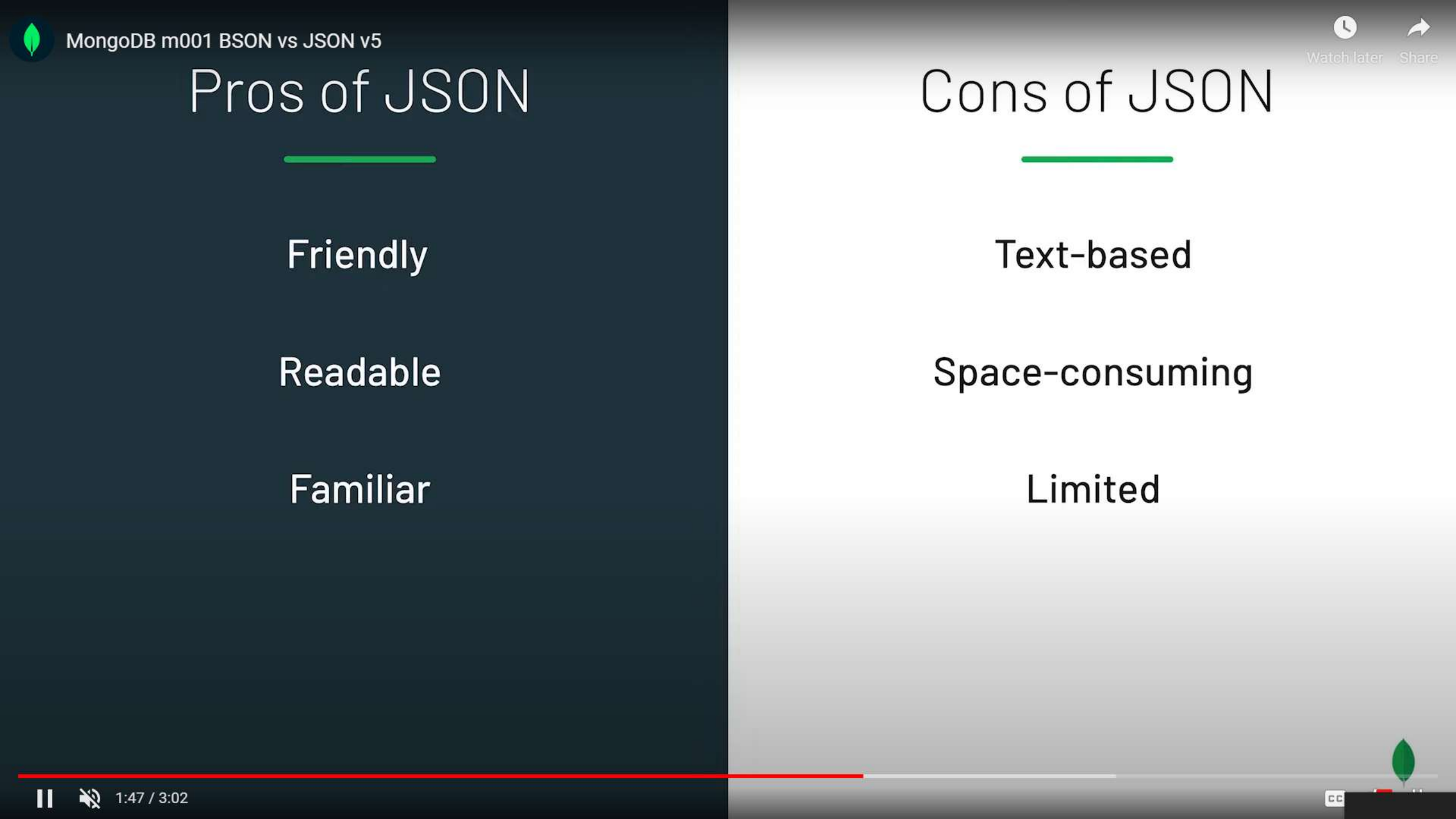
→ In MongoDB `"keys"` are called `"fields"`

```
{
  "_id" : "10021-2015-ENFO",
  "certificate_number" : 9278806,
  "business_name" : "ATLIXCO DELI",
  "date" : "Feb 20 2015",
  "result" : "No Violation Issued",
  "sector" : "Cigarette Retail - 127",
  "address" : {
      "city" : "RIDGEWOOD",
      "zip" : 11385,
      "street" : "MENAHAN ST",
      "number" : 1712
    }
}
```

```
[
  _id : "10021-2015-ENFO",
  certificate_number : 9278806,
  business_name : "ATLIXCO DELI",
  date : "Feb 20 2015",
  result : "No Violation Issued",
  sector : "Cigarette Retail - 127",
  address : {
      city : "RIDGEWOOD",
      zip : 11385,
      street : "MENAHAN ST",
      number : 1712
    }
]
```

```
{
  "_id" : "10021-2015-ENFO",
  "certificate_number" : 9278806,
  "business_name" : "ATLIXCO DELI",
  "date" : "Feb 20 2015",
  "result" : "No Violation Issued",
  "sector" : "Cigarette Retail - 127",
  "address" : {
      "city" : "RIDGEWOOD",
      "zip" : 11385,
      "street" : "MENAHAN ST",
      "number" : 1712
    }
}
```

