MongoDB Tutorial

MongoDB Tutorial

MongoDB tutorial provides basic and advanced concepts of SQL. Our MongoDB tutorial is designed for beginners and professionals.

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.

Our MongoDB tutorial includes all topics of MongoDB database such as insert documents, update documents, delete documents, query documents, projection, sort() and limit() methods, create a collection, drop collection, etc. There are also given MongoDB interview questions to help you better understand the MongoDB database.

What is MongoDB

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

In simple words, you can say that - Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.

MongoDB is available under General Public license for free, and it is also available under Commercial license from the manufacturer.

The manufacturing company 10gen has defined MongoDB as:

"MongoDB is a scalable, open source, high performance, document-oriented database." - 10gen

MongoDB was designed to work with commodity servers. Now it is used by the company of all sizes, across all industry.

History of MongoDB

The initial development of MongoDB began in 2007 when the company was building a platform as a service similar to window azure.

Window azure is a cloud computing platform and infrastructure, created by Microsoft, to build, deploy and manage applications and service through a global network.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc. It was initially developed as a PAAS (Platform as a Service). Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

MongoDB2.4.9 was the latest and stable version which was released on January 10, 2014.

Purpose of Building MongoDB

It may be a very genuine question that - "what was the need of MongoDB although there were many databases in action?"

There is a simple answer:

All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.

The primary purpose of building MongoDB is:

- Scalability
- Performance
- High Availability
- Scaling from single server deployments to large, complex multi-site architectures.
- Key points of MongoDB
- Develop Faster
- Deploy Easier
- Scale Bigger

First of all, we should know what is document oriented database?

Example of Document-Oriented Database

MongoDB is a document-oriented database. It is a key feature of MongoDB. It offers a document-oriented storage. It is very simple you can program it easily.

MongoDB stores data as documents, so it is known as document-oriented database.

1. FirstName = "John",

2. Address = "Detroit",
3. Spouse = [{**Name**: "Angela"}].

4. FirstName ="John",
5. Address = "Wick"

There are two different documents (separated by ".").

Storing data in this manner is called as document-oriented database.

Mongo DB falls into a class of databases that calls Document Oriented Databases. There is also a broad category of database known as No SQL Databases.

Features of MongoDB

These are some important features of MongoDB:

1. Support ad hoc queries

In MongoDB, you can search by field, range query and it also supports regular expression searches.

2. Indexing

You can index any field in a document.

3. Replication

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

4. Duplication of data

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

5. Load balancing

It has an automatic load balancing configuration because of data placed in shards.

Provides high performance.

- JSON data model with dynamic schemas
- Auto-sharding for horizontal scalability
- Built in replication for high availability
- Now a day many companies using MongoDB to create new types of applications, improve performance and availability.

Prerequisite

Before learning MongoDB, you must have the basic knowledge of SQL and OOPs.

A MongoDB Document

Records in a MongoDB database are called documents, and the field values may include numbers, strings, booleans, arrays, or even nested documents.

Example Document

```
{
        title: "Post Title 1",
        body: "Body of post.",
        category: "News",
        likes: 1,
        tags: ["news", "events"],
        date: Date()
}
```

How to set up the MongoDB environment

1. C:\Program Files\MongoDB\bin\mongod.exe

This will start the mongoDB database process. If you get a message "waiting for connection" in the console output, it indicates that the mongodb.exe process is running successfully.

For example:

When you connect to the MongoDB through the mongo.exe shell, you should follow these steps:

1. Open another command prompt.
2. At the time of connecting, specify the data directory if necessary.

> ***Note: If you use the default data directory while MongoDB installation, there is no need to specify the data directory.***

For example:

1. C:\mongodb\bin\mongo.exe

Install MongoDB Shell (mongosh)

There are many ways to connect to your MongoDB database.

We will start by using the MongoDB Shell, `mongosh`.

Use the official instructions to install `mongosh` on your operating system.

To verify that it has been installed properly, open your terminal and type:

```
mongosh --version
```

You should see that the latest verion is installed.

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

MongoDB `mongosh` Create Collection

Create Collection using `mongosh`

- There are 2 ways to create a collection.

Method 1

- You can create a collection using the `createCollection()` database method.

Example

```
db.createCollection("posts")
```

The createCollection() Method

- MongoDB **db.createCollection(name, options)** is used to create collection.
- Syntax
- Basic syntax of **createCollection()** command is as follows –

```
db.createCollection(name, options)
```

- In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Method 2

- You can also create a collection during the `insert` process.

Example

- We are here assuming `object` is a valid JavaScript object containing post data:

```
db.posts.insertOne(object)
```

- This will create the "posts" collection if it does not already exist.
- **Remember:** In MongoDB, a collection is not actually created until it gets content!

MongoDB `mongosh` Insert

Insert Documents

- There are 2 methods to insert documents into a MongoDB database.

`insertOne()`

- To insert a single document, use the `insertOne()` method.
- This method inserts a single object into the database.
- **Note:** When typing in the shell, after opening an object with curly braces "{" you can press enter to start a new line in the editor without executing the command. The command will execute when you press enter after closing the braces.

Example

```
db.posts.insertOne({
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
})
```

- **Note:** If you try to insert documents into a collection that does not exist, MongoDB will create the collection automatically.

`insertMany()`

- To insert multiple documents at once, use the `insertMany()` method.
- This method inserts an array of objects into the database.

Example

```
db.posts.insertMany([
  {
    title: "Post Title 2",
    body: "Body of post.",
    category: "Event",
    likes: 2,
    tags: ["news", "events"],
    date: Date()
  },
  {
    title: "Post Title 3",
    body: "Body of post.",
    category: "Technology",
    likes: 3,
    tags: ["news", "events"],
    date: Date()
```

```
    },
    {
      title: "Post Title 4",
      body: "Body of post.",
      category: "Event",
      likes: 4,
      tags: ["news", "events"],
      date: Date()
    }
  ])
```

The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows –

```
db.dropDatabase()
```

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, **show dbs**.

```
>show dbs
local     0.78125GB
mydb      0.23012GB
test      0.23012GB
>
```

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows –

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local     0.78125GB
test      0.23012GB
>
```

MongoDB `mongosh` Find

Find Data

There are 2 methods to find and select data from a MongoDB collection, `find()` and `findOne()`.

`find()`

To select data from a collection in MongoDB, we can use the `find()` method.

This method accepts a query object. If left empty, all documents will be returned.

Example

```
db.posts.find()
```

`findOne()`

To select only one document, we can use the `findOne()` method.

This method accepts a query object. If left empty, it will return the first document it finds.

Note: This method only returns the first match it finds.

Example

```
db.posts.findOne()
```

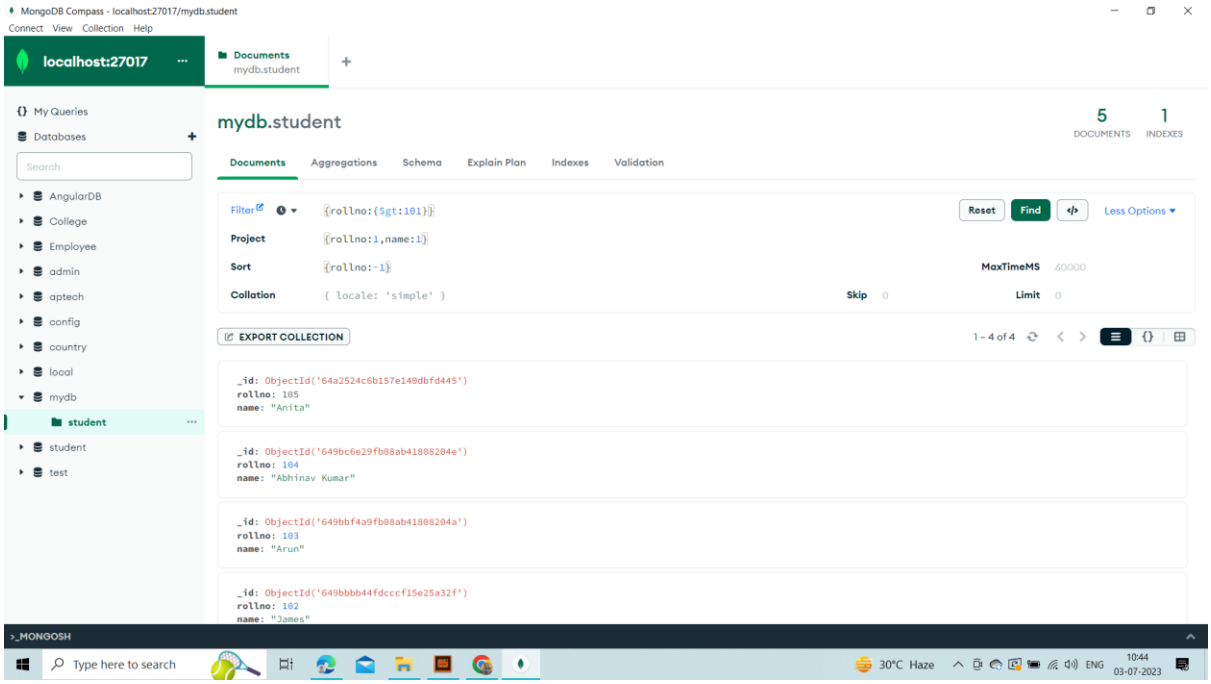
Querying Data

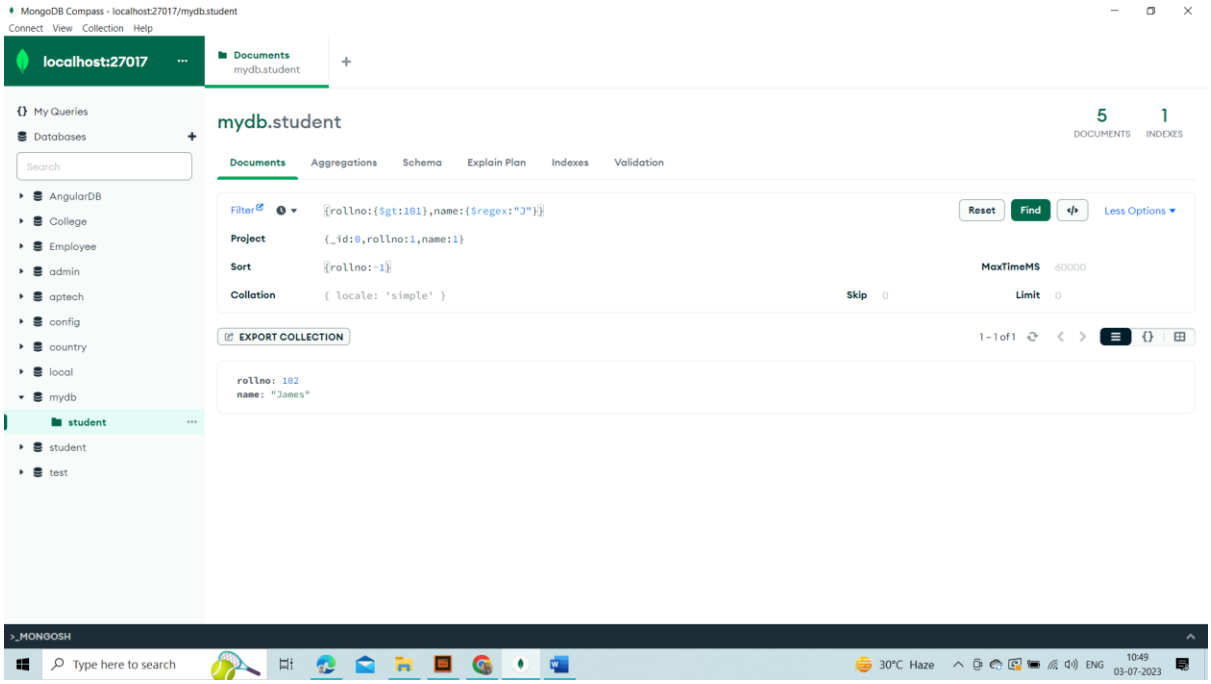
To query, or filter, data we can include a query in our `find()` or `findOne()` methods.

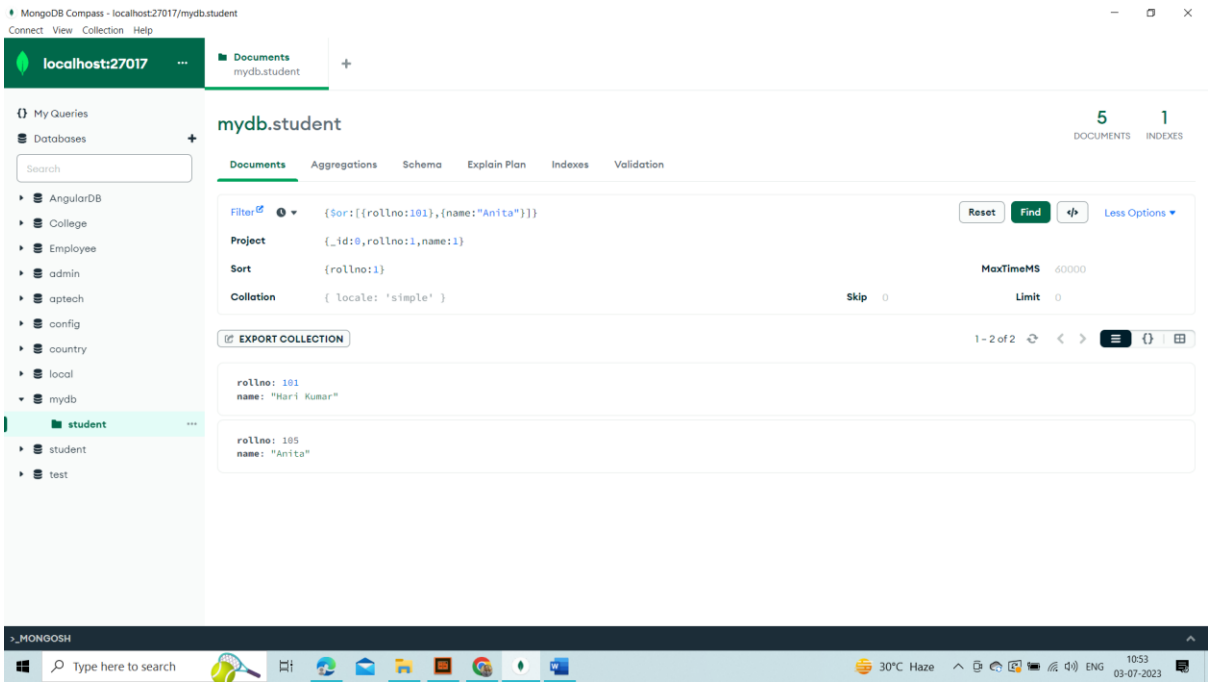
Example

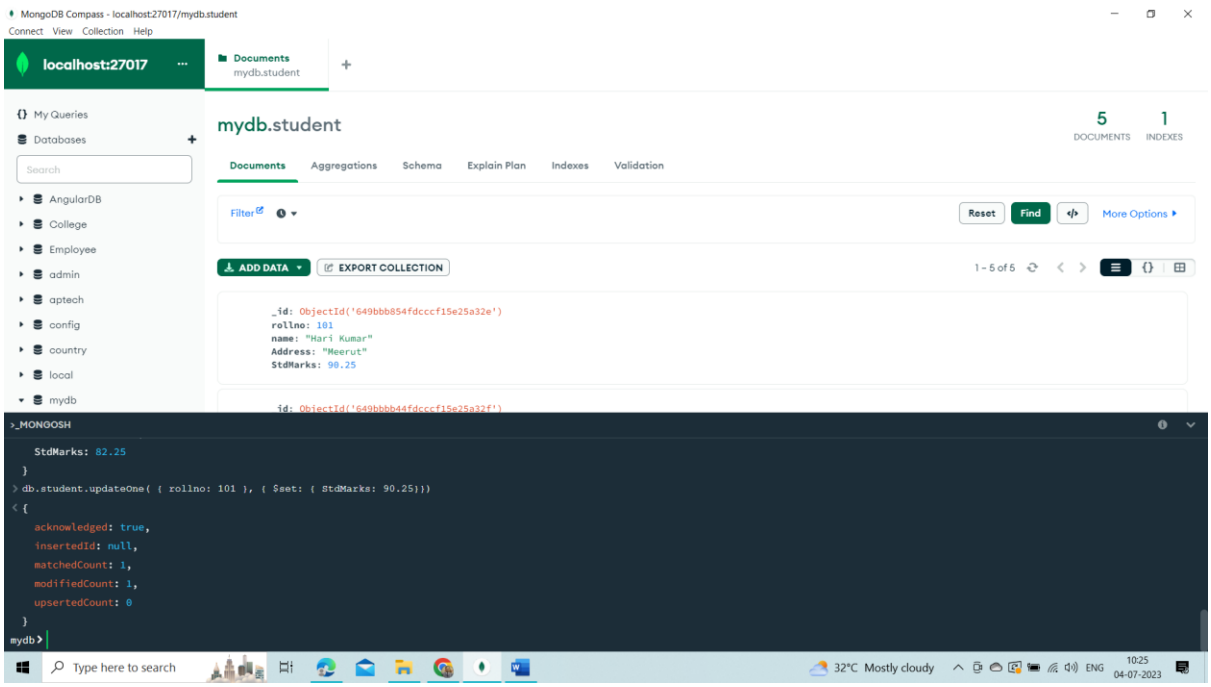
```
db.posts.find( {category: "News"} )
```

Projection


MongoDB Compass - localhost:27017/mydb.student
Connect View Collection Help
localhost:27017
Documents
mydb.student
My Queries
Databases
Search
AngularDB
College
Employee
admin
aptech
config
country
local
mydb
student
student
test
mydb.student
5
DOCUMENTS
1
INDEXES
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{rollno:{$gt:101}}
Reset
Find
Less Options
Project
{rollno:1,name:1}
Sort
{rollno:-1}
MaxTimeMS
60000
Collation
{ locale: 'simple' }
Skip
0
Limit
0
EXPORT COLLECTION
1 - 4 of 4
_id: ObjectId('64a2524c6b157e140dbfd445')
rollno: 105
name: "Anita"
_id: ObjectId('649bc6e29fb08ab41808204e')
rollno: 104
name: "Abhinav Kumar"
_id: ObjectId('649bbf4a9fb08ab41808204a')
rollno: 103
name: "Arun"
_id: ObjectId('649bbbb44fdcccf15e25a32f')
rollno: 102
name: "James"
>_MONGOSH
Type here to search
30°C Haze
ENG
10:44
03-07-2023


MongoDB Compass - localhost:27017/mydb.student
Connect View Collection Help
localhost:27017
Documents
mydb.student
My Queries
Databases
Search
AngularDB
College
Employee
admin
aptech
config
country
local
mydb
student
student
test
mydb.student
5
DOCUMENTS
1
INDEXES
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{rollno:{$gt:101},name:{$regex:"J"}}
Reset
Find
Less Options
Project
{_id:0,rollno:1,name:1}
Sort
{rollno:-1}
MaxTimeMS
60000
Collation
{ locale: 'simple' }
Skip
0
Limit
0
EXPORT COLLECTION
1 - 1 of 1
rollno: 102
name: "James"
>_MONGOSH
Type here to search
30°C Haze
ENG
10:49
03-07-2023





Both find methods accept a second parameter called `projection`.

This parameter is an `object` that describes which fields to include in the results.

Note: This parameter is optional. If omitted, all fields will be included in the results.

Example

This example will only display the `title` and `date` fields in the results.

```
db.posts.find({}, {title: 1, date: 1})
```

Notice that the `_id` field is also included. This field is always included unless specifically excluded.

We use a `1` to include a field and `0` to exclude a field.

Example

This time, let's exclude the `_id` field.