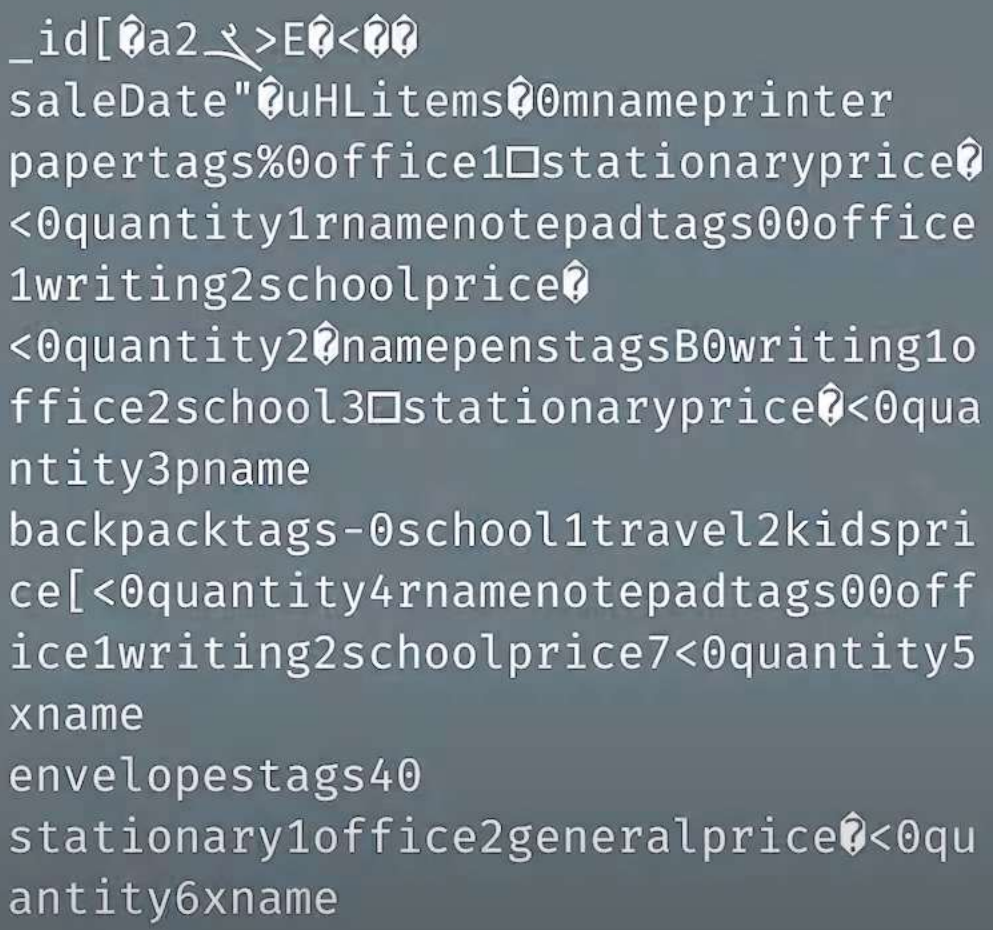
Sub-Document


MongoDB m001 BSON vs JSON v5
Watch later
Share
Pros of JSON
Friendly
Readable
Familiar
Cons of JSON
Text-based
Space-consuming
Limited
1:47 / 3:02

BSON

BSON:

Binary JSON



BSON

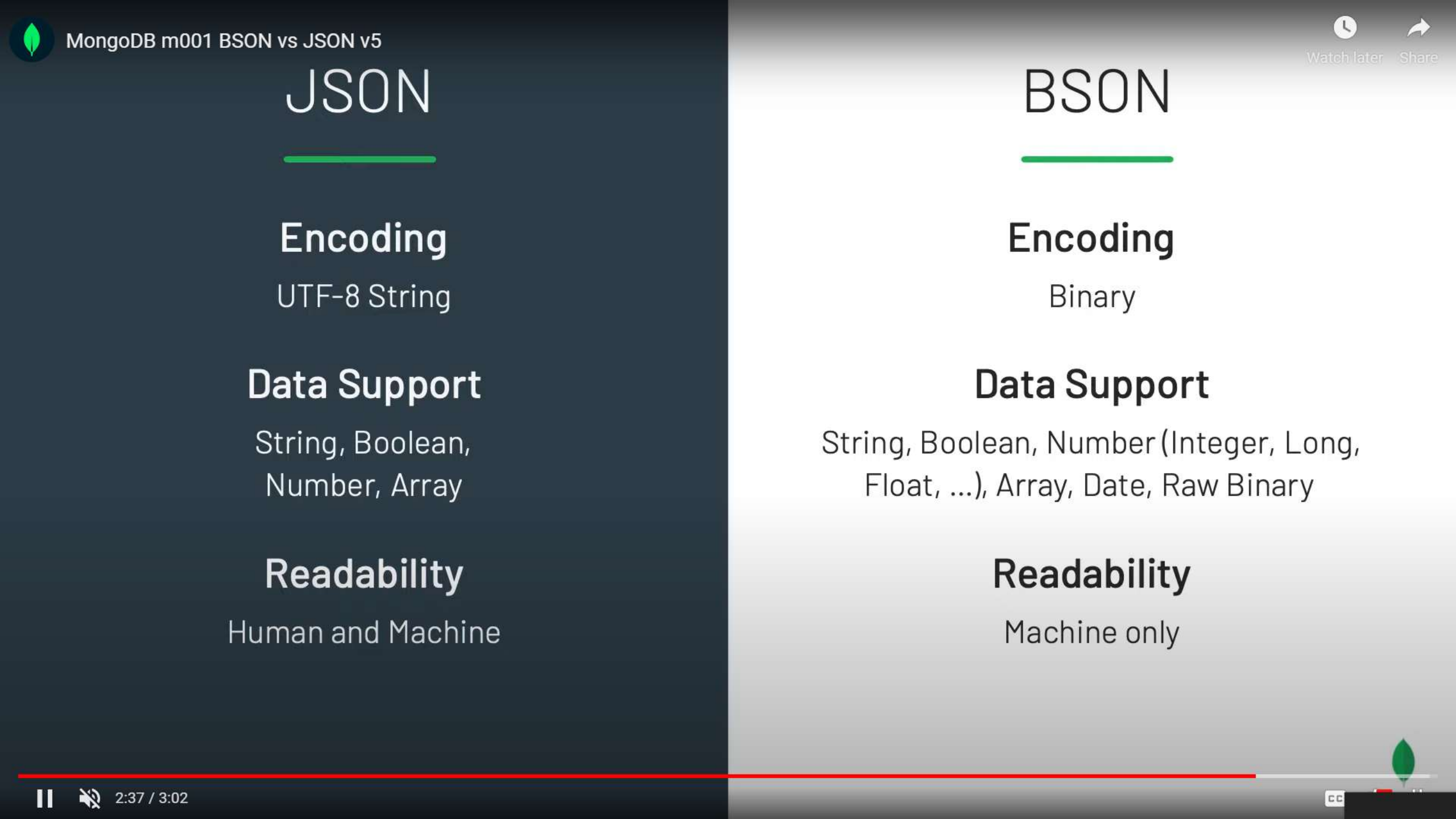
Bridges the gap between binary representation and JSON format

Optimized for:

- Speed
- Space
- Flexibility

High performance

General-purpose focus


MongoDB m001 BSON vs JSON v5
Watch later
Share
JSON
Encoding
UTF-8 String
Data Support
String, Boolean,
Number, Array
Readability
Human and Machine
BSON
Encoding
Binary
Data Support
String, Boolean, Number (Integer, Long,
Float, ...), Array, Date, Raw Binary
Readability
Machine only
2:37 / 3:02


mongoDB.
Importing and
Exporting

Interacting with the Atlas Cluster

JSON	BSON
mongoimport	mongorestore
mongoexport	mongodump

Export

```
mongodump --uri "<Atlas Cluster URI>"
```

Exports data in BSON

```
mongoexport --uri "<Atlas Cluster URI>"
            --collection=<collection name>
            --out=<filename>.json
```

Exports data in JSON

URI string

Uniform Resource Identifier

Target database name

```
mongodb+srv://user:password@clusterURI.mongodb.net
/database
```


MongoDB m001 import export v4
Watch later
Share
→ M001 mongodump --uri "mongodb+srv://m001-student:m001-mongodb-basics@sandbox. .mongodb.net/sample_supplies"
2020-08-28T13:00:48.496-0400 writing sample_supplies.sales to
2020-08-28T13:00:50.681-0400 [.....] sample_supplies.sales 101/5000 (2.0%)
2020-08-28T13:00:50.801-0400 [#####] sample_supplies.sales 5000/5000 (100.0%)
2020-08-28T13:00:50.807-0400 done dumping sample_supplies.sales (5000 documents)
→ M001 ls
dump
→ M001 cd dump
→ dump ls
sample_supplies
→ dump cd sample_supplies
→ sample_supplies ls
sales.bson sales.metadata.json
→ sample_supplies less sales.bson

```
M001 — yuliagenkina@yulia — ~/M001 — -zsh — 119×33
→  M001 mongoexport --uri="mongodb+srv://m001-student:m001-mongodb-basics@sandbox.      .mongodb.net/sample_supplies" --
collection=sales --out=sales.json
2020-08-28T13:03:09.961-0400    connected to: localhost
2020-08-28T13:03:10.758-0400    [........................]  sample_supplies.sales  0/5000  (0.0%)
2020-08-28T13:03:11.757-0400    [........................]  sample_supplies.sales  0/5000  (0.0%)
2020-08-28T13:03:12.762-0400    [........................]  sample_supplies.sales  0/5000  (0.0%)
2020-08-28T13:03:13.208-0400    [########################]  sample_supplies.sales  5000/5000  (100.0%)
2020-08-28T13:03:13.208-0400    exported 5000 records
→  M001 less sales.json
```

Import

```
mongorestore --uri "<Atlas Cluster URI>"
             --drop dump
```

Imports data in BSON dump

```
mongoimport --uri "<Atlas Cluster URI>"
            --drop=<filename>.json
```

Imports data in JSON

```
→  M001 ll
total 8240
drwxr-xr-x  3 yuliagenkina  staff    96B Aug 28 13:00 dump
-rw-r--r--  1 yuliagenkina  staff   4.0M Aug 28 13:03 sales.json
→  M001 mongorestore --uri "mongodb+srv://m001-student:m001-mongodb-basics@sandbox.[redacted].mongodb.net/sample_supplies" -
-drop dump
2020-08-28T13:06:51.899-0400    preparing collections to restore from
2020-08-28T13:06:52.086-0400    reading metadata for sample_supplies.sales from dump/sample_supplies/sales.metadata.jso
n
2020-08-28T13:06:52.192-0400    restoring sample_supplies.sales from dump/sample_supplies/sales.bson
2020-08-28T13:06:53.747-0400    no indexes to restore
2020-08-28T13:06:53.747-0400    finished restoring sample_supplies.sales (5000 documents)
2020-08-28T13:06:53.748-0400    done
→  M001 mongoimport --uri="mongodb+srv://m001-student:m001-mongodb-basics@sandbox.[redacted].mongodb.net/sample_supplies" --
drop sales.json
2020-08-28T13:08:17.959-0400    no collection specified
2020-08-28T13:08:17.960-0400    using filename 'sales' as collection
2020-08-28T13:08:18.254-0400    connected to: localhost
2020-08-28T13:08:18.366-0400    dropping: sample_supplies.sales
2020-08-28T13:08:20.206-0400    imported 5000 documents
→  M001 mongoimport --uri="mongodb+srv://m001-student:m001-mongodb-basics@sandbox.[redacted].mongodb.net/sample_supplies" --
drop sales.json --collection sales
```

Summary

Use `show dbs` and `show collections` for available namespaces

`find()` returns a cursor with documents that match the find query

`count()` returns the number of documents that match the find query

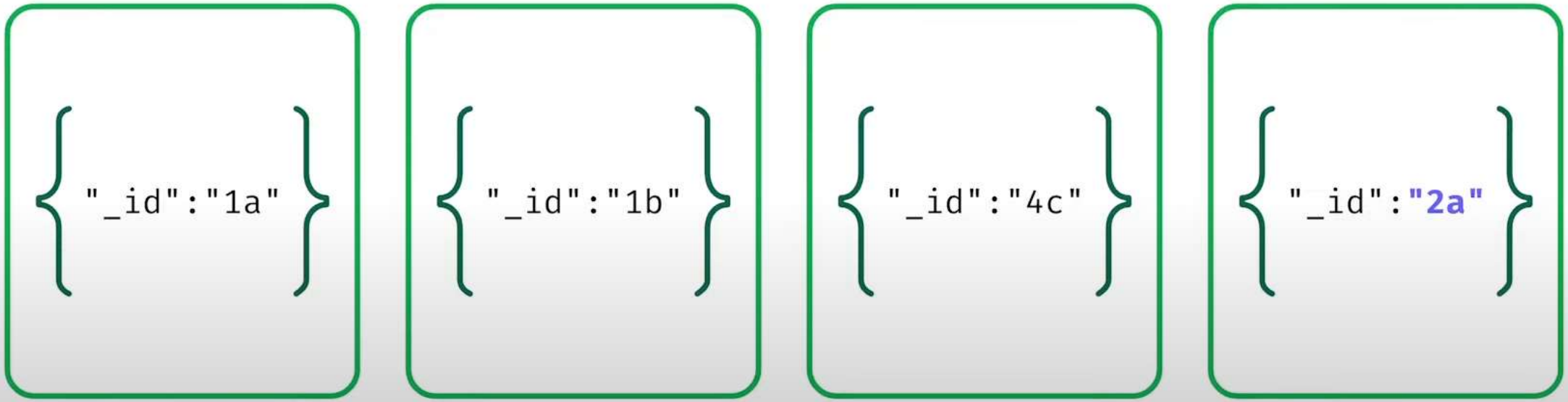
`pretty()` formats the documents in the cursor

All examples used in the lesson can be found in the lecture notes below the video.