```
db.posts.find({}, {_id: 0, title: 1, date: 1})
```

Note: You cannot use both 0 and 1 in the same object. The only exception is the `_id` field. You should either specify the fields you would like to include or the fields you would like to exclude.

Let's exclude the date category field. All other fields will be included in the results.

Example

```
db.posts.find({}, {category: 0})
```

We will get an error if we try to specify both 0 and 1 in the same object.

Example

```
db.posts.find({}, {title: 1, date: 0})
```

Search Query

```
Employee> db.Empinfo.find()
[
  {
    _id: ObjectId("63c0df6c1c3d68b027325a32"),
    empid: 101,
    title: 'IT Company Employee',
    salary: 41000
  },
  {
    _id: ObjectId("63c0e07a1c3d68b027325a34"),
    empid: 102,
    title: 'Sales Company Employee',
    salary: 20000
  }
]
Employee>
```

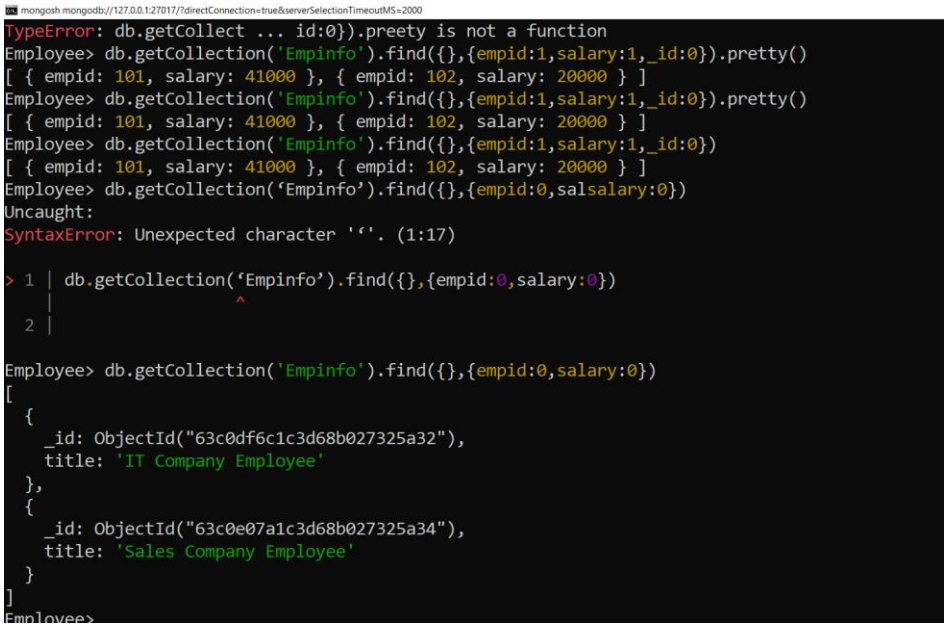
Search db.getCollection(‘Empinfo’).find({})

Search db.getCollection(‘Empinfo’).find({},{empid:1,salary:1})

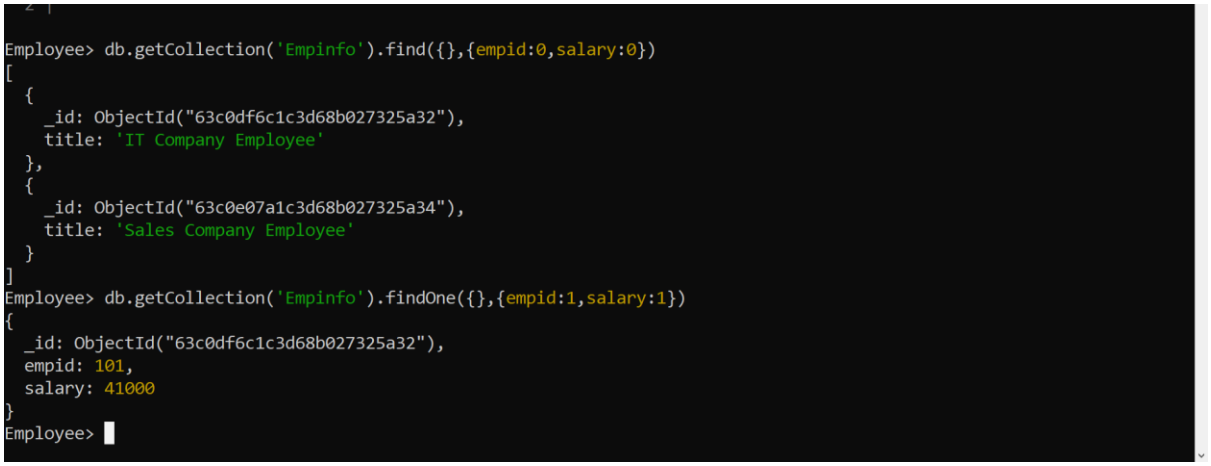
Search db.getCollection(‘Empinfo’).find({},{empid:1,salary:1,_id:0})

Zero means expect given field name

Search db.getCollection(‘Empinfo’).find({},{empid:0,salary:0})



db.getCollection(‘Empinfo’).findOne({},{empid:1,salary:1})



Use 1- true, o- False for sorting in ascending

Logical Condition operator

db.getCollection(‘Empinfo’).find({salary:{$lt:10000}});

Regular expression

db.getCollection(‘Empinfo’).find({name:{$regex:”p”}});

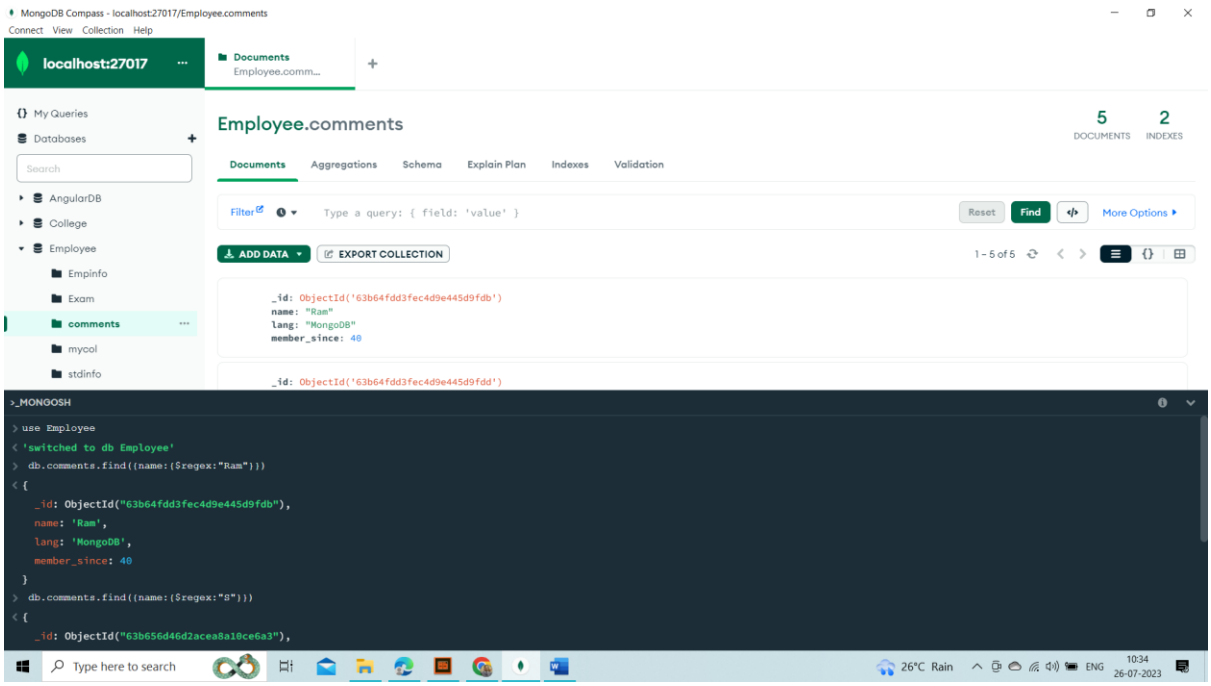
Logical Condition

db.getCollection(‘Empinfo’).find({$and:[{empid:101},{salary:20000}]});

db.getCollection(‘Empinfo’).find({$or:[{empid:101},{salary:20000}]});

salary:0 use for salary hide

db.getCollection(‘Empinfo’).find({},{salary:0}]});




```
db.comments.find({name:{$regex:"Ram"}})
db.comments.find({name:{$regex:"S"}})
```

##############--################

MongoDB mongosh Update

Update Document

To update an existing document we can use the `updateOne()` or `updateMany()` methods.

The first parameter is a query object to define which document or documents should be updated.

The second parameter is an object defining the updated data.

updateOne()

The `updateOne()` method will update the first document that is found matching the provided query.

Let's see what the "like" count for the post with the title of "Post Title 1":

Example

```
db.posts.find( { title: "Post Title 1" } )
```

Now let's update the "likes" on this post to 2. To do this, we need to use the `$set` operator.

Example

```
db.posts.updateOne( { title: "Post Title 1" }, { $set: { likes: 2 } } )
```

Check the document again and you'll see that the "like" have been updated.

Example

```
db.posts.find( { title: "Post Title 1" } )
```

Insert if not found

If you would like to insert the document if it is not found, you can use the `upsert` option.

Example

Update the document, but if not found insert it:

```
db.posts.updateOne(
  { title: "Post Title 5" },
  {
    $set:
      {
        title: "Post Title 5",
        body: "Body of post.",
        category: "Event",
        likes: 5,
        tags: ["news", "events"],
        date: Date()
      }
```

```
  },
  { upsert: true }
)
```

updateMany()

The `updateMany()` method will update all documents that match the provided query.

Example

Update `likes` on all documents by 1. For this we will use the `$inc` (increment) operator:

```
db.posts.updateMany({}, { $inc: { likes: 1 } })
```

MongoDB mongosh Delete

Delete Documents

We can delete documents by using the methods `deleteOne()` or `deleteMany()`.

These methods accept a query object. The matching documents will be deleted.

deleteOne()

The `deleteOne()` method will delete the first document that matches the query provided.

Example

```
db.posts.deleteOne({ title: "Post Title 5" })
```

deleteMany()

The `deleteMany()` method will delete all documents that match the query provided.

Example

```
db.posts.deleteMany({ category: "Technology" })
```

The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

Basic syntax of **drop()** command is as follows –

```
db.COLLECTION_NAME.drop()
```

Example

First, check the available collections into your database **mydb**.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

Now drop the collection with the name **mycollection**.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
tutorialspoint
>
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>,.....<valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all

MongoDB Query Operators

MongoDB Query Operators

There are many query operators that can be used to compare and reference document fields.

Comparison

The following operators can be used in queries to compare values:

- `$eq`: Values are equal
- `$ne`: Values are not equal
- `$gt`: Value is greater than another value
- `$gte`: Value is greater than or equal to another value
- `$lt`: Value is less than another value
- `$lte`: Value is less than or equal to another value
- `$in`: Value is matched within an array

Logical

The following operators can logically compare multiple queries.

- `$and`: Returns documents where both queries match
- `$or`: Returns documents where either query matches
- `$nor`: Returns documents where both queries fail to match
- `$not`: Returns documents where the query does not match

Evaluation

The following operators assist in evaluating documents.

- `$regex`: Allows the use of regular expressions when evaluating field values
- `$text`: Performs a text search
- `$where`: Uses a JavaScript expression to match documents

MongoDB Update Operators

MongoDB Update Operators

There are many update operators that can be used during document updates.

Fields

The following operators can be used to update fields:

- `$currentDate`: Sets the field value to the current date
- `$inc`: Increments the field value
- `$rename`: Renames the field
- `$set`: Sets the value of a field
- `$unset`: Removes the field from the document

Array

The following operators assist with updating arrays.

- `$addToSet`: Adds distinct elements to an array
- `$pop`: Removes the first or last element of an array
- `$pull`: Removes all elements from an array that match the query
- `$push`: Adds an element to an array

MongoDB Aggregation Pipelines

Aggregation Pipelines

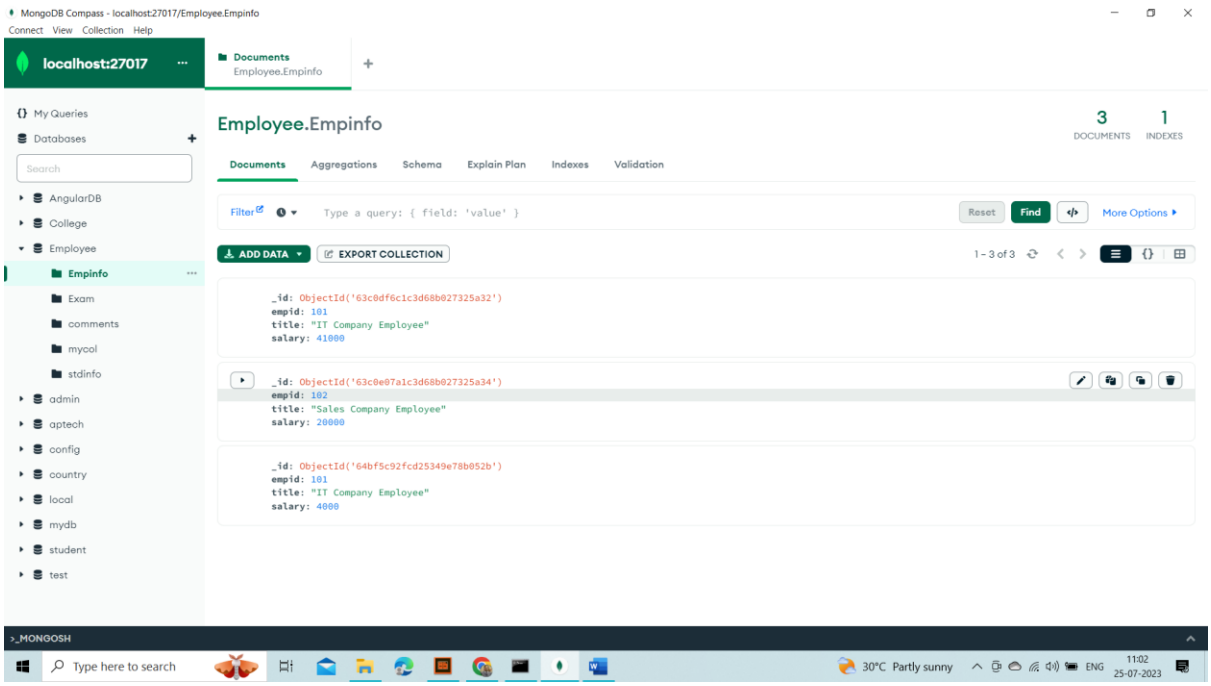
- Aggregation operations allow you to group, sort, perform calculations, analyze data, and much more.
- Aggregation pipelines can have one or more "stages". The order of these stages are important. Each stage acts upon the results of the previous stage.

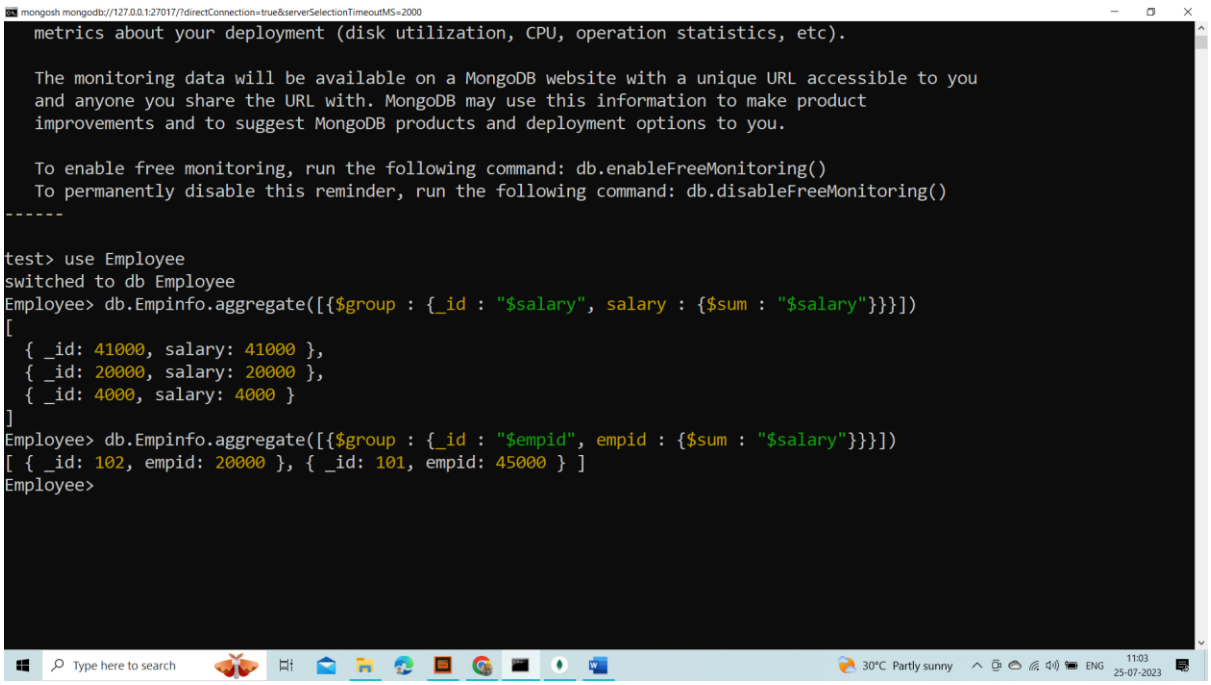
Example

```
db.posts.aggregate([
  // Stage 1: Only find documents that have more than 1 like
  {
    $match: { likes: { $gt: 1 } }
  },
  // Stage 2: Group documents by category and sum each categories likes
  {
    $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
  }
])
```

```
[ { _id: 'News', totalLikes: 3 }, { _id: 'Event', totalLikes: 8 } ]
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

----##

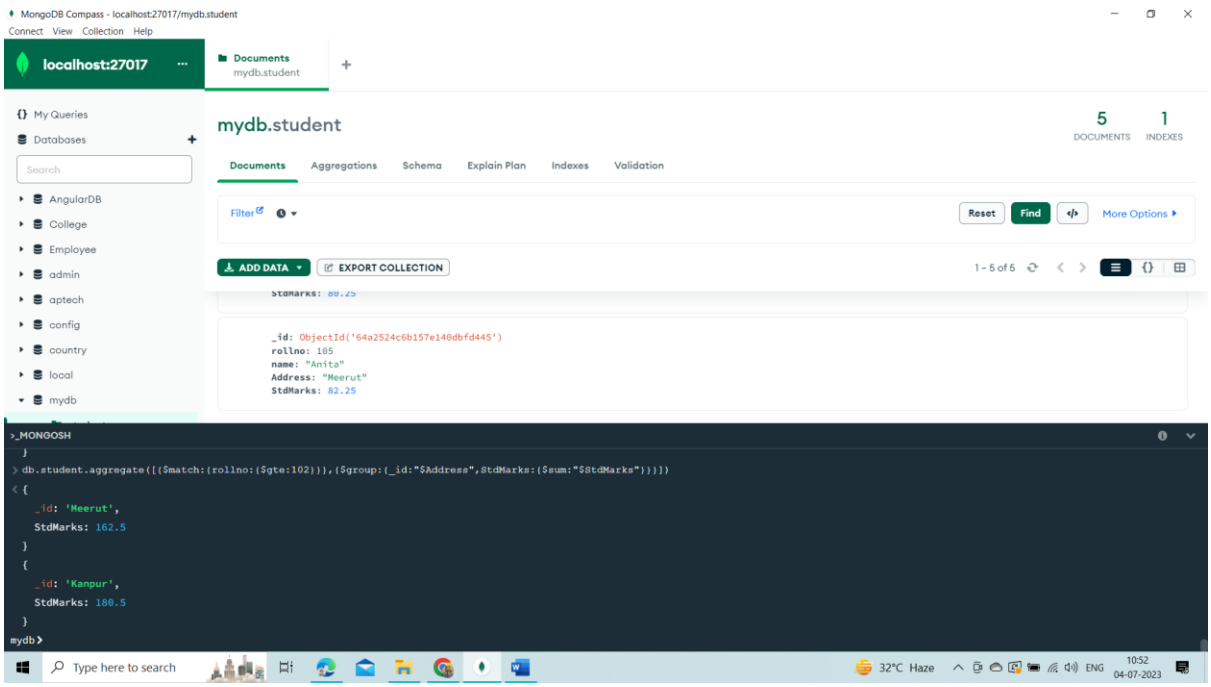




db.Empinfo.aggregate([{$group : {_id : "$empid", empid : {$sum : "$salary"}}}])

[{ _id: 102, empid: 20000 }, { _id: 101, empid: 45000 }]

##



Sorting with Aggregate

```
db.student.aggregate([{$sort:{rollno:-1}}])
```

Sorting with Sort Function

```
db.student.find({}).sort({rollno:-1})
```

- # Sample Data

- To demonstrate the use of stages in a aggregation pipeline, we will load sample data into our database.
- From the MongoDB Atlas dashboard, go to Databases. Click the ellipsis and select "Load Sample Dataset". This will load several sample datasets into your database.
- In the next sections we will explore several aggregation pipeline stages in more detail using this sample data.

MongoDB Aggregation $group

Aggregation $group

This aggregation stage groups documents by the unique `_id` expression provided.

Don't confuse this `_id` expression with the `_id` ObjectId provided to each document.

Example

In this example, we are using the "sample_airbnb" database loaded from our sample data in the Intro to Aggregations section.

```
db.listingsAndReviews.aggregate(

    [ { $group : { _id : "$property_type" } } ]

)
```

```
[
  { _id: 'Aparthotel' },
  { _id: 'Apartment' },
  { _id: 'Barn' },
  { _id: 'Bed and breakfast' },
  { _id: 'Boat' },
  { _id: 'Boutique hotel' },
  { _id: 'Bungalow' },
  { _id: 'Cabin' },
  { _id: 'Camper/RV' },
  { _id: 'Campsite' },
  { _id: 'Casa particular (Cuba)' },
  { _id: 'Castle' },
  { _id: 'Chalet' },
  { _id: 'Condominium' },
  { _id: 'Cottage' },
  { _id: 'Earth house' },
  { _id: 'Farm stay' },
  { _id: 'Guest suite' },
  { _id: 'Guesthouse' },
  { _id: 'Heritage hotel (India)' }
]
Atlas atlas-8iy36m-shard-0 [primary] sample_airbnb>
```

This will return the distinct values from the `property_type` field.