MongoDB m001 Inserting New Documents ObjectId v4
Press Esc to exit full screen
Watch later
Share
mongoDB.
Inserting New Documents
0:07 / 5:58

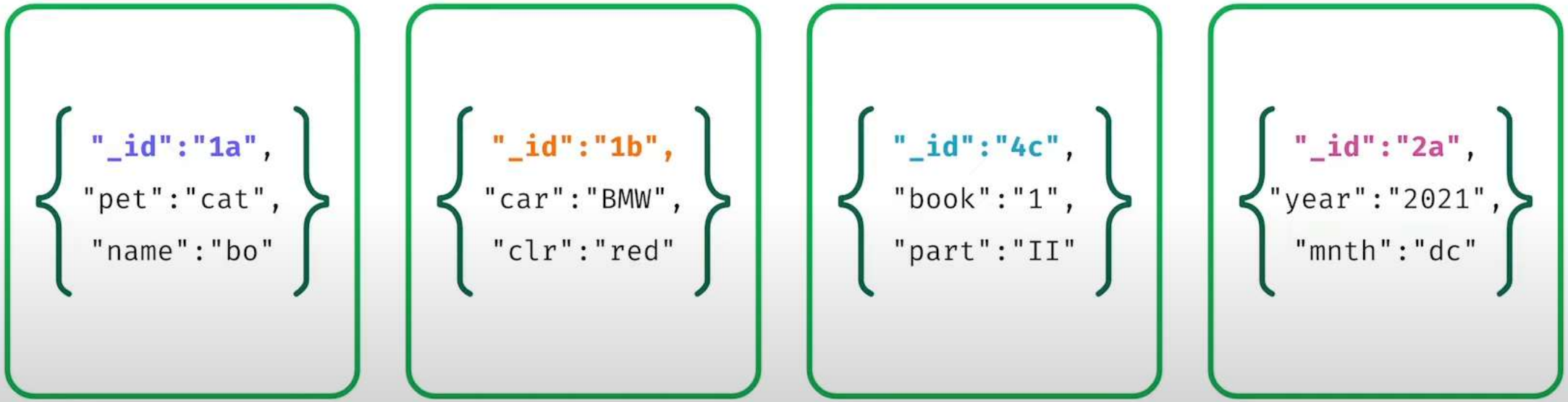
_id: why is it important

Every document must have a **unique** `_id` value.



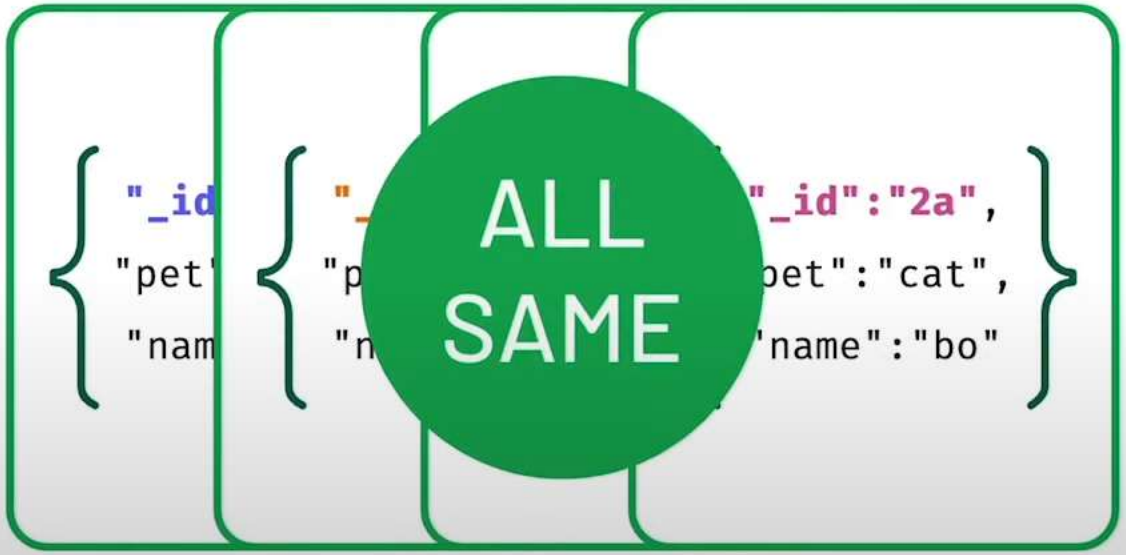
What about this?

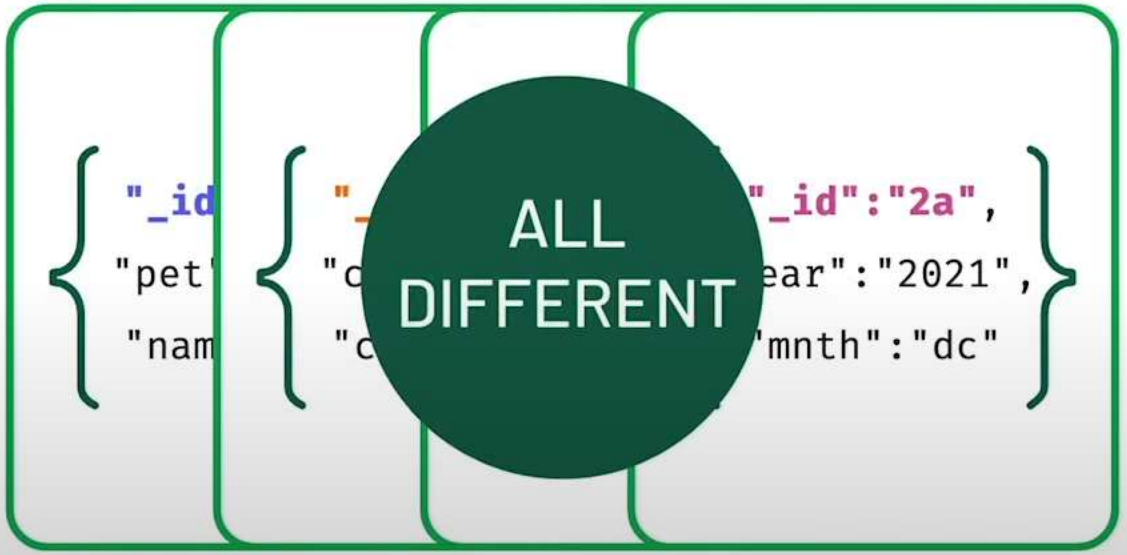
Also, yes. Every document in this collection is very different in every way.

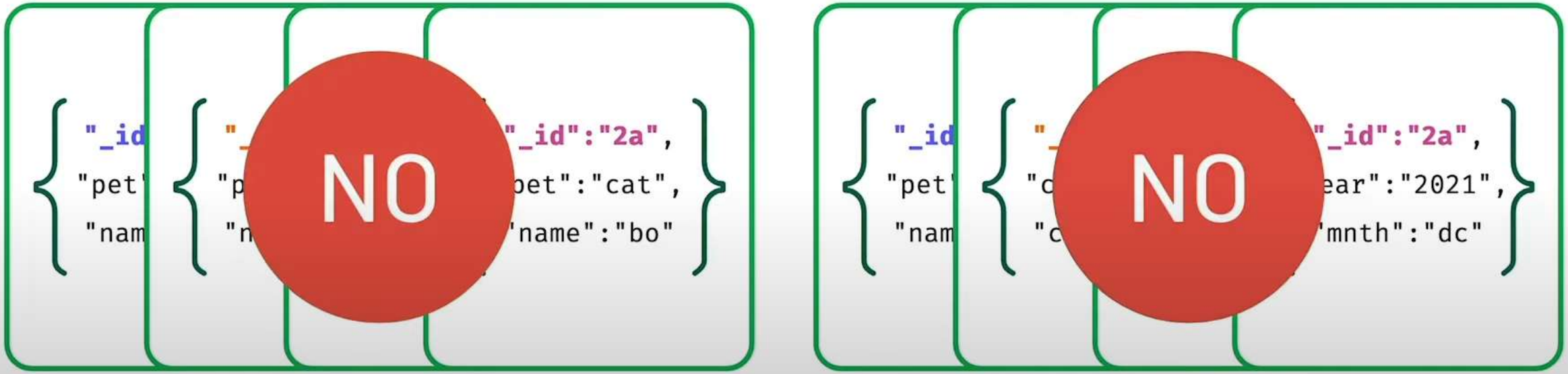


Is either a good idea?

No, not usually.






"_id
"pet
"nam
"_id":"2a",
pet":"cat",
name":"bo"
NO
"_id
"pet
"nam
"_id":"2a",
ear":"2021",
mnth":"dc"
NO

ObjectId()

ObjectId()

Default value for the `_id` field unless otherwise specified.

Examples

```
"_id": ObjectId("5ec5f1b710ca9222e6a46cab")

"_id": "710ca922"

"_id": "101-EXG-27"
```

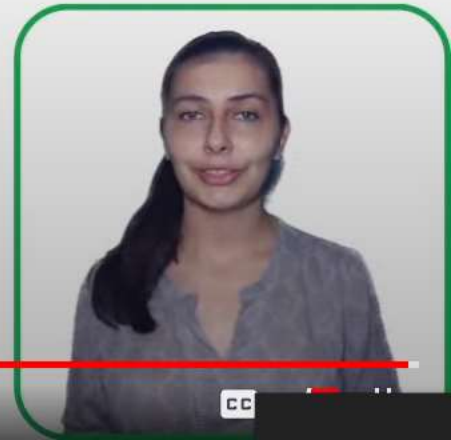
Summary

`"_id"` unique identifier for a document in a collection.

`"_id"` required in every MongoDB document .

`ObjectId()` is the default value for the `"_id"` field unless otherwise specified.

How to insert documents into a collection using the Data Explorer.

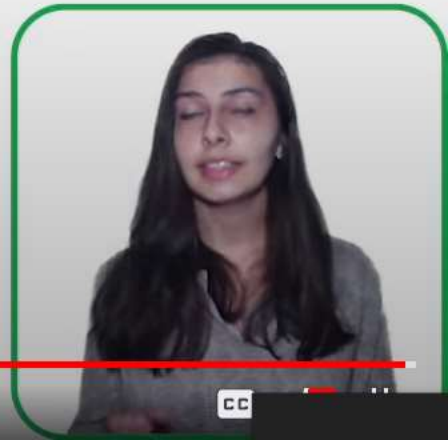



MongoDB m001 Insert Errors v2
Press Esc to exit full screen
mongoDB.
Watch later
Share
Inserting Documents: Errors
0:02 / 4:46

Summary

Identical documents can exist in the same collection as long as their `_id` values are different.

MongoDB has schema validation functionality allows you to enforce document structure.



Update operators

```
{"$inc": {"pop": 10, "<field2>": <increment value>, … }}
```

increments field value by a specified amount.

```
{"$set": {"pop": 17630, "<field2>": <new value>, … }}
```

sets field value to a new specified value.

```
{ $push: { <field1>: <value1>, ... } }
```

adds an element to an array field.