MongoDB Aggregation $limit

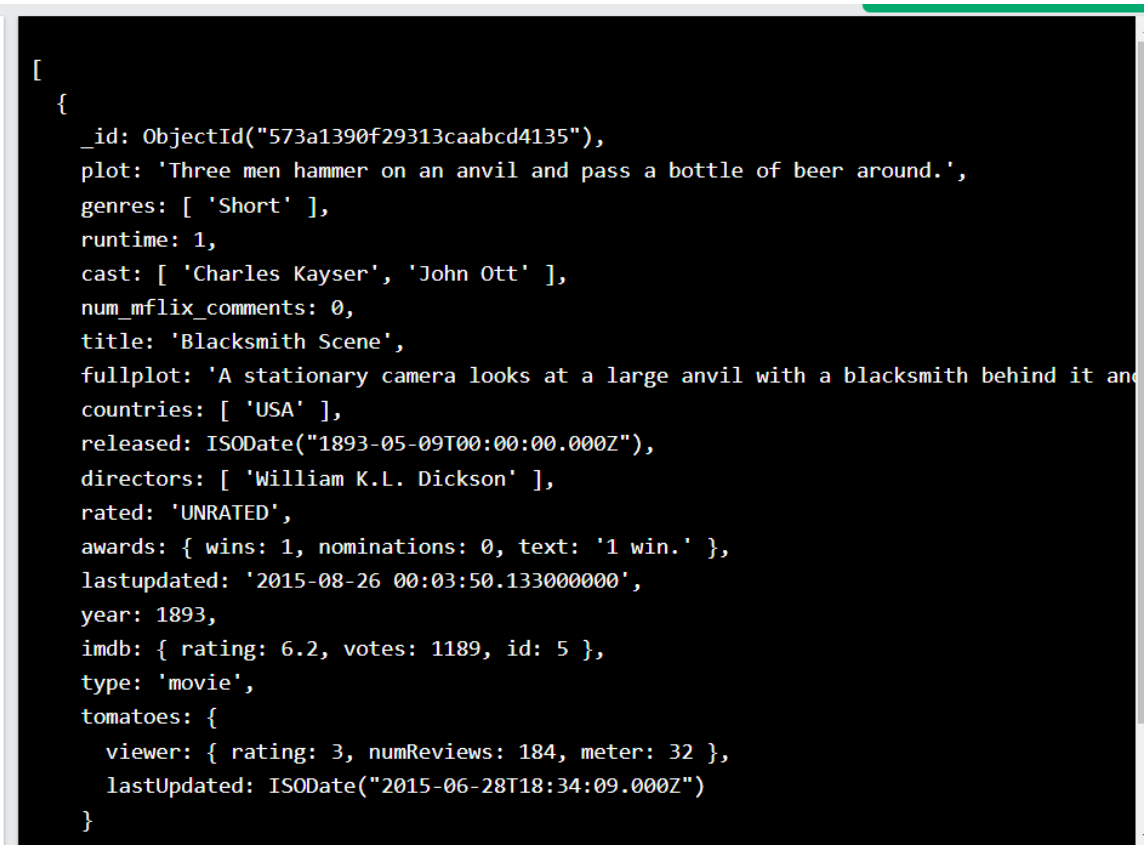
Aggregation $limit

This aggregation stage limits the number of documents passed to the next stage.

Example

In this example, we are using the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

```
db.movies.aggregate([ { $limit: 1 } ])
```



```
[
  {
    _id: ObjectId("573a1390f29313caabcd4135"),
    plot: 'Three men hammer on an anvil and pass a bottle of beer around.',
    genres: [ 'Short' ],
    runtime: 1,
    cast: [ 'Charles Kayser', 'John Ott' ],
    num_mflix_comments: 0,
    title: 'Blacksmith Scene',
    fullplot: 'A stationary camera looks at a large anvil with a blacksmith behind it an
    countries: [ 'USA' ],
    released: ISODate("1893-05-09T00:00:00.000Z"),
    directors: [ 'William K.L. Dickson' ],
    rated: 'UNRATED',
    awards: { wins: 1, nominations: 0, text: '1 win.' },
    lastupdated: '2015-08-26 00:03:50.133000000',
    year: 1893,
    imdb: { rating: 6.2, votes: 1189, id: 5 },
    type: 'movie',
    tomatoes: {
      viewer: { rating: 3, numReviews: 184, meter: 32 },
      lastUpdated: ISODate("2015-06-28T18:34:09.000Z")
    }
```

MongoDB
Aggregation $project

Aggregation $project

This aggregation stage passes only the specified fields along to the next aggregation stage.

This is the same projection that is used with the `find()` method.

Example

In this example, we are using the "sample_restaurants" database loaded from our sample data in the Intro to Aggregations section.

```
db.restaurants.aggregate([
  {
    $project: {
      "name": 1,
      "cuisine": 1,
      "address": 1
    }
  },
  {
    $limit: 5
  }
])
```

```
[
    {
      _id: ObjectId("5eb3d668b31de5d588f4292a"),
      address: {
        building: '2780',
        coord: [ -73.98241999999999, 40.579505 ],
        street: 'Stillwell Avenue',
        zipcode: '11224'
      },
      cuisine: 'American',
      name: 'Riviera Caterer'
    },
    {
      _id: ObjectId("5eb3d668b31de5d588f4292b"),
      address: {
        building: '7114',
        coord: [ -73.9068506, 40.6199034 ],
        street: 'Avenue U',
        zipcode: '11234'
      },
      cuisine: 'Delicatessen',
      name: "Wilken'S Fine Food"
    },
    {
```

This will return the documents but only include the specified fields.

Notice that the `_id` field is also included. This field is always included unless specifically excluded.

We use a `1` to include a field and `0` to exclude a field.

Note: You cannot use both 0 and 1 in the same object. The only exception is the `_id` field. You should either specify the fields you would like to include or the fields you would like to exclude.

MongoDB Aggregation `$sort`

Aggregation `$sort`

This aggregation stage groups sorts all documents in the specified sort order.

Remember that the order of your stages matters. Each stage only acts upon the documents that previous stages provide.

Example

In this example, we are using the "sample_airbnb" database loaded from our sample data in the Intro to Aggregations section.

```
db.listingsAndReviews.aggregate([
  {
    $sort: { "accommodates": -1 }
  },
  {
    $project: {
      "name": 1,
      "accommodates": 1
    }
  },
  {
    $limit: 5
  }
])
```

```
[
  {
    _id: '19990097',
    name: 'House close to station & direct to opera house....',
    accommodates: 16
  },
  { _id: '19587001', name: 'Kaena O Kekai', accommodates: 16 },
  {
    _id: '20958766',
    name: 'Great Complex of the Cellars',
    accommodates: 16
  },
  {
    _id: '12509339',
    name: 'Barra da Tijuca beach house',
    accommodates: 16
  },
  {
    _id: '20455499',
    name: 'DOWNTOWN VIP MONTREAL ,HIGH END DECOR,GOOD VALUE..',
    accommodates: 16
  }
]
Atlas atlas-8iy36m-shard-0 [primary] sample_airbnb>
```

This will return the documents sorted in descending order by the `accommodates` field.

The sort order can be chosen by using `1` or `-1`. `1` is ascending and `-1` is descending.

MongoDB Aggregation `$match`

Aggregation `$match`

This aggregation stage behaves like a find. It will filter documents that match the query provided.

Using `$match` early in the pipeline can improve performance since it limits the number of documents the next stages must process.

Example

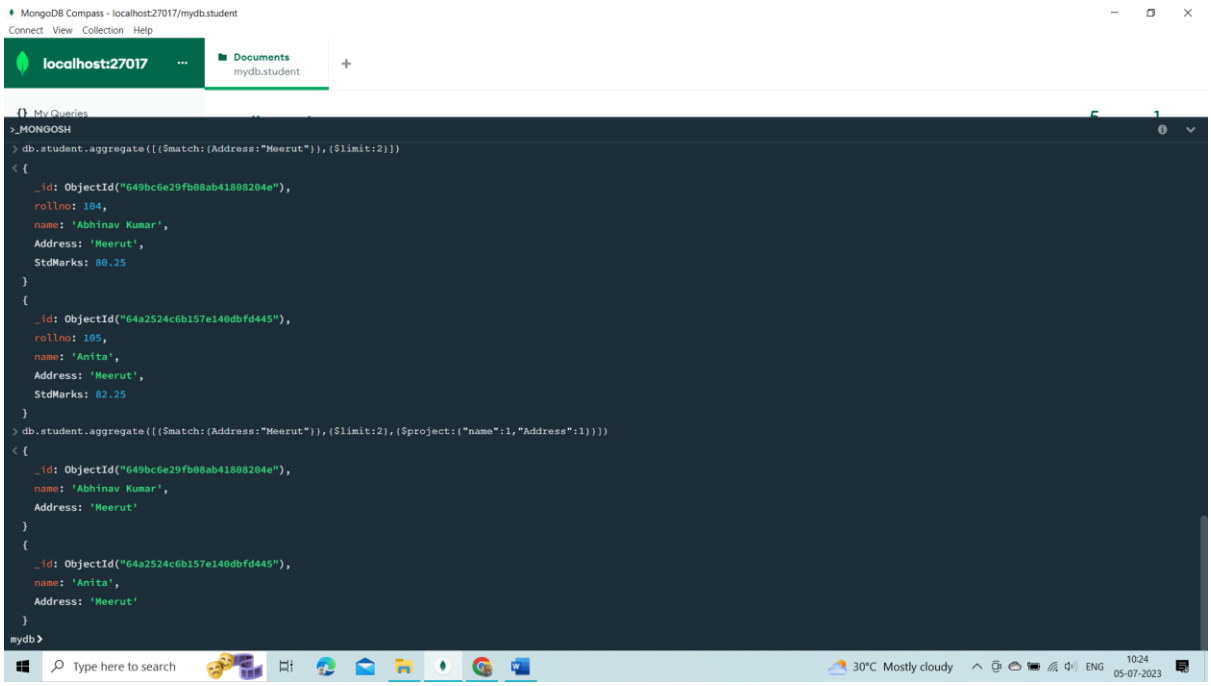
In this example, we are using the "sample_airbnb" database loaded from our sample data in the Intro to Aggregations section.

```
db.listingsAndReviews.aggregate([

  { $match : { property_type : "House" } },

  { $limit: 2 },
```

```
  { $project: {
    "name": 1,
    "bedrooms": 1,
    "price": 1
  }}
])
```

```
[
  {
    _id: '16253247',
    name: 'Gorgeous Remodeled Modern Home w/ Beach Across St.',
    bedrooms: 2,
    price: 194.00
  },
  {
    _id: '18616850',
    name: 'The Paddington Cottage | Sydney Eastern Suburbs',
    bedrooms: 3,
    price: 350.00
  }
]
Atlas atlas-8iy36m-shard-0 [primary] sample_airbnb>
```

This will only return documents that have the `property_type` of "House".



MongoDB Aggregation $addFields

Aggregation $addFields

This aggregation stage adds new fields to documents.

Example

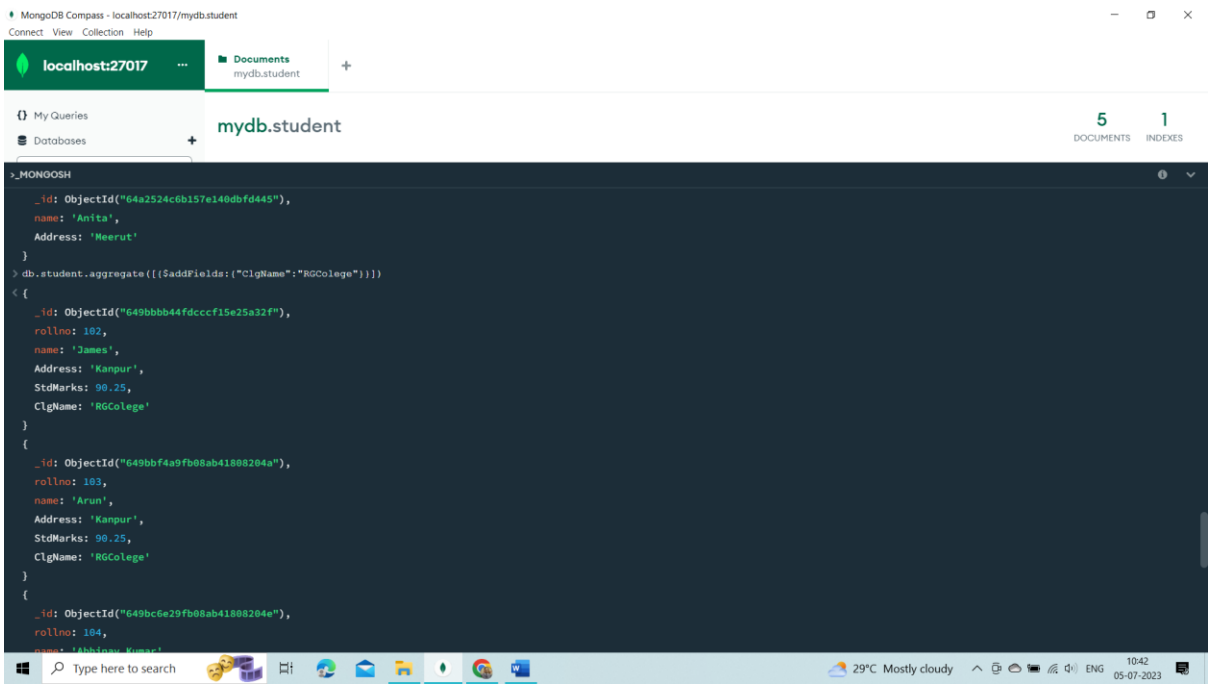
In this example, we are using the "sample_restaurants" database loaded from our sample data in the Intro to Aggregations section.

```
db.restaurants.aggregate([
  {
    $addFields: {
      avgGrade: { $avg: "$grades.score" }
    }
  },
  {
    $project: {
      "name": 1,
      "avgGrade": 1
    }
  },
```

```
  {
    $limit: 5
  }
])
```

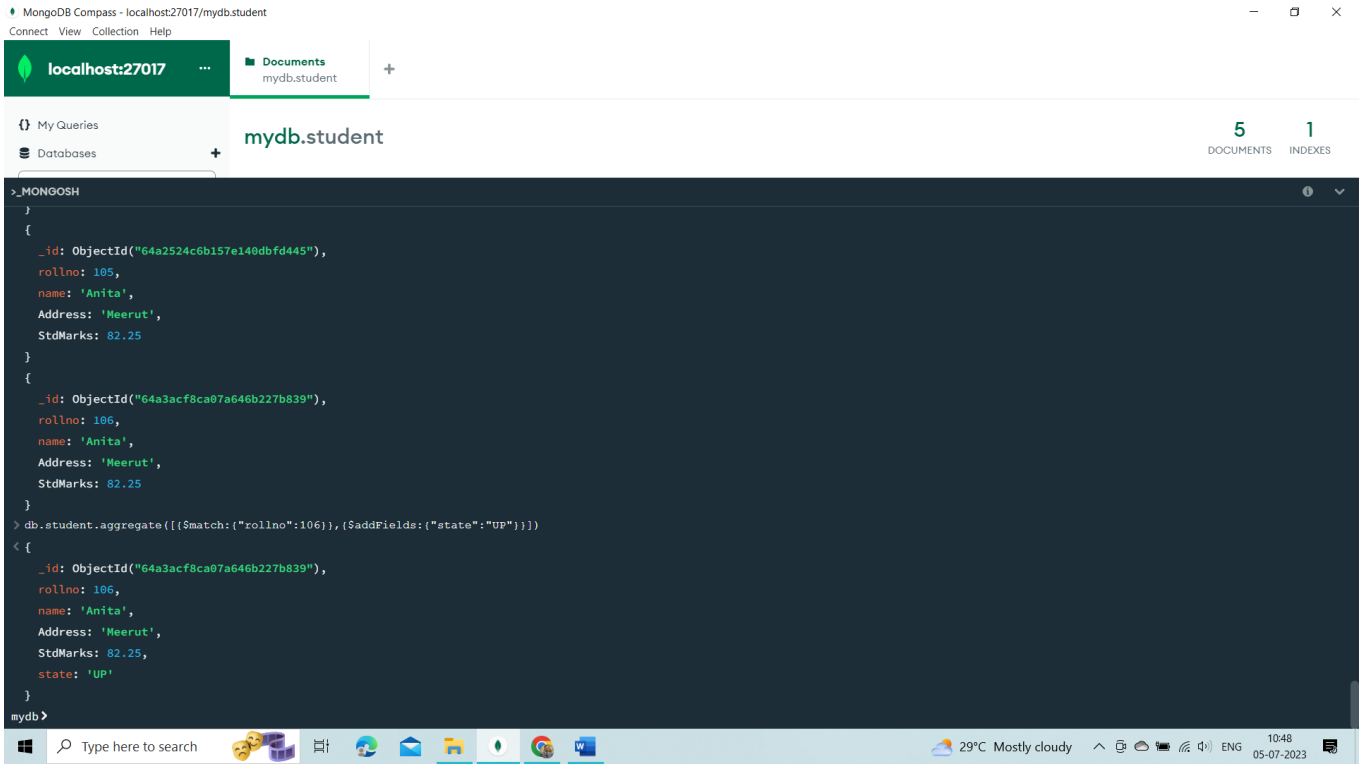
```
[
  {
    _id: ObjectId("5eb3d668b31de5d588f4292a"),
    name: 'Riviera Caterer',
    avgGrade: 9
  },
  {
    _id: ObjectId("5eb3d668b31de5d588f4292b"),
    name: "Wilken'S Fine Food",
    avgGrade: 10
  },
  {
    _id: ObjectId("5eb3d668b31de5d588f4292c"),
    name: 'Kosher Island',
    avgGrade: 10.5
  },
  {
    _id: ObjectId("5eb3d668b31de5d588f4292d"),
    name: "Wendy'S",
    avgGrade: 13.75
  },
  {
    _id: ObjectId("5eb3d668b31de5d588f4292e"),
    name: 'Morris Park Bake Shop',
```

Add Fields to all record



This will return the documents along with a new field, `avgGrade`, which will contain the average of each restaurants `grades.score`.

How to add Field to select record



MongoDB Aggregation $count

Aggregation $count

This aggregation stage counts the total amount of documents passed from the previous stage.

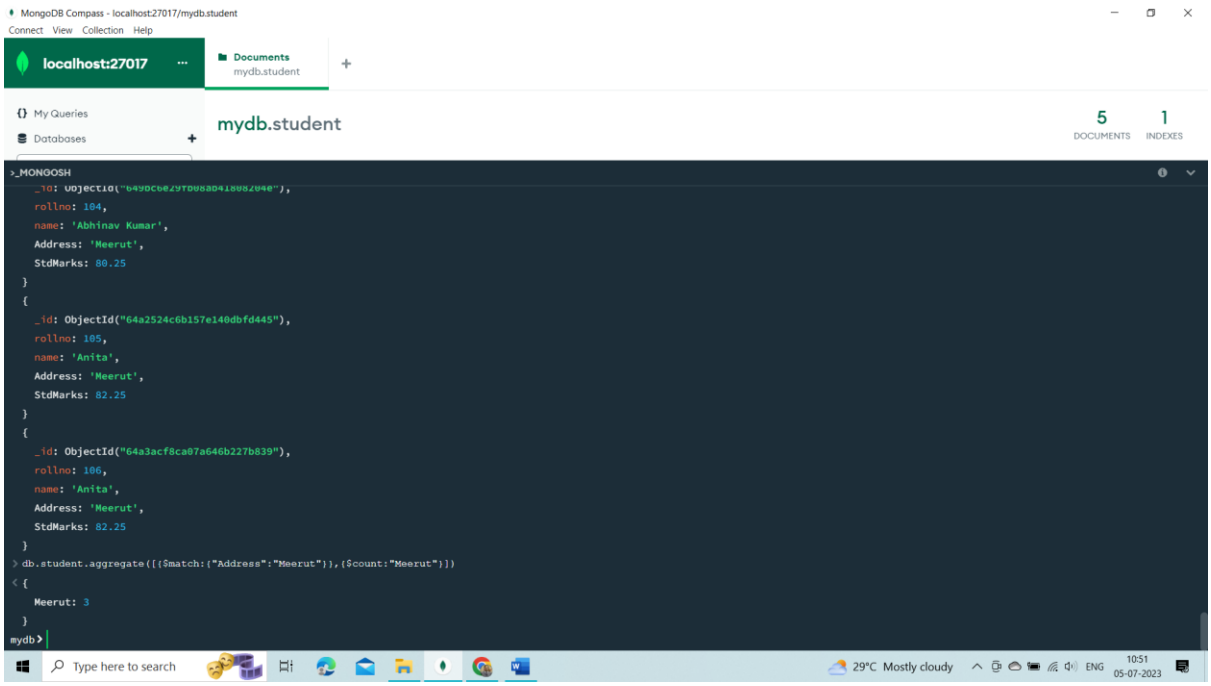
Example

In this example, we are using the "sample_restaurants" database loaded from our sample data in the Intro to Aggregations section.

```
db.restaurants.aggregate([
  {
    $match: { "cuisine": "Chinese" }
  },
  {
    $count: "totalChinese"
  }
])
```

```
[ { totalChinese: 2418 } ]
Atlas atlas-8iy36m-shard-0 [primary] sample_restaurants>
```

Counting



This will return the number of documents at the `$count` stage as a field called "totalChinese".

```
nployee> db.getCollection('Empinfo').find({})

{
  _id: ObjectId("63c0df6c1c3d68b027325a32"),
  empid: 101,
  title: 'IT Company Employee',
  salary: 41000
},
{
  _id: ObjectId("63c0e07a1c3d68b027325a34"),
  empid: 102,
  title: 'Sales Company Employee',
  salary: 20000
}

nployee> db.getCollection('Empinfo').find({},{empid:1,salary:1})

{
  _id: ObjectId("63c0df6c1c3d68b027325a32"),
  empid: 101,
  salary: 41000
},
{
  _id: ObjectId("63c0e07a1c3d68b027325a34"),
  empid: 102,
nployee>
```

$filter Aggregation

Json Data (name - collection)

[

{ "_id": 1,

"collapsed": [10, 20, 30, 40]

},

{ "_id": 2,

"collapsed": [11, 22, 33, 44] }

]

Query

db.collection.aggregate([

{

"$project": {

"evenArray": {

"$filter": {

```
        "input": "$collapsed",
        "as": "colsp",
        "cond": {
          "$eq": [ "$_id",  2 ]
        }    } }    }
  }
])
```

Output

```
[ {
   "_id": 1,
   "evenArray": []
 },
 {   "_id": 2,
   "evenArray": [     11,     22,     33,     44    ]
 }
]
```

Aggregation `$lookup`

This aggregation stage performs a left outer join to a collection in the same database.

There are four required fields:

- `from`: The collection to use for lookup in the same database
- `localField`: The field in the primary collection that can be used as a unique identifier in the `from` collection.
- `foreignField`: The field in the `from` collection that can be used as a unique identifier in the primary collection.

- `as`: The name of the new field that will contain the matching documents from the `from` collection.

Lookup complete Exercise use for join

1) Create Student Collections

```
db.students .insertMany([
{"id" : 1 , "pupil" : "John" , "std" : 6, "ht" : 153 , "wt" : 43},
{"id" : 2 , "pupil" : "Jack" , "std" : 6, "ht" : 164 , "wt" : 54},
])
```

2) Create sport Collections

```
db.sports .insertMany([
{"id" : 1, "sport" : "Basketball" , "winner" : "John"},
{"id" : 2, "sport" : "TT", "winner" : "Jack"},
{"id" : 3, "sport" : "Tennis" , "winner" : "John"},
])
```

The following query joins matches from sports collection to students collection based on the field value pupil and winner in students and sports respectively

Using Lookup

```
db.students.aggregate([
{
$lookup:
{
from : "sports",
localField : "pupil",
foreignField : "winner",
as : "games"
} } ] )
```

Result

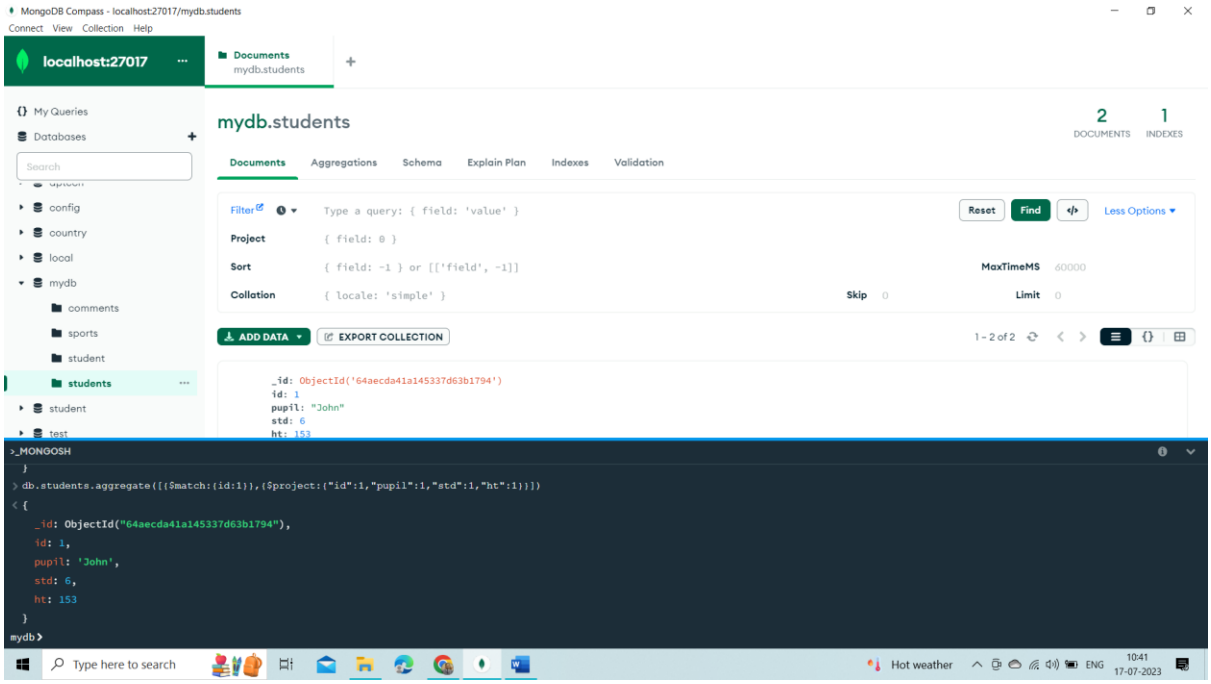
```
{
_id: ObjectId("64aecda41a145337d63b1794"),
id: 1,
pupil: 'John',
std: 6,
ht: 153,
```

```
wt: 43,
games: [
{
_id: ObjectId("64aece5d1a145337d63b1796"),
id: 1,
sport: 'Basketball',
winner: 'John'
},
{
_id: ObjectId("64aece5d1a145337d63b1798"),
id: 3,
sport: 'Tennis',
winner: 'John'
}
]
}
{
_id: ObjectId("64aecda41a145337d63b1795"),
id: 2,
pupil: 'Jack',
std: 6,
ht: 164,
wt: 54,
games: [
{
_id: ObjectId("64aece5d1a145337d63b1797"),
id: 2,
sport: 'TT',
winner: 'Jack'
}
]
}
```

Using Limit

```
db.students.aggregate([
{
$lookup:
{
from : "sports",
localField : "pupil",
foreignField : "winner",
as : "games"
} },
{ $limit: 1

 }])
```

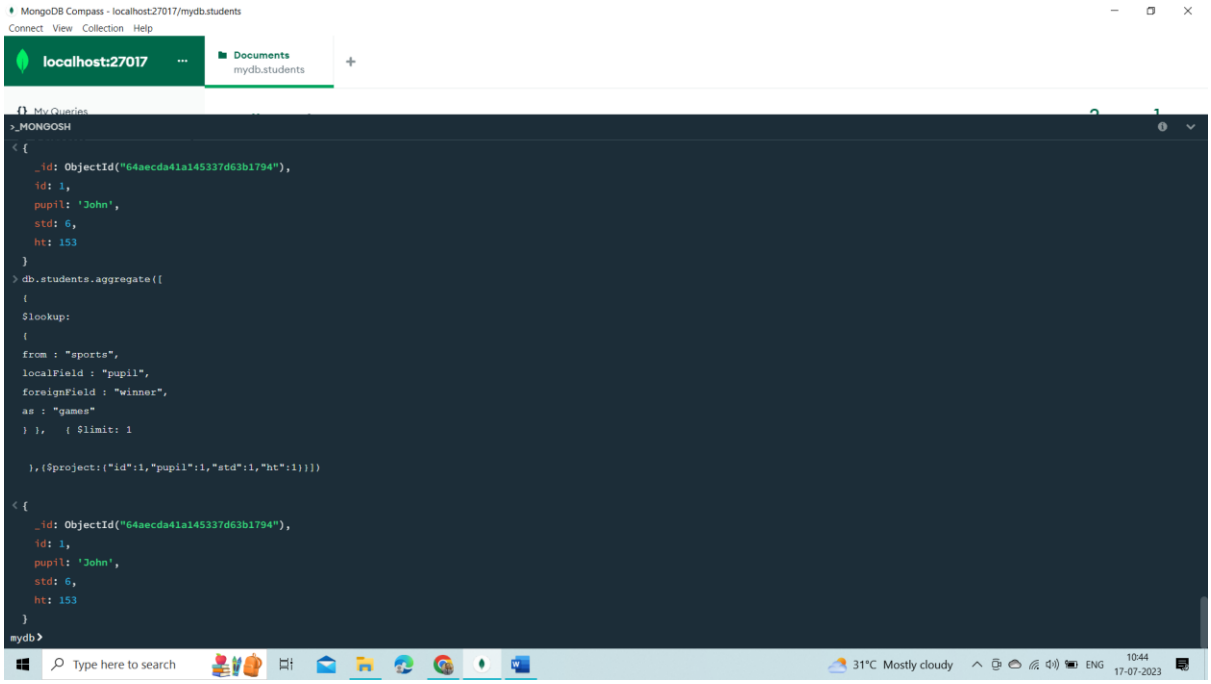
Match with project



Lookup with Limit and Project

```
db.students.aggregate([
{
$lookup:
{
from : "sports",
localField : "pupil",
foreignField : "winner",
as : "games"
} },
 {
   $limit: 1
 },
{
$project:
        {"id":1,
        "pupil":1,
        "std":1,
        "ht":1
         }
}])
```

Output



Second Exercise

Create two sample collections, `inventory` and `orders`:

```
db.inventory.insertMany( [
{ prodId: 100, price: 20, quantity: 125 },
{ prodId: 101, price: 10, quantity: 234 },
{ prodId: 102, price: 15, quantity: 432 },
{ prodId: 103, price: 17, quantity: 320 }
] )
```

```
db.orders.insertMany( [
{ orderId: 201, custid: 301, prodId: 100, numPurchased: 20 },
{ orderId: 202, custid: 302, prodId: 101, numPurchased: 10 },
{ orderId: 203, custid: 303, prodId: 102, numPurchased: 5 },
{ orderId: 204, custid: 303, prodId: 103, numPurchased: 15 },
{ orderId: 205, custid: 303, prodId: 103, numPurchased: 20 },
{ orderId: 206, custid: 302, prodId: 102, numPurchased: 1 },
{ orderId: 207, custid: 302, prodId: 101, numPurchased: 5 },
{ orderId: 208, custid: 301, prodId: 100, numPurchased: 10 },
{ orderId: 209, custid: 303, prodId: 103, numPurchased: 30 }
] )
```

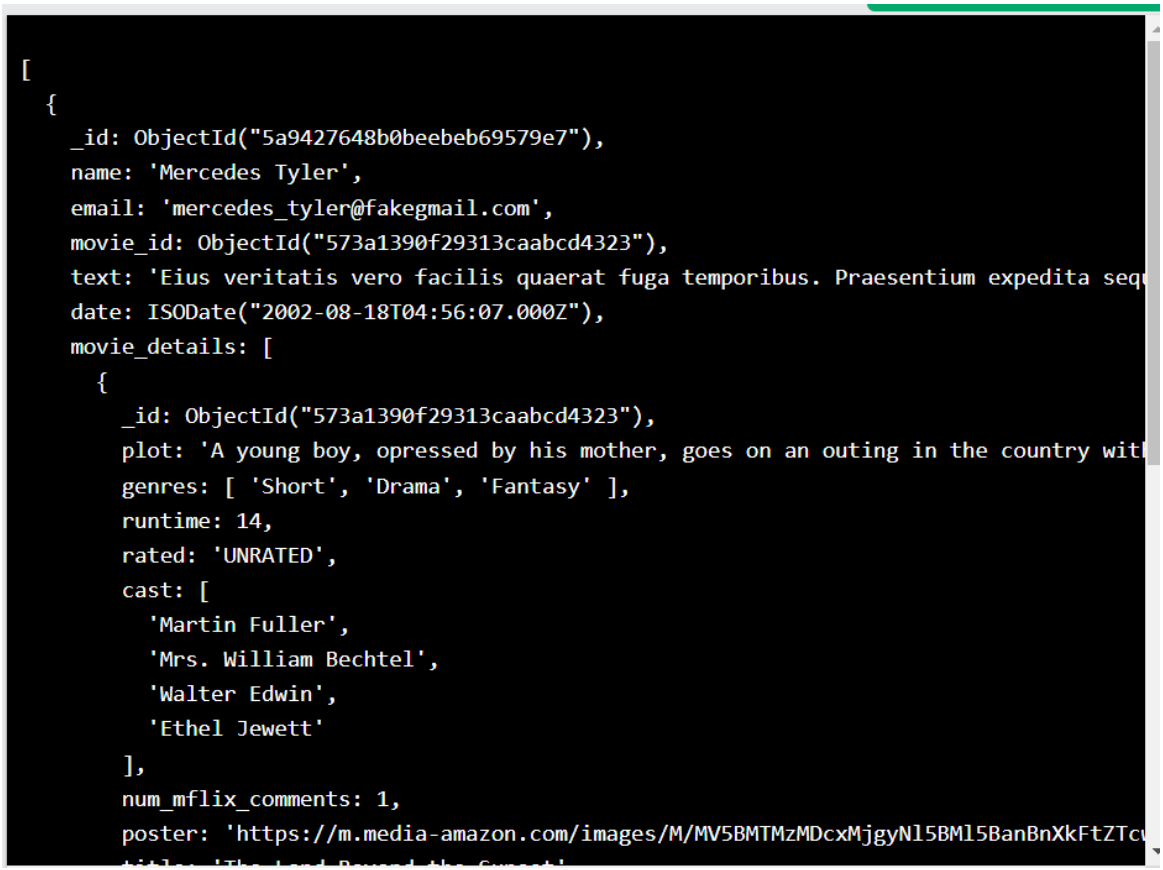
```
db.createView( "sales", "orders", [
    {
        $lookup:
            {
                from: "inventory",
                localField: "prodId",
```

```
                foreignField: "prodId",
                as: "inventoryDocs"
            }
    },
    {
        $project:
            {
              _id: 0,
              prodId: 1,
              orderId: 1,
              numPurchased: 1,
              price: "$inventoryDocs.price"
            }
    },
        { $unwind: "$price" }
] )
```

Example

In this example, we are using the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

```
db.comments.aggregate([
  {
    $lookup: {
      from: "movies",
      localField: "movie_id",
      foreignField: "_id",
      as: "movie_details",
    },
  },
  {
    $limit: 1
  }
])
```



This will return the movie data along with each comment.

This is the full set of aggregation stages:

- `$match` – Filter documents
- `$geoNear` – Sort documents based on geographic proximity
- `$project` – Reshape documents (remove or rename keys or add new data based on calculations on the existing data)
- `$lookup` – Coming in 3.2 – Left-outer joins
- `$unwind` – Expand documents (for example create multiple documents where each contains one element from an array from the original document)
- `$group` – Summarize documents
- `$sample` – Randomly selects a subset of documents
- `$sort` – Order documents

- `$skip` – Jump over a number of documents
- `$limit` – Limit number of documents
- `$redact` – Restrict sensitive content from documents
- `$out` – *Coming in 3.2** – store the results in a new collection

MongoDB Aggregation $out

Aggregation $out

This aggregation stage writes the returned documents from the aggregation pipeline to a collection.

The $out stage must be the last stage of the aggregation pipeline.

Example

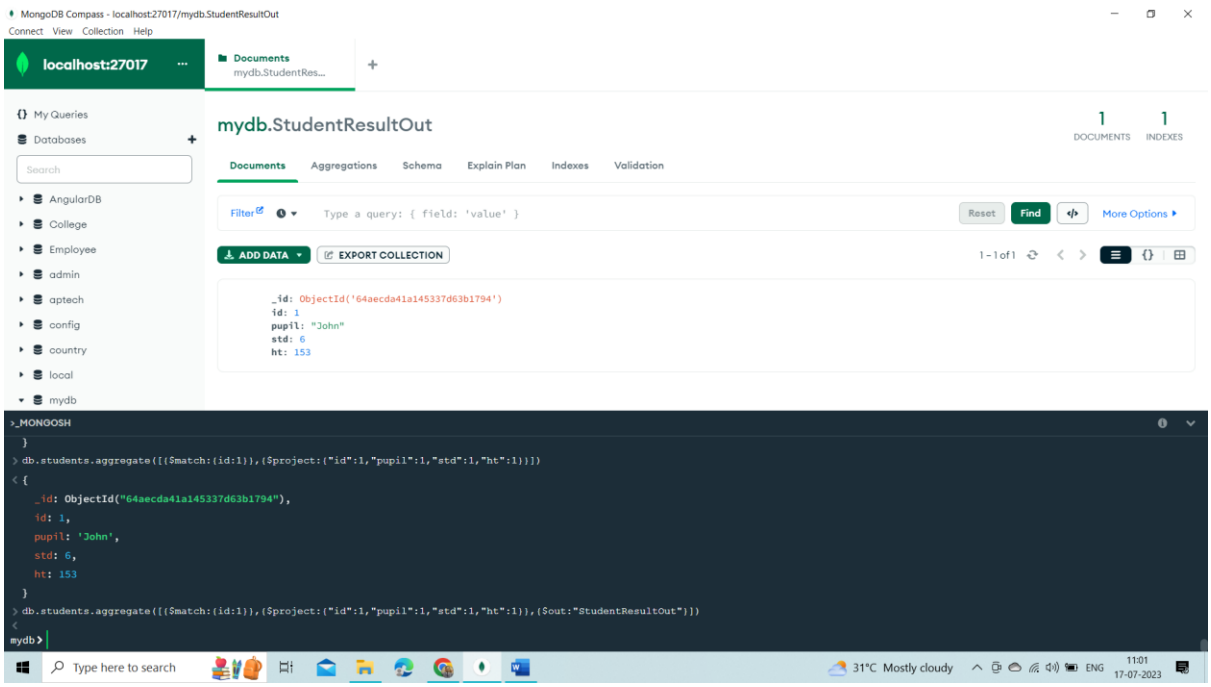
In this example, we are using the "sample_airbnb" database loaded from our sample data in the Intro to Aggregations section.

```
db.listingsAndReviews.aggregate([
  {
    $group: {
      _id: "$property_type",
      properties: {
        $push: {
          name: "$name",
          accommodates: "$accommodates",
          price: "$price",
        },
      },
    },
  },
  { $out: "properties_by_type" },
])
```

```
Results saved to collection: properties_by_type
```

Save Result into out

```
db.students.aggregate([
{
$match:{id:1}
},
{
$project:{"id":1,"pupil":1,"std":1,"ht":1}
},
{
$out:"StudentResultOut"}
])
```



The first stage will group properties by the `property_type` and include the `name`, `accommodates`, and `price` fields for each. The `$out` stage will create a new collection called `properties_by_type` in the current database and write the resulting documents into that collection.

Indexing & Search

```
> db.content.find().pretty()
{
        "_id" : ObjectId("603622eef19652db63812eb5"),
        "name" : "Rohit",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb6"),
        "name" : "Priya",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb7"),
        "name" : "Suman",
        "line" : "I dont like dogs and cats but i like cow"
}
>
```

Step 2: Create index:

Now we create a string index on the name and pet field with the help of the createIndex() method. So we can search text over the name and line fields:

```
db.content.createIndex({name:"text",line:"text"})
```

```
> db.content.createIndex({name:"text",line:"text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Step 3: Search Text:

Now we are ready to search text. For example, we are going to search all the document which contain text love.