Query Operators: Comparison

MQL operators

Update Operators

Enable us to modify date in the database

Example: `$inc, $set, $unset`

Query Operators

Provide additional ways to locate data within the database

$ has multiple uses

Precedes MQL operators

Precedes Aggregation pipeline stages

Allows Access to Field Values

Comparison Operators

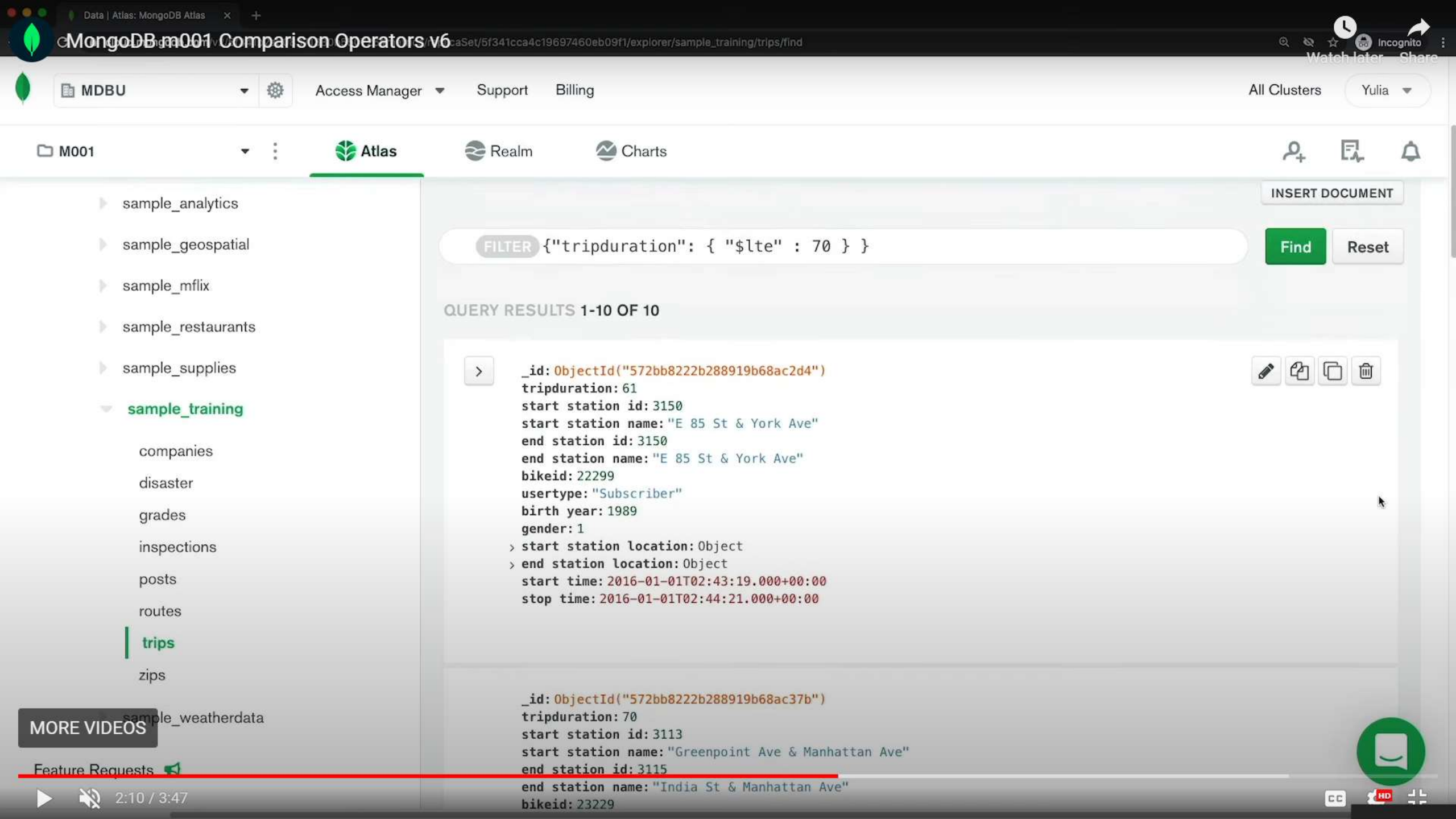
- `$eq` (=)
- `$gt` (>)
- `$gte` (≥)
- `$lt` (<)
- `$lte` (≤)
- `$ne` (≠)
- `$in` (∈)
- `$nin` (∉)