```
db.content.find({$text:{$search:"love"}})
```

Indexing & Search

MongoDB Atlas comes with a full-text search engine that can be used to search for documents in a collection.

Atlas Search is powered by Apache Lucene.

Creating an Index

We'll use the Atlas dashboard to create an index on the "sample_mflix" database from the sample data that we loaded in the Intro to Aggregations section.

1. From the Atlas dashboard, click on your **Cluster name** then the **Search** tab.
2. Click on the **Create Search Index** button.
3. Use the **Visual Editor** and click Next.
4. Name your index, choose the Database and Collection you want to index and click Next.
    - If you name your index "default" you will not have to specify the index name in the `$search` pipeline stage.
    - Choose the `sample_mflix` database and the `movies` collection.
5. Click **Create Search Index** and wait for the index to complete.

Running a Query

To use our search index, we will use the `$search` operator in our aggregation pipeline.

Example

```
db.movies.aggregate([
  {
    $search: {
      index: "default", // optional unless you named your index something other than "default"
      text: {
        query: "star wars",
        path: "title"
      },
    },
  },
  {
    $project: {
      title: 1,
      year: 1,
    }
  }
```

```
])
```

```
[
  {
    _id: ObjectId("573a13c0f29313caabd62f62"),
    year: 2008,
    title: 'Star Wars: The Clone Wars'
  },
  {
    _id: ObjectId("573a13bdf29313caabd599ca"),
    title: 'Robot Chicken: Star Wars',
    year: 2007
  },
  {
    _id: ObjectId("573a1396f29313caabce37ff"),
    title: 'Star!',
    year: 1968
  },
  {
    _id: ObjectId("573a13a6f29313caabd17d08"),
    title: 'Star',
    year: 2001
  },
  {
    _id: ObjectId("573a1397f29313caabce68f6"),
    year: 1977,
```

The first stage of this aggregation pipeline will return all documents in the `movies` collection that contain the word "star" or "wars" in the `title` field.

The second stage will project the `title` and `year` fields from each document.

MongoDB Schema Validation

Schema Validation

By default MongoDB has a flexible schema. This means that there is no strict schema validation set up initially.

Schema validation rules can be created in order to ensure that all documents a collection share a similar structure.

Schema Validation

MongoDB supports JSON Schema validation. The `$jsonSchema` operator allows us to define our document structure.

Example

```
db.createCollection("posts", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "title", "body" ],
      properties: {
        title: {
          bsonType: "string",
          description: "Title of post - Required."
        },
        body: {
          bsonType: "string",
          description: "Body of post - Required."
        },
        category: {
          bsonType: "string",
          description: "Category of post - Optional."
        },
        likes: {
          bsonType: "int",
          description: "Post like count. Must be an integer -
Optional."
        },
        tags: {
          bsonType: ["string"],
          description: "Must be an array of strings - Optional."
        },
        date: {
          bsonType: "date",
          description: "Must be a date - Optional."
        }
      }
    }
  }
```

```
{ ok: 1 }
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

This will create the `posts` collection in the current database and specify the JSON Schema validation requirements for the collection.

Employee.mycol

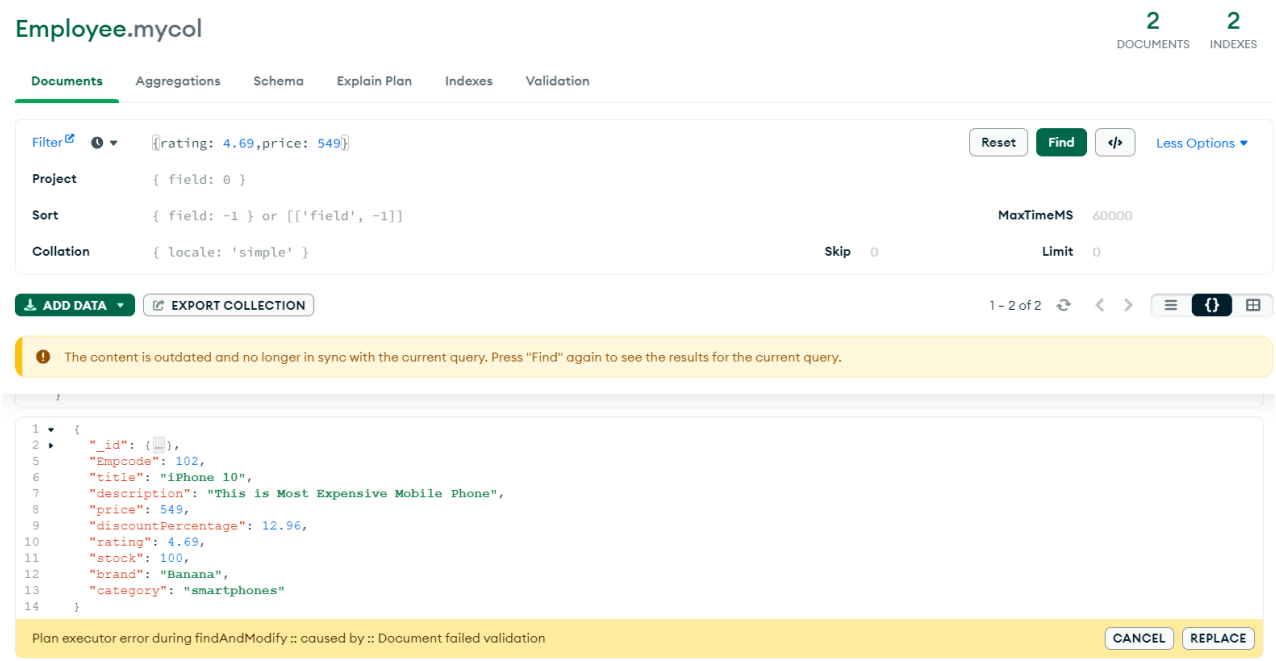
Documents | Aggregations | Schema | Explain Plan | Indexes | **Validation**

Validation Action: Error | Validation Level: Strict

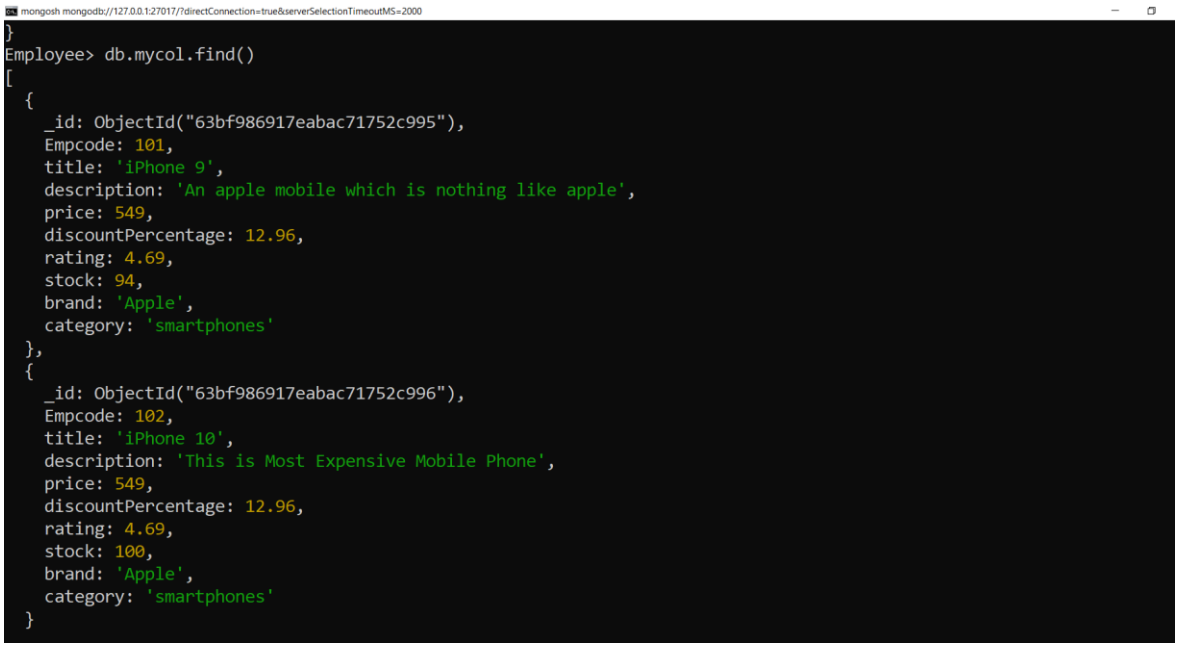
```
{
  $jsonSchema: {
    bsonType: 'object',
    properties: {
      brand: {
        bsonType: 'string',
        'enum': [
          'Apple',
          'mango'
        ]
      }
    }
  }
}
```

Sample document that passed validation

```
_id: ObjectId('63bf986917eabac71752c995')
Empcode: 101
title: "iPhone 9"
description: "An apple mobile which is nothing like apple"
price: 549
discountPercentage: 12.96
rating: 4.69
stock: 94
brand: "Apple"
category: "smartphones"
```



Document failed validation



```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
]
Employee> db.mycol.insert({Empcode:103,brand:'Banana'});
Uncaught:
MongoBulkWriteError: Document failed validation
Result: BulkWriteResult {
  result: {
    ok: 1,
    writeErrors: [
      WriteError {
        err: {
          index: 0,
          code: 121,
          errmsg: 'Document failed validation',
          errInfo: {
            failingDocumentId: ObjectId("63c1929f114972ece2ab5c3c"),
            details: {
              operatorName: '$jsonSchema',
              schemaRulesNotSatisfied: [Array]
            }
          },
          op: {
            Empcode: 103,
            brand: 'Banana',
            _id: ObjectId("63c1929f114972ece2ab5c3c")
          }
        }
      }
```

If we insert brand:'mango' then successfully inserted into table

```
  }
}
Employee> db.mycol.insert({Empcode:103,brand:'mango'});
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("63c1931b114972ece2ab5c3d") }
}
Employee>
```

MongoDB Data API

MongoDB Data API

The MongoDB Data API can be used to query and update data in a MongoDB database without the need for language specific drivers.

Language drivers should be used when possible, but the MongoDB Data API comes in handy when drivers are not available or drivers are overkill for the application.

Read & Write with the MongoDB Data API

The MongoDB Data API is a pre-configured set of HTTPS endpoints that can be used to read and write data to a MongoDB Atlas database.

With the MongoDB Data API, you can create, read, update, delete, or aggregate documents in a MongoDB Atlas database.

Cluster Configuration

In order to use the Data API, you must first enable the functionality from the Atlas UI.

From the MongoDB Atlas dashboard, navigate to **Data API** in the left menu.

Select the data source(s) you would like to enable the API on and click **Enable the Data API**.

Access Level

By default, no access is granted. Select the access level you'd like to grant the Data API. The choices are: No Access, Read Only, Read and Write, or Custom Access.

Data API Key

In order to authenticate with the Data API, you must first create a Data API key.

Click **Create API Key**, enter a name for the key, then click **Generate API Key**.

Be sure to copy the API key and save it somewhere safe. *You will not get another chance to see this key again.*

Sending a Data API Request

We can now use the Data API to send a request to the database.

In the next example, we'll use curl to find the first document in the `movies` collection of our `sample_mflix` database. We loaded this sample data in the Intro to Aggregations section.

To run this example, you'll need your App Id, API Key, and Cluster name.

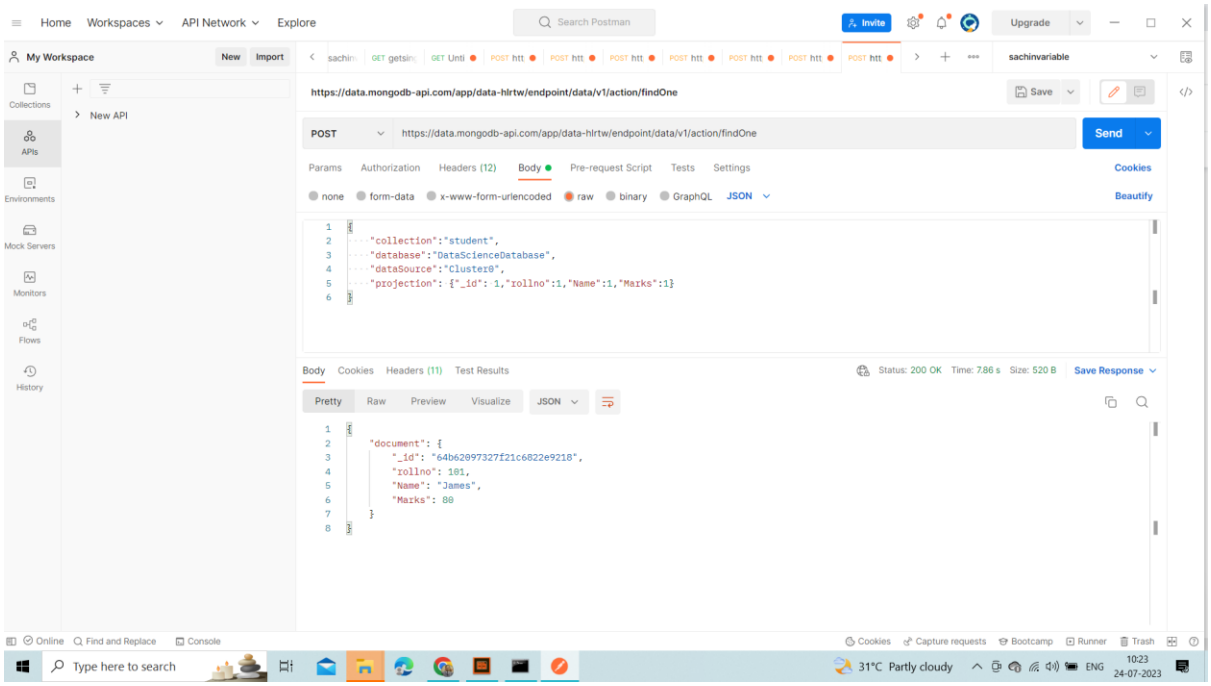
You can find your App Id in the **URL Endpoint** field of the Data API page in the MongoDB Atlas UI.

Example

```
curl --location --request POST 'https://data.mongodb-api.com/app/<DATA
API APP ID>/endpoint/data/v1/action/findOne' \

--header 'Content-Type: application/json' \

--header 'Access-Control-Request-Headers: *' \

--header 'api-key: <DATA API KEY>' \

--data-raw '{

    "dataSource":"<CLUSTER NAME>",

    "database":"sample_mflix",

    "collection":"movies",

    "projection": {"title": 1}

}'
```

```
curl --request POST \
  'https://data.mongodb-api.com/app/data-
abcde/endpoint/data/v1/action/insertOne' \
  --header 'Content-Type: application/json' \
  --header 'api-key:
TpqAKQgvhZE4r6AOzpVydJ9a3tB1BLMrgDzLlBLbihKNDzSJWTAHMVbsMoIOpnM6' \
  --data-raw '{
      "dataSource": "Cluster0",
      "database": "learn-data-api",
      "collection": "hello",
      "document": {
        "text": "Hello from the Data API!",
      }
  }'
```

```
{"document":{"_id":"573a1390f29313caabcd4135","title":"Blacksmith Scene"}}
$
```



Data API Endpoints

In the previous example, we used the `findOne` endpoint in our URL.

There are several endpoints available for use with the Data API.

All endpoints start with the Base URL: `https://data.mongodb-api.com/app/<Data API App ID>/endpoint/data/v1/action/`

Find a Single Document

Endpoint

`POST Base_URL/findOne`

The `findOne` endpoint is used to find a single document in a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
```

```
  "collection": "<collection name>",
  "filter": <query filter>,
  "projection": <projection>
}
```

Find Multiple Documents

Endpoint

`POST Base_URL/find`

The `find` endpoint is used to find multiple documents in a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "projection": <projection>,
  "sort": <sort expression>,
  "limit": <number>,
  "skip": <number>
}
```

Insert a Single Document

Endpoint

`POST Base_URL/insertOne`

The `insertOne` endpoint is used to insert a single document into a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "document": <document>
}
```

Insert Multiple Documents

Endpoint

`POST Base_URL/insertMany`

The `insertMany` endpoint is used to insert multiple documents into a collection.

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "documents": [<document>, <document>, ...]
}
```

Update a Single Document

Endpoint

```
POST Base_URL/updateOne
```

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "update": <update expression>,
  "upsert": true|false
}
```

Update Multiple Documents

Endpoint

```
POST Base_URL/updateMany
```

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>,
  "update": <update expression>,
  "upsert": true|false
}
```

Delete a Single Document

Endpoint

`POST Base_URL/deleteOne`

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>
}
```

Delete Multiple Documents

Endpoint

`POST Base_URL/deleteMany`

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "filter": <query filter>
}
```

Aggregate Documents

Endpoint

```
POST Base_URL/aggregate
```

Request Body

Example

```
{
  "dataSource": "<data source name>",
  "database": "<database name>",
  "collection": "<collection name>",
  "pipeline": [<pipeline expression>, ...]
}
```

MongoDB Drivers

MongoDB Drivers

The MongoDB Shell (mongosh) is great, but generally you will need to use MongoDB in your application. To do this, MongoDB has many language drivers.

The language drivers allow you to interact with your MongoDB database using the methods you've learned so far in `mongosh` but directly in your application.

These are the current officially supported drivers:

- C
- C++
- C#
- Go
- Java
- Node.js

- PHP
- Python
- Ruby
- Rust
- Scala
- Swift

There are other community supported libraries as well.

Let's see how to use the drivers using Node.js next.

MongoDB Node.js Database Interaction

Node.js Database Interaction

For this tutorial, we will use a MongoDB Atlas database. If you don't already have a MongoDB Atlas account, you can create one for free at MongoDB Atlas.

We will also use the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

MongoDB Node.js Driver Installation

To use MongoDB with Node.js, you will need to install the `mongodb` package in your Node.js project.

Use the following command in your terminal to install the `mongodb` package:

```
npm install mongodb
```

We can now use this package to connect to a MongoDB database.

Create an `index.js` file in your project directory.

`index.js`

```
const { MongoClient } = require('mongodb');
```

Connection String

In order to connect to our MongoDB Atlas database, we'll need to get our connection string from the Atlas dashboard.

Go to **Database** then click the **CONNECT** button on your Cluster.

Choose **Connect your application** then copy your connection string.

Example: `mongodb+srv://<username>:<password>@<cluster.string>.mongodb.net/myFirstDatabase?retryWrites=true&w=majority`

You will need to replace the `<username>`, `<password>`, and `<cluster.string>` with your MongoDB Atlas username, password, and cluster string.

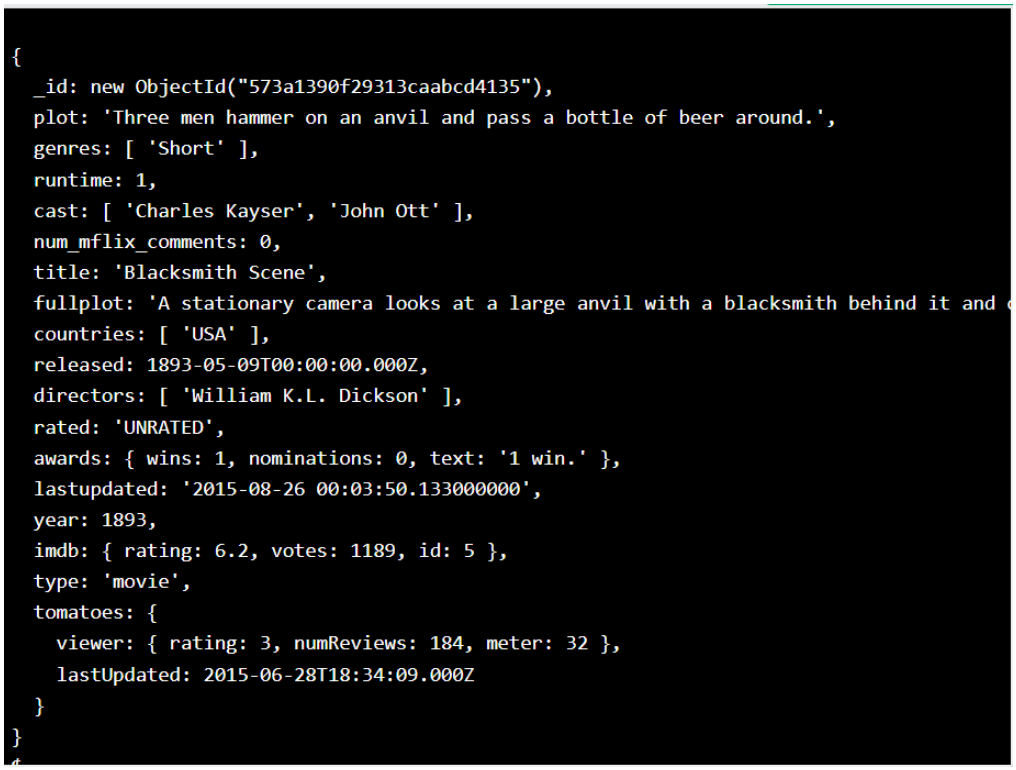
Connecting to MongoDB

Let's add to our `index.js` file.

index.js

```
const { MongoClient } = require('mongodb');


const uri = "<Your Connection String>";

const client = new MongoClient(uri);


async function run() {

  try {

    await client.connect();

    const db = client.db('sample_mflix');

    const collection = db.collection('movies');


    // Find the first document in the collection
```

```
    const first = await collection.findOne();
    console.log(first);
  } finally {
    // Close the database connection when finished or an error occurs
    await client.close();
  }
}
run().catch(console.error);
```



Run this file in your terminal.

```
node index.js
```

You should see the first document logged to the console.

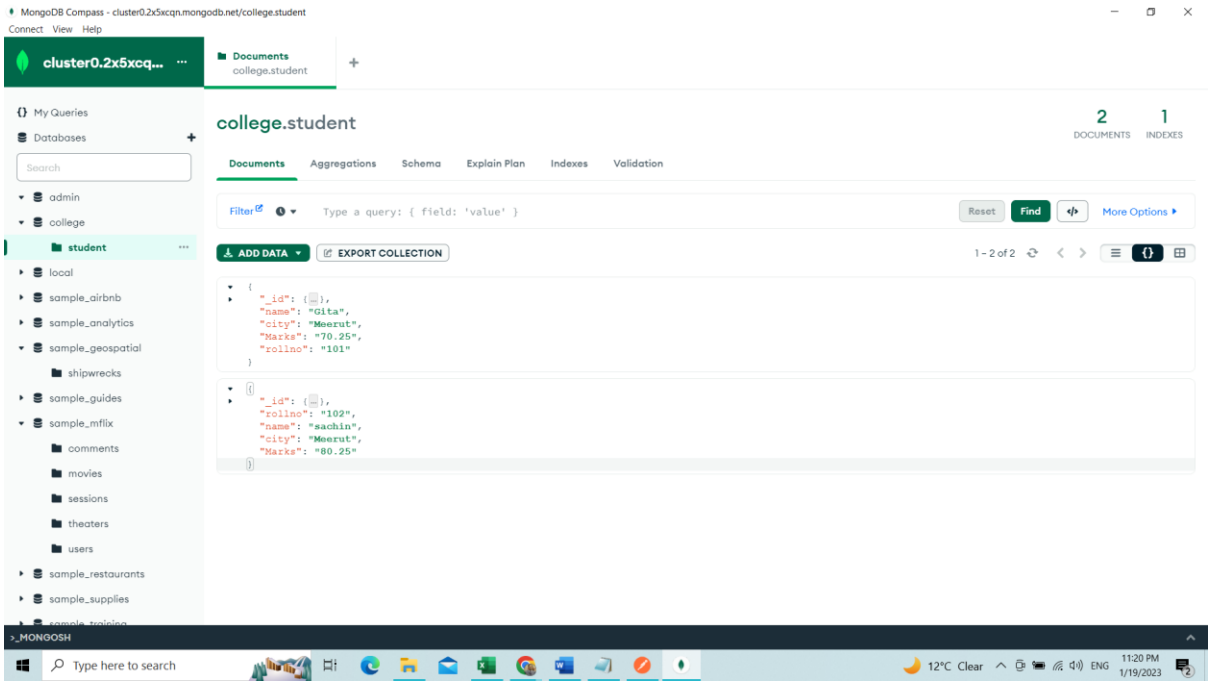
CRUD & Document Aggregation

Just as we did using `mongosh`, we can use the MongoDB Node.js language driver to create, read, update, delete, and aggregate documents in the database.

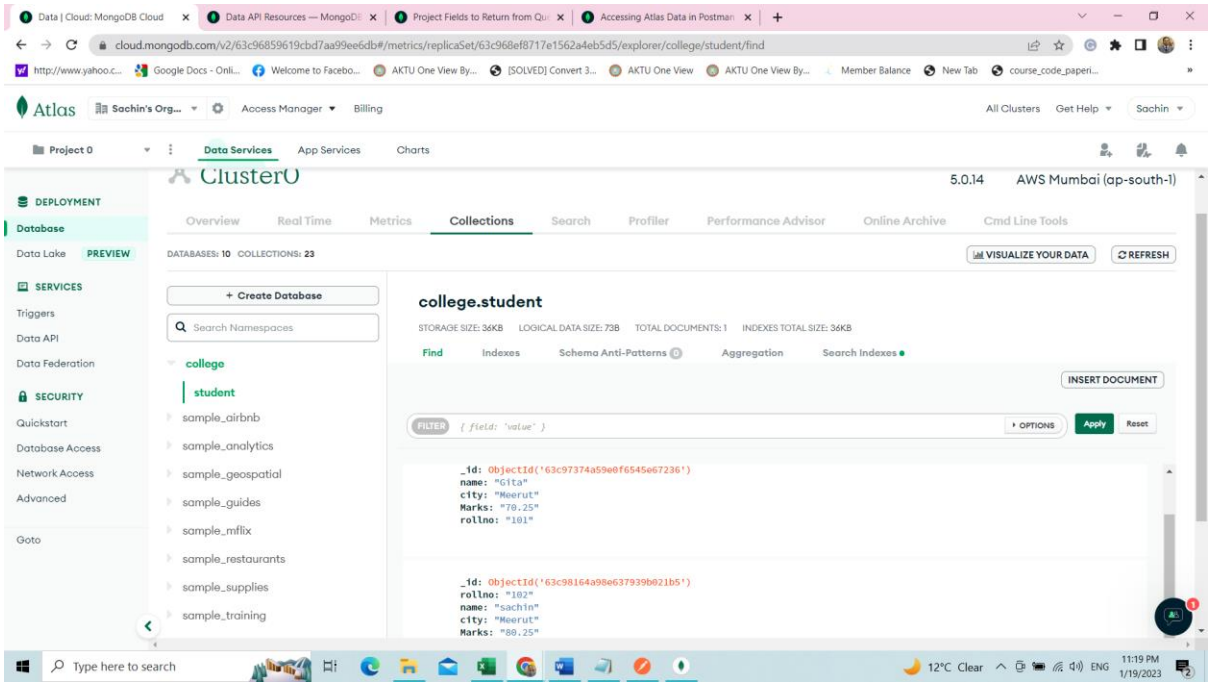
Expanding on the previous example, we can replace the `collection.findOne()` with `find()`, `insertOne()`, `insertMany()`, `updateOne()`, `updateMany()`, `deleteOne()`, `deleteMany()`, or `aggregate()`.

Give some of those a try.

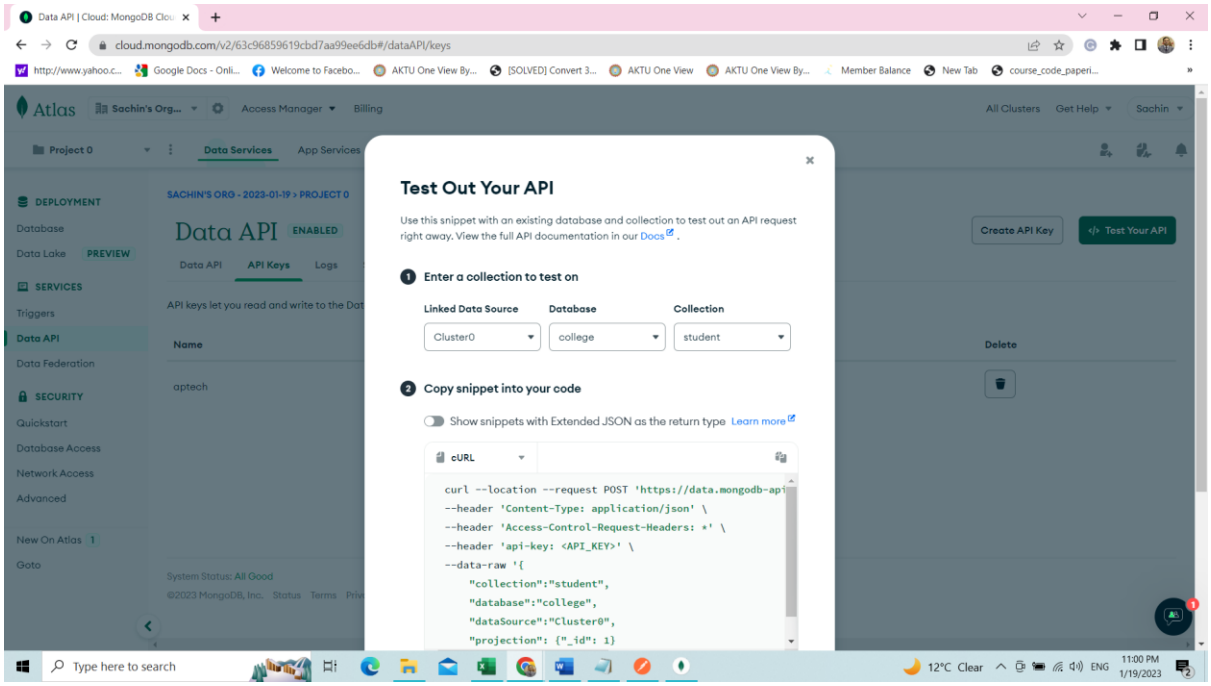
Mongodb Compass



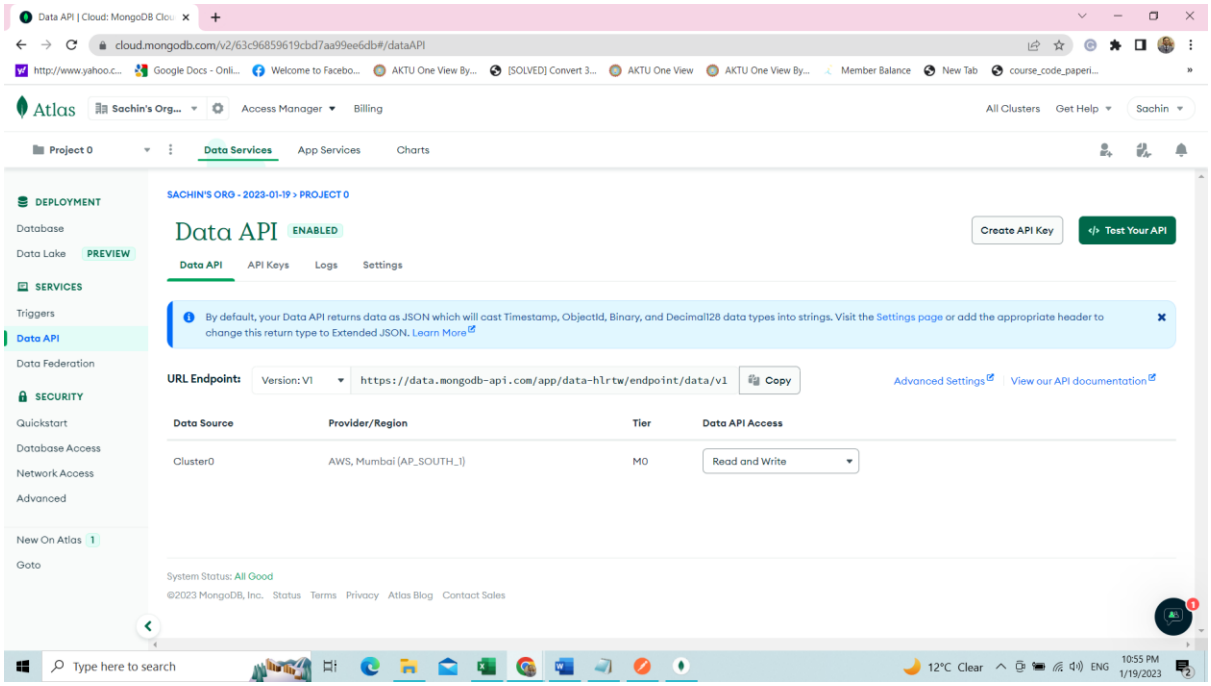
Mongodb Atlas



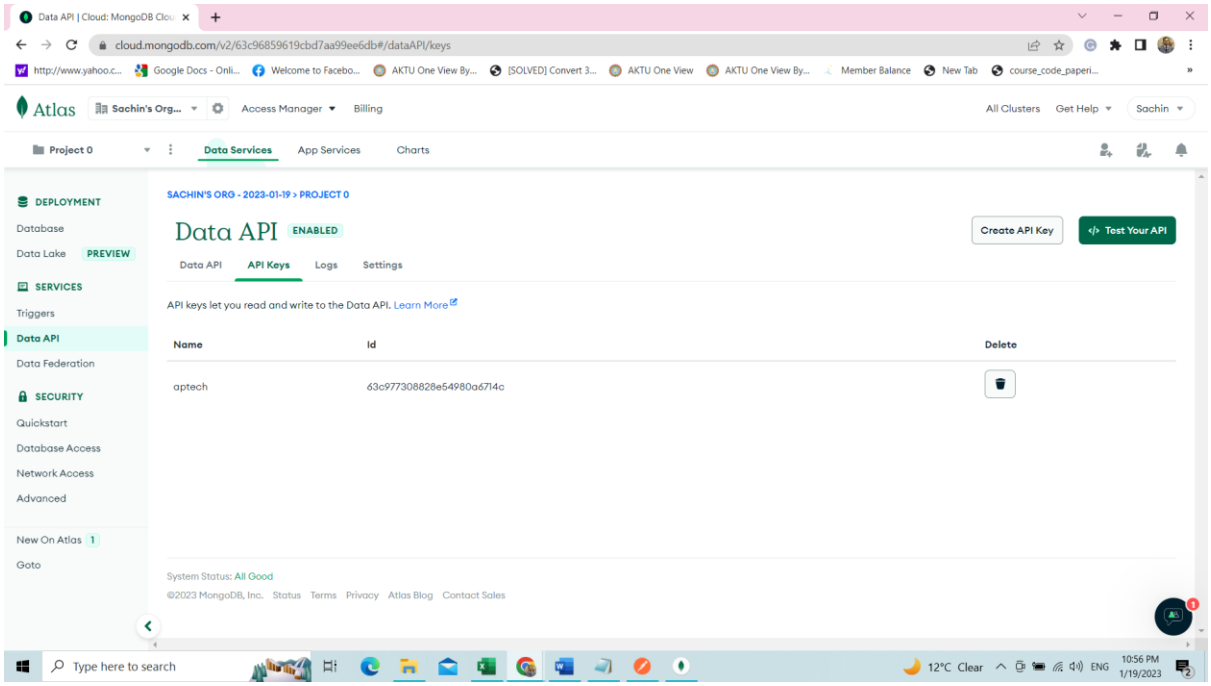
Test API Database



Data API

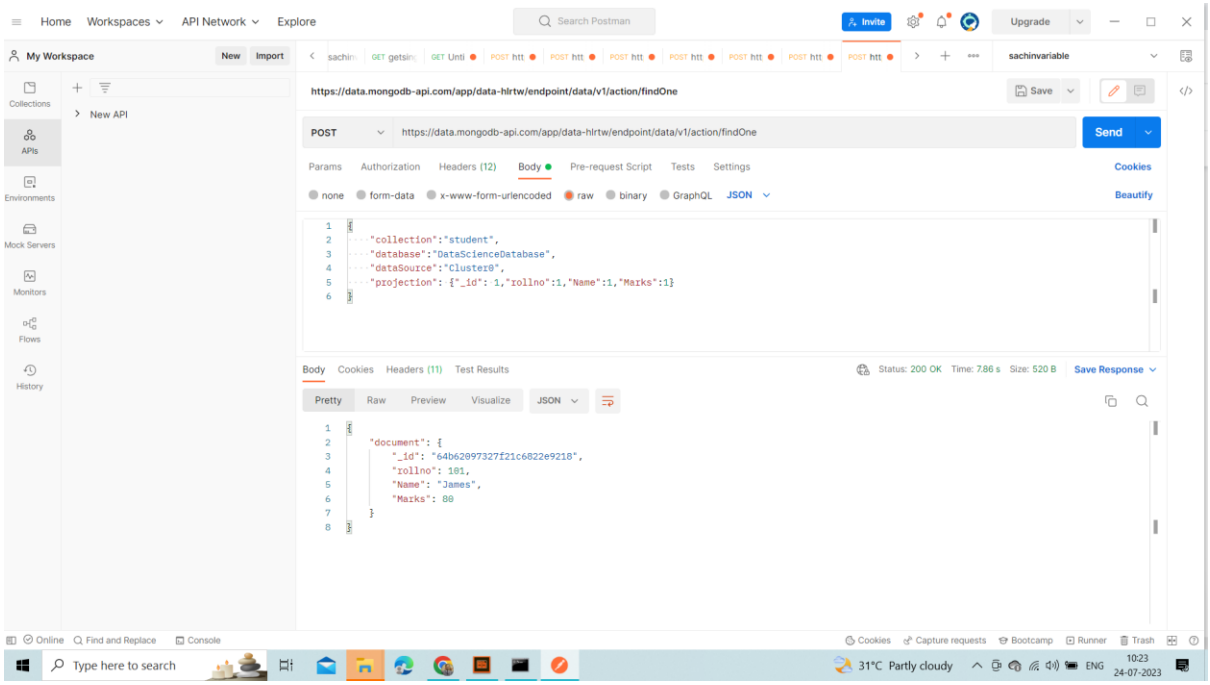


Data API Key

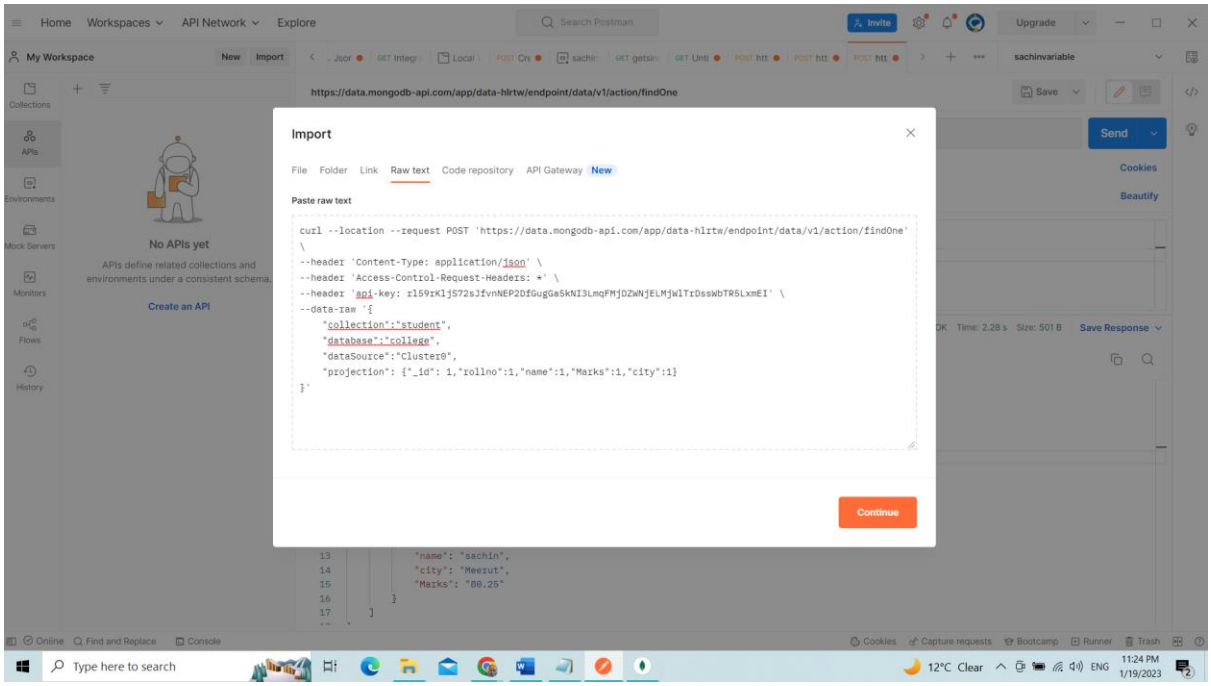


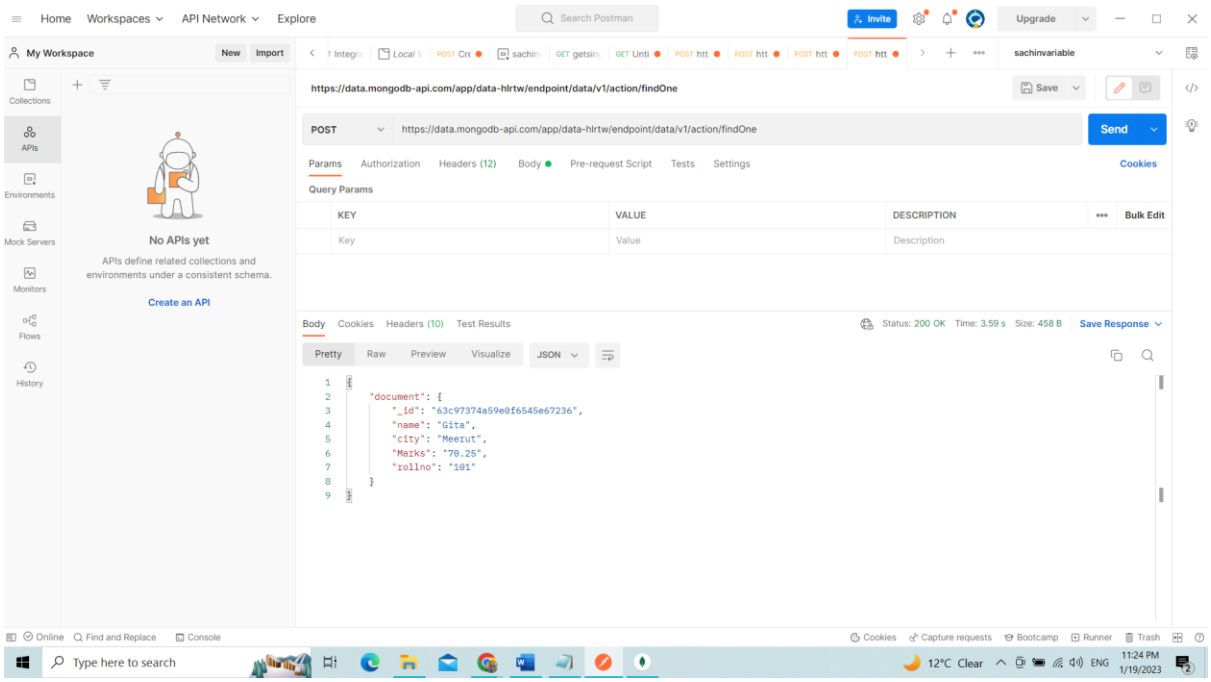
Postman Curl Command fetch data using Mongodb import