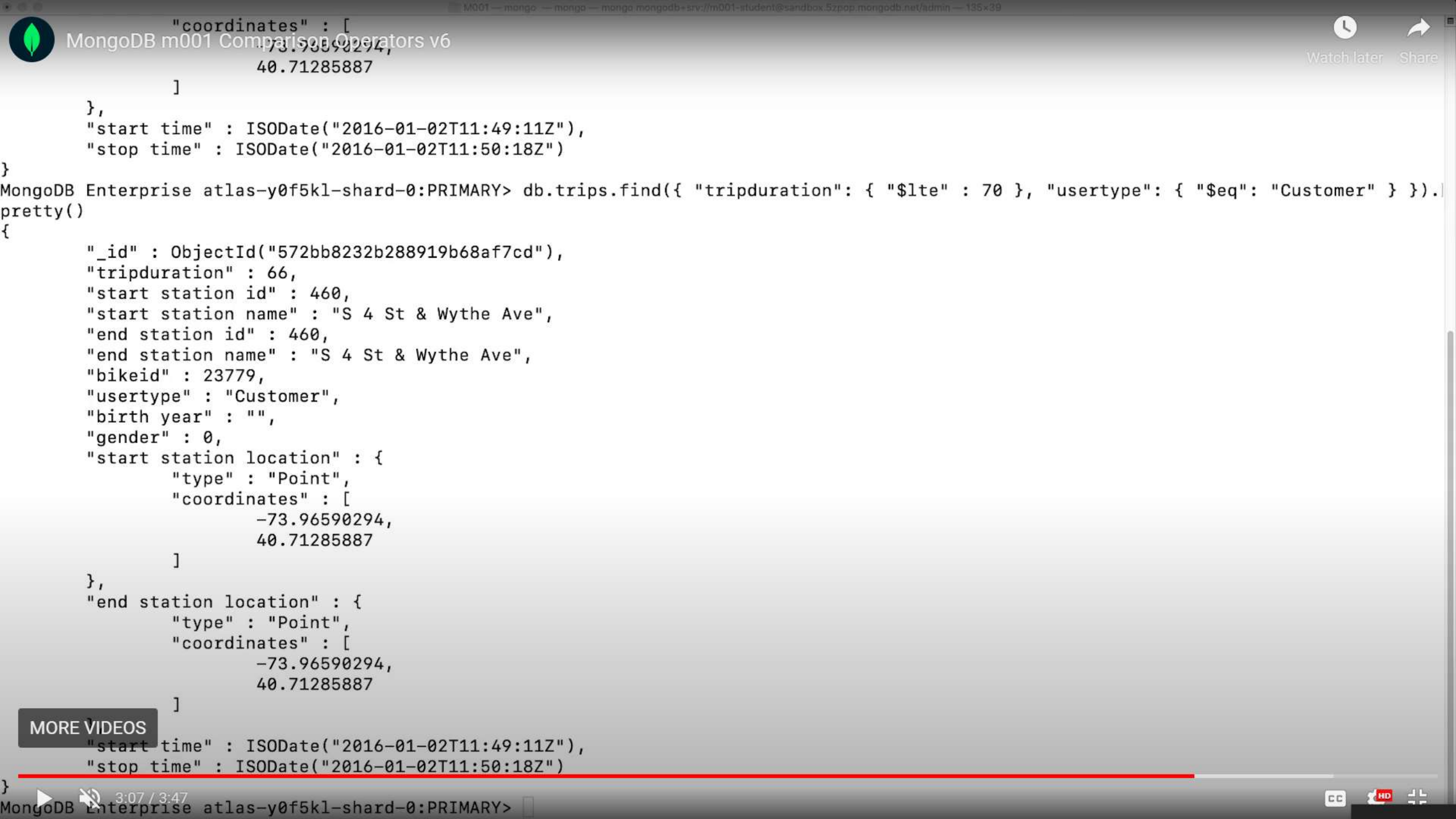
Comparison operators

`$eq`	=	EQual to	`$ne`	=	Not Equal to
`$gt`	>	Greater Than	`$lt`	<	Less Than
`$gte`	≥	Greater Than or Equal to	`$lte`	≤	Less Than or Equal to

```
{ <field>: { <operator>: <value> } }
```

MongoDB m001 Comparison Operators v6




MongoDB m001 Comparison Operators v6
Watch later
Share
"coordinates" : [
-73.96590294,
40.71285887
]
},
"start time" : ISODate("2016-01-02T11:49:11Z"),
"stop time" : ISODate("2016-01-02T11:50:18Z")
}
MongoDB Enterprise atlas-y0f5kl-shard-0:PRIMARY> db.trips.find({ "tripduration": { "$lte" : 70 }, "usertype": { "$eq": "Customer" } }).pretty()
{
"_id" : ObjectId("572bb8232b288919b68af7cd"),
"tripduration" : 66,
"start station id" : 460,
"start station name" : "S 4 St & Wythe Ave",
"end station id" : 460,
"end station name" : "S 4 St & Wythe Ave",
"bikeid" : 23779,
"usertype" : "Customer",
"birth year" : "",
"gender" : 0,
"start station location" : {
"type" : "Point",
"coordinates" : [
-73.96590294,
40.71285887
]
},
"end station location" : {
"type" : "Point",
"coordinates" : [
-73.96590294,
40.71285887
]
},
"start time" : ISODate("2016-01-02T11:49:11Z"),
"stop time" : ISODate("2016-01-02T11:50:18Z")
}
MongoDB Enterprise atlas-y0f5kl-shard-0:PRIMARY>
MORE VIDEOS
3:07 / 3:47

Comparison operators

Query operators provide additional ways to locate data within the database.

Comparison operators specifically allow us to find data within a certain range.

```
{ <field>: { <operator>: <value> } }
```

`$eq` is used as the default operator when an operator is not specified.


MongoDB m001 Logic Operators v4
Watch later
Share
mongoDB.
Query Operators:
Logic
0:01 / 6:13

Logic operators

`$and`	Match **all** of the specified query clauses
`$or`	**At least one** of the query clauses is matched
`$nor`	**Fail to match** both given clauses
`$not`	**Negates** the query requirement

Logic operators

$and

$or

$nor

```
{<operator> : [{statement1},{statement2},...]}
```

Logic operators

$and

$or {<operator> : [{statement1},{statement2},...]}

$nor

$not {$not: {statement}}

Implicit $and

$and is used as the default operator when an operator is not specified.

```
{sector : "Mobile Food Vendor - 881",   result: "Warning"}
```

Is the same as:

```
{"$and": [{sector : "Mobile Food Vendor - 881"}, {result:"Warning"}]}
```

Explicit $and

When you need to include the same operator more than once in a query

Using the `routes` collection find out how many `CR2` and `A81` airplanes come through the `KZN` airport?

```
{"$or" :[{dst_airport : "KZN"},{src_airport : "KZN"}]}
```

and

```
{"$or" :[{airplane : "CR2"},{airplane : "A81"}]}
```

Explicit $and

When you need to include the same operator more than once in a query

Using the `routes` collection find out how many `CR2` and `A81` airplanes come through the `KZN` airport?

```
{"$or" :[{dst_airport : "KZN"},{src_airport : "KZN"}]}
```

and

```
{"$or" :[{airplane : "CR2"},{airplane : "A81"}]}
```

{ "$and": [{ "$or" :[{ "dst_airport": "KZN" },{ "src_airport": "KZN" }] },{ "$or" :[{ "airplane": "CR2" },{ "airplane": "A81" }] }]}

Logic operators

Logic operators allow us to be more granular in our search for data.

Syntax

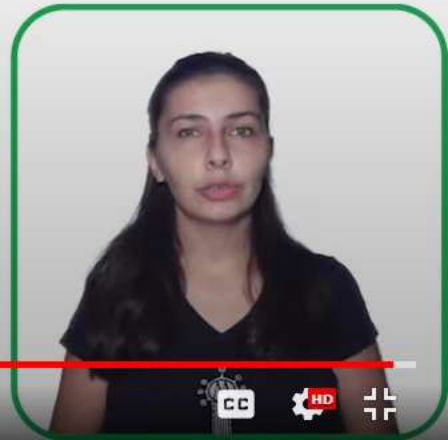
```
{ "$<operator>": [{ <clause1> }, {<clause2>}, … ] }
```

Syntax for `$not:`

```
{$not: {<clause>}}
```

`$and` is used as the default operator when an operator is not specified.

Explicitly use `$and` when you need to include the same operator more than once in a query.



Chapter 4: Advanced CRUD Operations

Quiz 1: Logic Operators

Problem:

To complete this exercise connect to your Atlas cluster using the in-browser IDE space at the end of this chapter.

How many businesses in the `sample_training.inspections` dataset have the inspection result `"Out of Business"` and belong to the `"Home Improvement Contractor - 100"` sector?

Enter the exact numeric value of the result that you get into the response field.