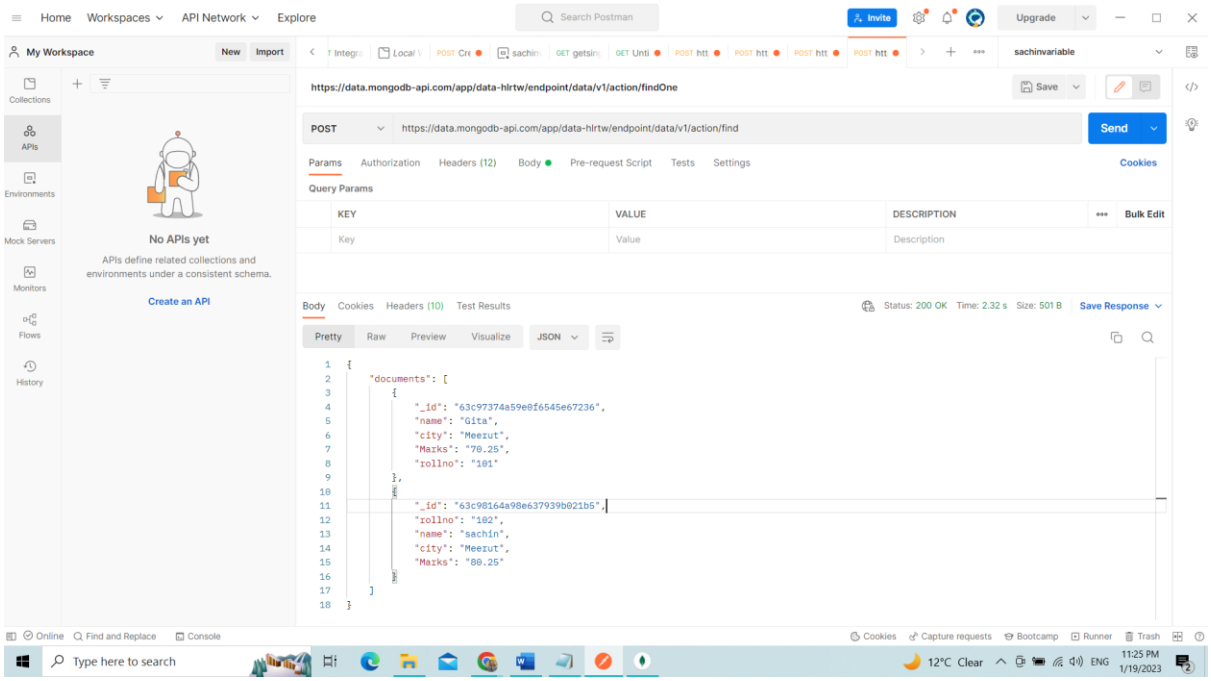
```
curl --location --request POST 'https://data.mongodb-api.com/app/data-hlrtw/endpoint/data/v1/action/findOne' \

--header 'Content-Type: application/json' \

--header 'Access-Control-Request-Headers: *' \

--header 'api-key: rl59rKljS72sJfvnNEP2DfGugGa5kNI3LmqFMjDZWNjELMjWlTrDssWbTR5LxmEI' \

--data-raw '{

    "collection":"student",

    "database":"college",

    "dataSource":"Cluster0",

    "projection": {"_id": 1,"rollno":1,"name":1,"Marks":1,"city":1}

}'
```

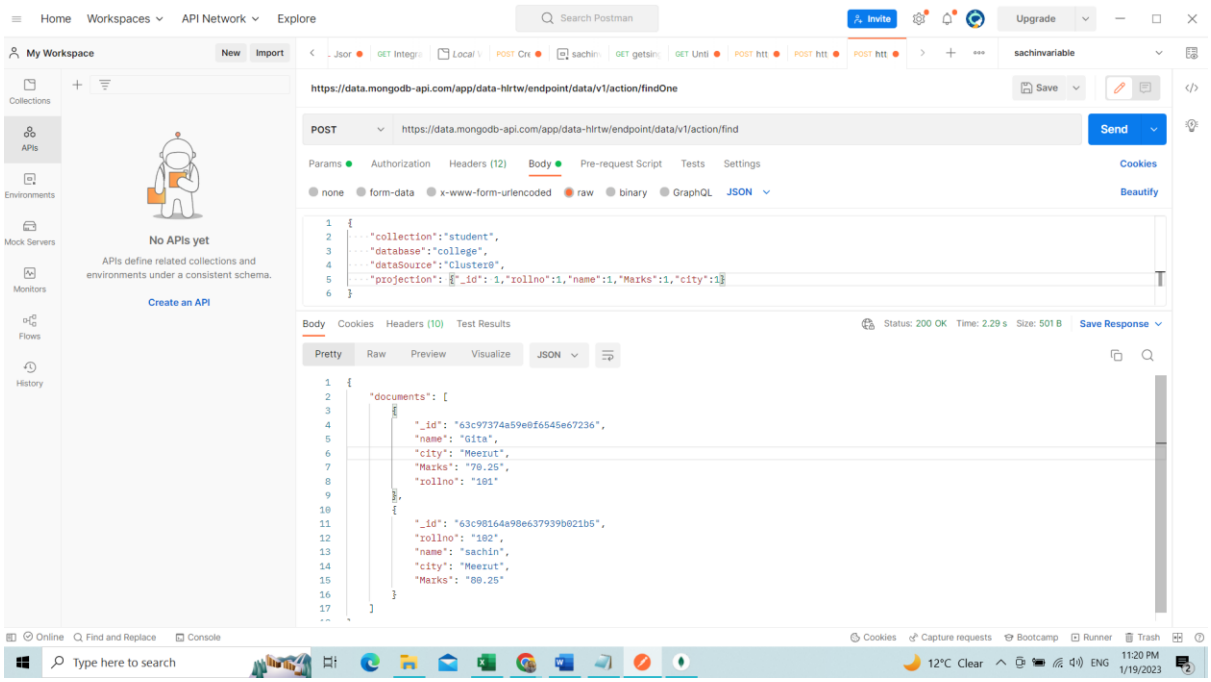


Please enter ApI key into curl command


Import
File Folder Link Raw text Code repository API Gateway New
Paste raw text
curl --location --request POST 'https://data.mongodb-api.com/app/data-hlrtw/endpoint/data/v1/action/findOne' \
--header 'Content-Type: application/json' \
--header 'Access-Control-Request-Headers: *' \
--header 'api-key: rl59rKljS72sJfvnNEP2DfGugGa5kNI3LmqFMjDZWNjELMjWlTrDssWbTR5LxmEI' \
--data-raw '{
"collection":"student",
"database":"college",
"dataSource":"Cluster0",
"projection": {"_id": 1,"rollno":1,"name":1,"Marks":1,"city":1}
}'
Continue
No APIs yet
APIs define related collections and environments under a consistent schema.
Create an API


https://data.mongodb-api.com/app/data-hlrtw/endpoint/data/v1/action/findOne
POST
https://data.mongodb-api.com/app/data-hlrtw/endpoint/data/v1/action/findOne
Send
Params Authorization Headers (12) Body Pre-request Script Tests Settings
Query Params
KEY VALUE DESCRIPTION Bulk Edit
Body Cookies Headers (10) Test Results
Status: 200 OK Time: 3.59 s Size: 458 B Save Response
Pretty Raw Preview Visualize JSON
{
"document": {
"_id": "63c97374a59e0f6545e67236",
"name": "Gita",
"city": "Meerut",
"Marks": "70.25",
"rollno": "101"
}
}
No APIs yet
APIs define related collections and environments under a consistent schema.
Create an API





For more details read this link

https://www.mongodb.com/docs/atlas/api/data-api-resources/

MongoDB Charts

MongoDB Charts

MongoDB Charts lets you visualize your data in a simple, intuitive way.

MongoDB Charts Setup

From the MongoDB Atlas dashboard, go to the **Charts** tab.

If you've never used Charts before, click the **Activate Now** button. This will take about 1 minute to complete.

You'll see a new dashboard. Click the dashboard name to open it.

Creating a Chart

Create a new chart by clicking the **Add Chart** button.

Visually creating a chart is intuitive. Select the data sources that you want to use.

Example:

In this example, we are using the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

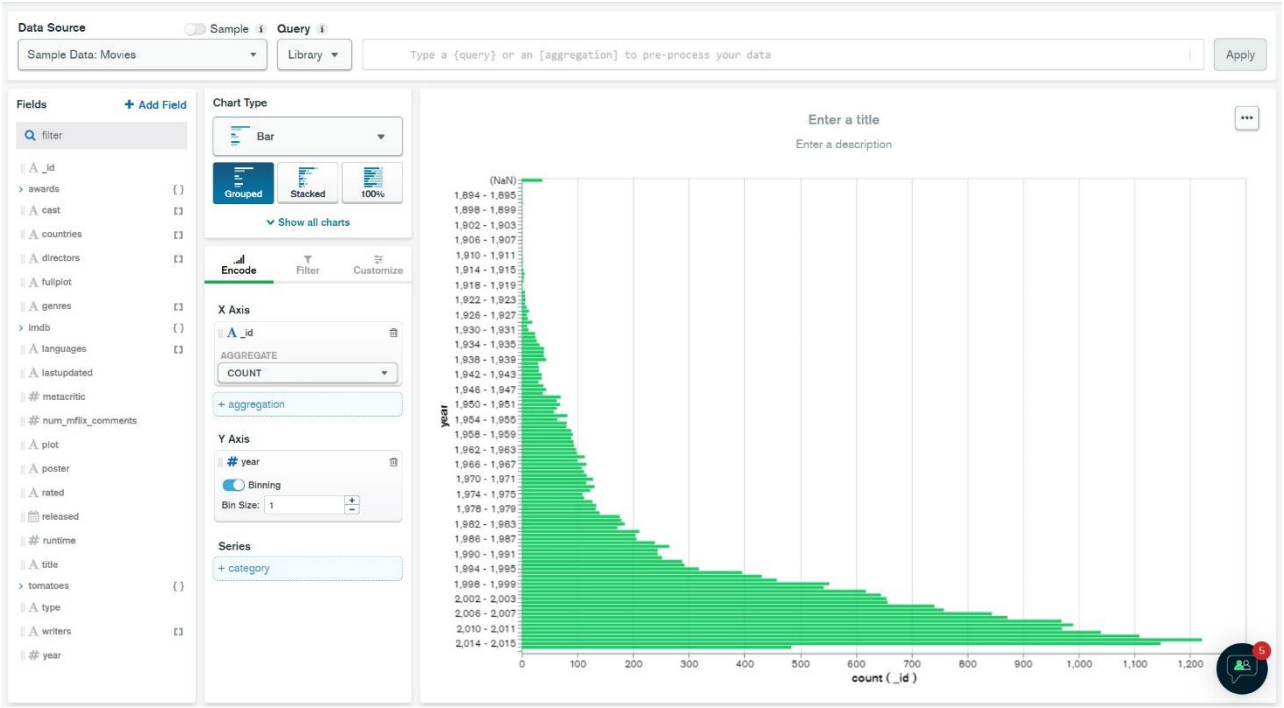
Under **Data Source**, select the **Movies** collection.

Let's visualize how many movies were released in each year.

Drag the **Year** field to the **Y Axis** field and set the **Bin Size** to 1.

Drag the **_id** field to the **X Axis** field and make sure `COUNT` is selected for the Aggregate.

You should now see a bar chart with the number of movies released in each year.



MongoDB - Create Backup

Dump MongoDB Data

To create backup of database in MongoDB, you should use **mongodump** command. This command will dump the entire data of your server into the dump directory. There are many options available by which you can limit the amount of data or create backup of your remote server.

Syntax

The basic syntax of **mongodump** command is as follows –

```
>mongodump
```

Example

Start your mongod server. Assuming that your mongod server is running on the localhost and port 27017, open a command prompt and go to the bin directory of your mongodb instance and type the command **mongodump**

Consider the mycol collection has the following data.

```
>mongodump
```

The command will connect to the server running at **127.0.0.1** and port **27017** and back all data of the server to directory **/bin/dump/**. Following is the output of the command –

Restore data

To restore backup data MongoDB's **mongorestore** command is used. This command restores all of the data from the backup directory.

Syntax

The basic syntax of **mongorestore** command is –

```
>mongorestore
```

MongoDB – Deployment

When you are preparing a MongoDB deployment, you should try to understand how your application is going to hold up in production. It's a good idea to develop a consistent, repeatable approach to managing your deployment environment so that you can minimize any surprises once you're in production.

The best approach incorporates prototyping your set up, conducting load testing, monitoring key metrics, and using that information to scale your set up. The key part of the approach is to proactively monitor your entire system - this will help you understand how your production system will hold up before deploying, and determine where you will need to add capacity. Having insight into potential spikes in your memory usage, for example, could help put out a write-lock fire before it starts.

To monitor your deployment, MongoDB provides some of the following commands –

mongostat

This command checks the status of all running mongod instances and return counters of database operations. These counters include inserts, queries, updates, deletes, and cursors. Command also shows when you're hitting page faults, and showcase your lock percentage. This means that you're running low on memory, hitting write capacity or have some performance issue.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongostat**.

```
D:\set up\mongodb\bin>mongostat
```

mongotop

This command tracks and reports the read and write activity of MongoDB instance on a collection basis. By default, **mongotop** returns information in each second, which you can change it accordingly. You should check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time, reading too frequently from a disk, or are exceeding your working set size.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongotop**.

```
D:\set up\mongodb\bin>mongotop
```

To change **mongotop** command to return information less frequently, specify a specific number after the mongotop command.

```
D:\set up\mongodb\bin>mongotop 30
```

The above example will return values every 30 seconds.

Apart from the MongoDB tools, 10gen provides a free, hosted monitoring service, MongoDB Management Service (MMS), that provides a dashboard and gives you a view of the metrics from your entire cluster.

All MongoDb commands you will ever need

we will see a comprehensive list of all the MongoDB commands you will ever need as a MongoDB beginner. This list covers almost all the most used commands in MongoDB.
I will assume that you are working inside a collection named 'comments' on a MongoDB database of your choice

1. Database Commands

View all databases

```
show dbs
```

Create a new or switch databases

```
use dbName
```

View current Database

```
db
```

Delete Database

```
db.dropDatabase()
```

2. Collection Commands

Show Collections

```
show collections
```

Create a collection named 'comments'

```
db.createCollection('comments')
```

Drop a collection named 'comments'

```
db.comments.drop()
```

3. Row(Document) Commands

Show all Rows in a Collection

```
db.comments.find()
```

Show all Rows in a Collection (Prettified)

```
db.comments.find().pretty()
```

Find the first row matching the object

```
db.comments.findOne({name: 'Sachin'})
```

Insert One Row

```
db.comments.insert({
    'name': 'Sachin',
    'lang': 'JavaScript',
    'member_since': 5
 })
```

Insert many Rows

```
db.comments.insertMany([{
    'name': 'Sachin',
    'lang': 'JavaScript',
    'member_since': 5
    },
    {'name': 'Rohan',
    'lang': 'Python',
    'member_since': 3
    },
    {'name': 'Gita',
    'lang': 'Java',
    'member_since': 4
}])
```

Search in a MongoDb Database

```
db.comments.find({lang:'Python'})
```

Limit the number of rows in output

```
db.comments.find().limit(2)
```

Count the number of rows in the output

```
db.comments.find().count()
```

Update a row

```
db.comments.updateOne({name: 'Sachin'},
{$set: {'name': 'Ram',
    'lang': 'JavaScript',
    'member_since': 51
}}, {upsert: true})
```

Mongodb Increment Operator

```
db.comments.update({name: 'Rohan'},
{$inc:{
    member_since: 2
}})
```

Mongodb Rename Operator

```
db.comments.update({name: 'Rohan'},
{$rename:{
    member_since: 'member'
}})
```

Delete Row

```
db.comments.remove({name: 'Sachin'})
```

Less than/Greater than/ Less than or Eq/Greater than or Eq

```
db.comments.find({member_since: {$lt: 90}})
db.comments.find({member_since: {$lte: 90}})
db.comments.find({member_since: {$gt: 90}})
db.comments.find({member_since: {$gte: 90}})
```

RockMongo

RockMongo is a MongoDB administration tool using which you can manage your server, databases, collections, documents, indexes, and a lot more. It provides a very user-friendly way for reading, writing, and creating documents. It is similar to PHPMyAdmin tool for PHP and MySQL.

GridFS

GridFS is the MongoDB specification for storing and retrieving large files such as images, audio files, video files, etc. It is kind of a file system to store files but its data is stored within MongoDB collections. GridFS has the capability to store files even greater than its document size limit of 16MB.

GridFS divides a file into chunks and stores each chunk of data in a separate document, each of maximum size 255k.

GridFS by default uses two collections **fs.files** and **fs.chunks** to store the file's metadata and the chunks. Each chunk is identified by its unique _id ObjectId field. The fs.files serves as a parent document. The **files_id** field in the fs.chunks document links the chunk to its parent.

Following is a sample document of fs.files collection –

```
{
  "filename": "test.txt",
  "chunkSize": NumberInt(261120),
```

```
  "uploadDate": ISODate("2014-04-13T11:32:33.557Z"),
  "md5": "7b762939321e146569b07f72c62cca4f",
  "length": NumberInt(646)
}
```

The document specifies the file name, chunk size, uploaded date, and length.

Following is a sample document of fs.chunks document –

```
{
  "files_id": ObjectId("534a75d19f54bfec8a2fe44b"),
  "n": NumberInt(0),
  "data": "Mongo Binary Data"
}
```

Adding Files to GridFS

Now, we will store an mp3 file using GridFS using the **put** command. For this, we will use the **mongofiles.exe** utility present in the bin folder of the MongoDB installation folder.

Open your command prompt, navigate to the mongofiles.exe in the bin folder of MongoDB installation folder and type the following code –

```
>mongofiles.exe -d gridfs put song.mp3
```

Here, **gridfs** is the name of the database in which the file will be stored. If the database is not present, MongoDB will automatically create a new document on the fly. Song.mp3 is the name of the file uploaded. To see the file's document in database, you can use find query –

```
>db.fs.files.find()
```

The above command returned the following document –

```
{
  _id: ObjectId('534a811bf8b4aa4d33fdf94d'),
  filename: "song.mp3",
  chunkSize: 261120,
  uploadDate: new Date(1397391643474), md5: "e4f53379c909f7bed2e9d631e15c1c41",
  length: 10401959
}
```

We can also see all the chunks present in fs.chunks collection related to the stored file with the following code, using the document id returned in the previous query –

```
>db.fs.chunks.find({files_id:ObjectId('534a811bf8b4aa4d33fdf94d')})
```

In my case, the query returned 40 documents meaning that the whole mp3 document was divided in 40 chunks of data.

Capped collections

Capped collections are fixed-size circular collections that follow the insertion order to support high performance for create, read, and delete operations. By circular, it means that when the fixed size allocated to the collection is exhausted, it will start deleting the oldest document in the collection without providing any explicit commands.

Capped collections restrict updates to the documents if the update results in increased document size. Since capped collections store documents in the order of the disk storage, it ensures that the document size does not increase the size allocated on the disk. Capped collections are best for storing log information, cache data, or any other high volume data.

Creating Capped Collection

To create a capped collection, we use the normal createCollection command but with **capped** option as **true** and specifying the maximum size of collection in bytes.

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000})
```

In addition to collection size, we can also limit the number of documents in the collection using the **max** parameter –

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000,max:1000})
```

MongoDB - Auto-Increment Sequence

MongoDB does not have out-of-the-box auto-increment functionality, like SQL databases. By default, it uses the 12-byte ObjectId for the **_id** field as the primary key to uniquely identify the documents. However, there may be scenarios where we may want the _id field to have some auto-incremented value other than the ObjectId.