Attempts Remaining:

Enter answer here:

```
db.inspections.find({ "result": "Out of Business", "sector": "Home Improvement Contractor - 100" }).count()
```


MongoDB m001 Expressive Query Operator v2
Watch later
Share
mongoDB.
Expressive Query Operator
0:04 / 5:22

Expressive $expr

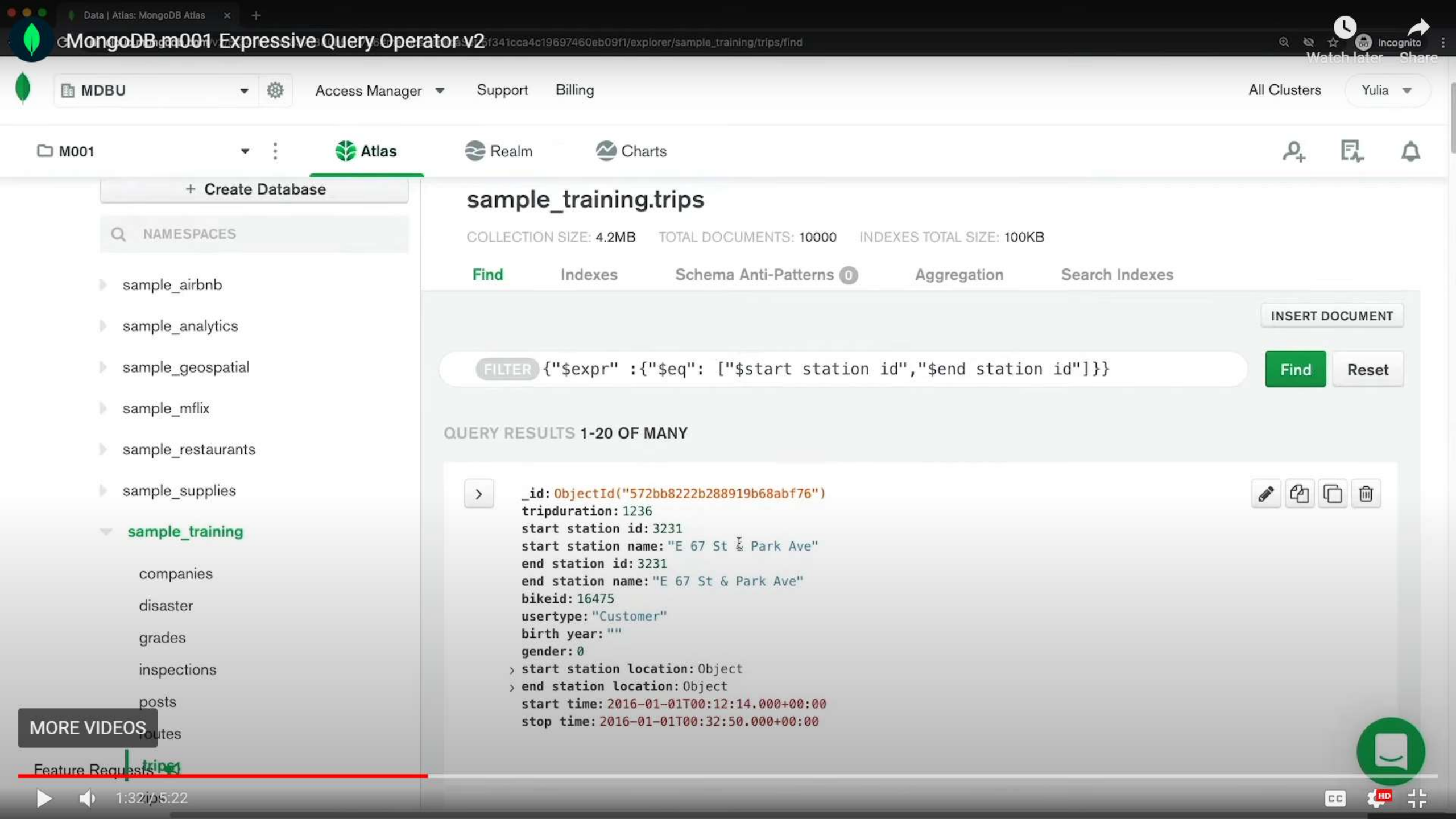
$expr allows the use of aggregation expressions within the query language

```
{ $expr: { <expression> } }
```

$expr allows us to use variables and conditional statements

Woo-hoo!

Can we compare fields **within the same document** to each other?


MongoDB m001 Expressive Query Operator v2
MDBU
Access Manager
Support
Billing
All Clusters
Yulia
M001
Atlas
Realm
Charts
+ Create Database
NAMESPACES
sample_airbnb
sample_analytics
sample_geospatial
sample_mflix
sample_restaurants
sample_supplies
sample_training
companies
disaster
grades
inspections
posts
routes
trips
sample_training.trips
COLLECTION SIZE: 4.2MB
TOTAL DOCUMENTS: 10000
INDEXES TOTAL SIZE: 100KB
Find
Indexes
Schema Anti-Patterns 0
Aggregation
Search Indexes
INSERT DOCUMENT
FILTER {"$expr" :{"$eq": ["$start station id","$end station id"]}}
Find
Reset
QUERY RESULTS 1-20 OF MANY
_id: ObjectId("572bb8222b288919b68abf76")
tripduration: 1236
start station id: 3231
start station name: "E 67 St & Park Ave"
end station id: 3231
end station name: "E 67 St & Park Ave"
bikeid: 16475
usertype: "Customer"
birth year: ""
gender: 0
start station location: Object
end station location: Object
start time: 2016-01-01T00:12:14.000+00:00
stop time: 2016-01-01T00:32:50.000+00:00
MORE VIDEOS
Feature Requests
1:32 / 5:22

$ addressing the field value

```
{"$expr" :{"$eq": ["$start station id","$end station id"]}}
```

```
{    "_id":"572bb8222b288919b68abf70",
     "tripduration": 110,
     "start station id": 439,
     "start station name":"E 4 St & 2 Ave",
     "end station id": 439,
     "end station name":"E 4 St & 2 Ave",
 …
     "start station location":
         { "type":"Point",
           "coordinates": [-73.98978041,
                           40.7262807]
         },
```

A closer look

```
{"$expr": {
        "$and": [
                { "$gt": ["$tripduration", 1200]},
                { "$eq": ["$end station id", "$start station id" ]}
            ]
        }
}
```

A closer look

```
{"$expr": {
        "$and": [
                { "$gt": ["$tripduration", 1200]},
                { "$eq": ["$end station id", "$start station id" ]}
              ]
        }
}
```

MQL syntax: { <field>: { **<operator>**: <value> } }

Aggregation syntax: { **<operator>**: { <field>, <value> } }

$expr: Expressive Query Operator

Allows for more complex queries and for comparing fields within a document.

The $ can be used to access the field value

Syntax for comparison operators using aggregation:

```
{ <operator>: { <field>, <value> } }
```