Since this is not a default feature in MongoDB, we will programmatically achieve this functionality by using a **counters** collection as suggested by the MongoDB documentation.

Using Counter Collection

Consider the following **products** document. We want the _id field to be an **auto-incremented integer sequence** starting from 1,2,3,4 upto n.

```
{
 "_id":1,
 "product_name": "Apple iPhone",
 "category": "mobiles"
}
```

For this, create a **counters** collection, which will keep track of the last sequence value for all the sequence fields.

```
>db.createCollection("counters")
```

Now, we will insert the following document in the counters collection with **productid** as its key –

```
> db.counters.insert({
            "_id":"productid",
            "sequence_value": 0
})
WriteResult({ "nInserted" : 1 })
>
```

The field **sequence_value** keeps track of the last value of the sequence.

Use the following code to insert this sequence document in the counters collection –

```
>db.counters.insert({_id:"productid",sequence_value:0})
```

Replication

Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability with multiple copies of data on different database servers. Replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

Why Replication?

- To keep your data safe
- High (24*7) availability of data
- Disaster recovery

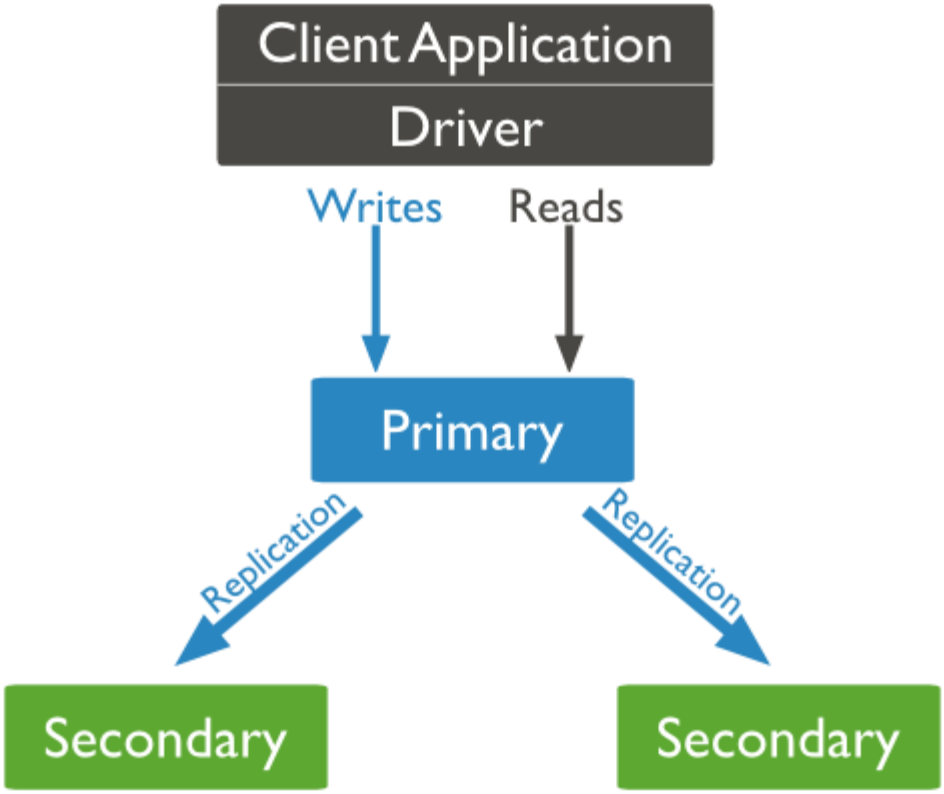
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

How Replication Works in MongoDB

MongoDB achieves replication by the use of replica set. A replica set is a group of **mongod** instances that host the same data set. In a replica, one node is primary node that receives all write operations. All other instances, such as secondaries, apply operations from the primary so that they have the same data set. Replica set can have only one primary node.

- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica set, one node is primary node and remaining nodes are secondary.
- All data replicates from primary to secondary node.
- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
- After the recovery of failed node, it again join the replica set and works as a secondary node.

A typical diagram of MongoDB replication is shown in which client application always interact with the primary node and the primary node then replicates the data to the secondary nodes.



Replica Set Features

- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary
- Automatic failover
- Automatic recovery
- Consensus election of primary

Set Up a Replica Set

In this tutorial, we will convert standalone MongoDB instance to a replica set. To convert to replica set, following are the steps –

- Shutdown already running MongoDB server.
-
- Start the MongoDB server by specifying -- replSet option. Following is the basic syntax of --replSet –

```
mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet
"REPLICA_SET_INSTANCE_NAME"
```

Example

```
mongod --port 27017 --dbpath "D:\set up\mongodb\data" --replSet rs0
```

- It will start a mongod instance with the name rs0, on port 27017.
- Now start the command prompt and connect to this mongod instance.

- In Mongo client, issue the command **rs.initiate()** to initiate a new replica set.
- To check the replica set configuration, issue the command **rs.conf()**. To check the status of replica set issue the command **rs.status()**.

Add Members to Replica Set

To add members to replica set, start mongod instances on multiple machines. Now start a mongo client and issue a command **rs.add()**.

Syntax

The basic syntax of **rs.add()** command is as follows –

```
>rs.add(HOST_NAME:PORT)
```

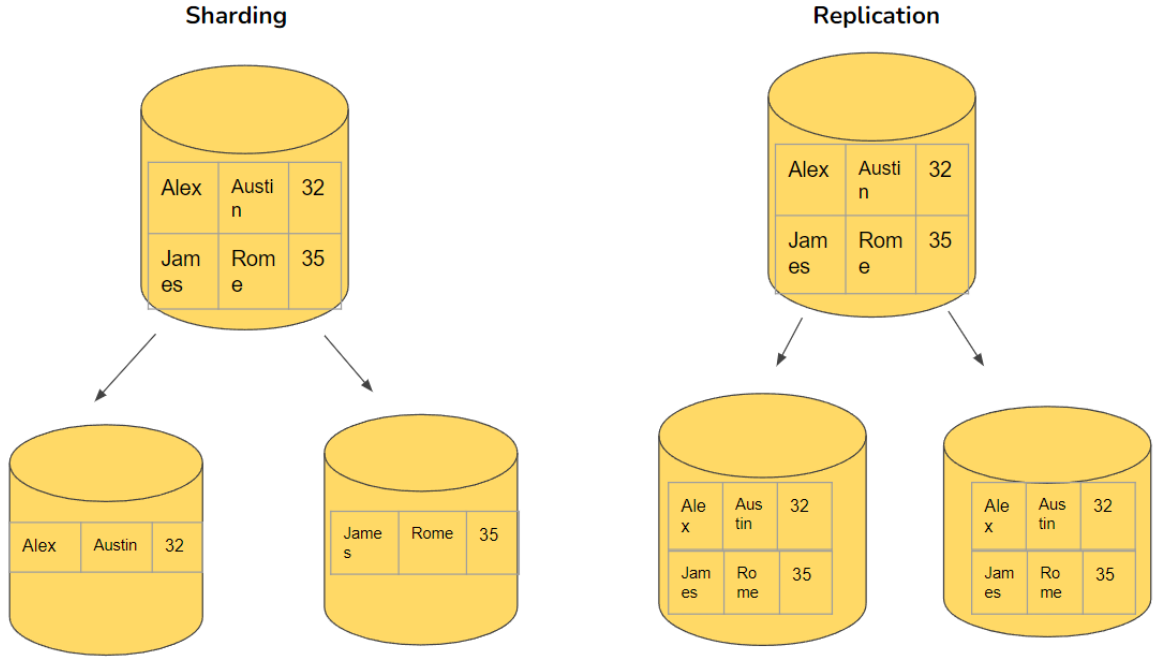
Example

Suppose your mongod instance name is **mongod1.net** and it is running on port **27017**. To add this instance to replica set, issue the command **rs.add()** in Mongo client.

```
>rs.add("mongod1.net:27017")
```

MongoDB – Sharding

- **Sharding:** Sharding is the process of splitting data up across machines.



Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data

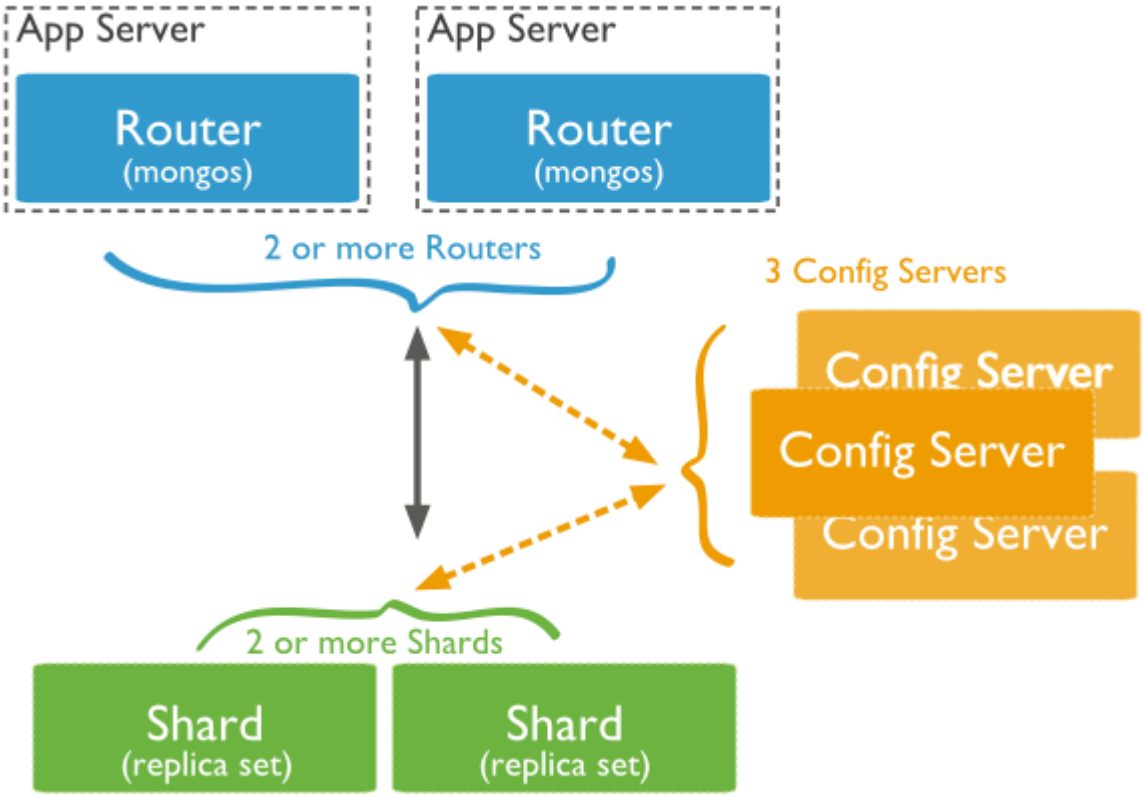
increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Why Sharding?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

Sharding in MongoDB

The following diagram shows the Sharding in MongoDB using sharded cluster.



In the following diagram, there are three main components –

- **Shards** – Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.
- **Config Servers** – Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query

router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.

- **Query Routers** – Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

Backup

Dump MongoDB Data

To create backup of database in MongoDB, you should use **mongodump** command. This command will dump the entire data of your server into the dump directory. There are many options available by which you can limit the amount of data or create backup of your remote server.

Syntax

The basic syntax of **mongodump** command is as follows –

```
>mongodump
```

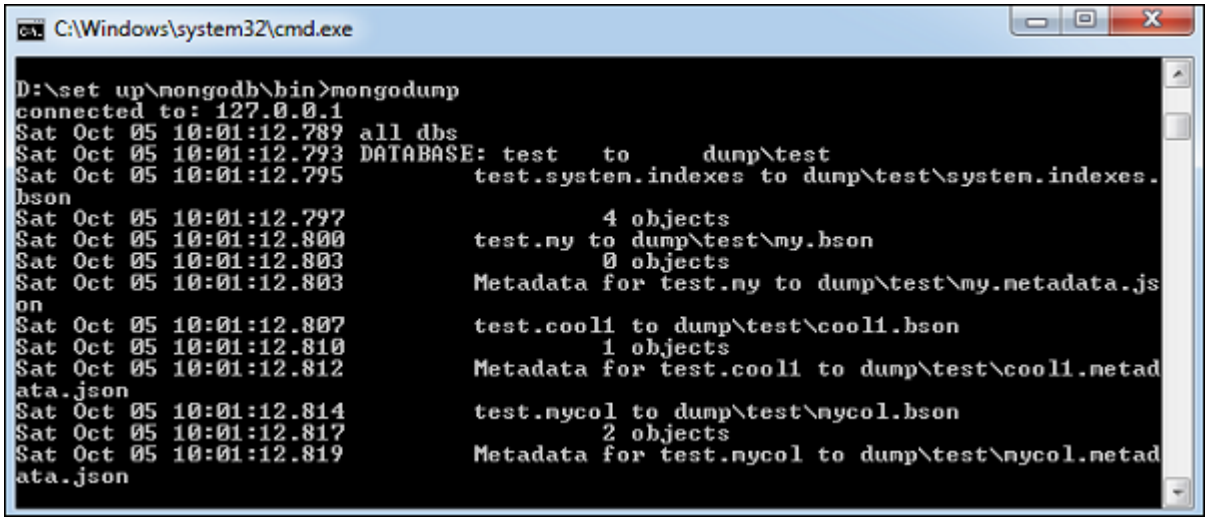
Example

Start your mongod server. Assuming that your mongod server is running on the localhost and port 27017, open a command prompt and go to the bin directory of your mongodb instance and type the command **mongodump**

Consider the mycol collection has the following data.

```
>mongodump
```

The command will connect to the server running at **127.0.0.1** and port **27017** and back all data of the server to directory **/bin/dump/**. Following is the output of the command –



Following is a list of available options that can be used with the **mongodump** command.

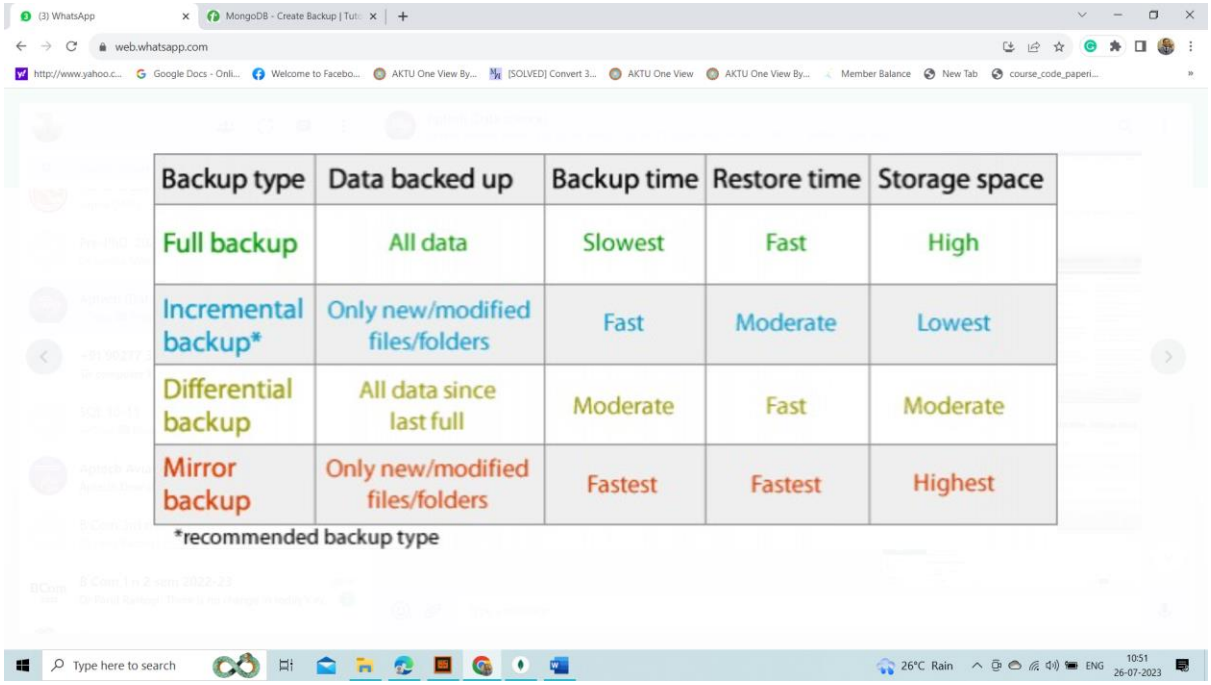
Syntax	Description	Example
mongodump --host HOST_NAME --port PORT_NUMBER	This commmand will backup all databases of specified mongod instance.	mongodump --host tutorialspoint.com --port 27017
mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY	This command will backup only specified database at specified path.	mongodump --dbpath /data/db/ --out /data/backup/
mongodump --collection COLLECTION --db DB_NAME	This command will backup only specified collection of specified database.	mongodump --collection mycol --db test

Restore data

To restore backup data MongoDB's **mongorestore** command is used. This command restores all of the data from the backup directory.

Syntax

The basic syntax of **mongorestore** command is –



Backup type	Data backed up	Backup time	Restore time	Storage space
Full backup	All data	Slowest	Fast	High
Incremental backup*	Only new/modified files/folders	Fast	Moderate	Lowest
Differential backup	All data since last full	Moderate	Fast	Moderate
Mirror backup	Only new/modified files/folders	Fastest	Fastest	Highest

*recommended backup type

Deployment

When you are preparing a MongoDB deployment, you should try to understand how your application is going to hold up in production. It's a good idea to develop a consistent, repeatable approach to managing your deployment environment so that you can minimize any surprises once you're in production.

The best approach incorporates prototyping your set up, conducting load testing, monitoring key metrics, and using that information to scale your set up. The key part of the approach is to proactively monitor your entire system - this will help you understand how your production system will hold up before deploying, and determine where you will need to add capacity. Having insight into potential spikes in your memory usage, for example, could help put out a write-lock fire before it starts.

To monitor your deployment, MongoDB provides some of the following commands –

mongostat

This command checks the status of all running mongod instances and return counters of database operations. These counters include inserts, queries, updates, deletes, and cursors. Command also shows when you're hitting page faults, and showcase your lock percentage. This means that you're running low on memory, hitting write capacity or have some performance issue.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongostat**.

```
D:\set up\mongodb\bin>mongostat
```

MongoDB - Relationships

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

Following is the sample document structure of **user** document –

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

Following is the sample document structure of **address** document –

```
{
  "_id":ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
> db.users.insert({
        {
                "_id":ObjectId("52ffc33cd85242f436000001"),
                "contact": "987654321",
                "dob": "01-01-1991",
                "name": "Tom Benzamin",
                "address": [
                        {
```

```
                                        "building": "22 A, Indiana Apt",
                                        "pincode": 123456,
                                        "city": "Los Angeles",
                                        "state": "California"
                                },
                                {
                                        "building": "170 A, Acropolis Apt",
                                        "pincode": 456789,
                                        "city": "Chicago",
                                        "state": "Illinois"
                                }
                        ]
        }
})
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as –

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Note that in the above query, **db** and **users** are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

MongoDB - Relationships

Previous Page

Next Page

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

Following is the sample document structure of **user** document –

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

Following is the sample document structure of **address** document –

```
{
  "_id":ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
> db.users.insert({
        {
                "_id":ObjectId("52ffc33cd85242f436000001"),
                "contact": "987654321",
                "dob": "01-01-1991",
                "name": "Tom Benzamin",
                "address": [
                        {
                                "building": "22 A, Indiana Apt",
                                "pincode": 123456,
```

```
                                        "city": "Los Angeles",
                                        "state": "California"
                                },
                                {
                                        "building": "170 A, Acropolis Apt",
                                        "pincode": 456789,
                                        "city": "Chicago",
                                        "state": "Illinois"
                                }
                        ]
        }
})
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as –

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Note that in the above query, **db** and **users** are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

What is a Covered Query?

As per the official MongoDB documentation, a covered query is a query in which –

- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.

Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries

To test covered queries, consider the following document in the **users** collection –

```
{
  "_id": ObjectId("53402597d852426020000003"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "gender": "M",
  "name": "Tom Benzamin",
  "user_name": "tombenzamin"
}
```

We will first create a compound index for the **users** collection on the fields **gender** and **user_name** using the following query –

```
>db.users.createIndex({gender:1,user_name:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Now, this index will cover the following query –

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
{ "user_name" : "tombenzamin" }
```

That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.

Since our index does not include **_id** field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above –

```
>db.users.find({gender:"M"},{user_name:1})
{ "_id" : ObjectId("53402597d852426020000003"), "user_name" : "tombenzamin" }
```

Lastly, remember that an index cannot cover a query if –

- Any of the indexed fields is an array
- Any of the indexed fields is a subdocument

Model Data for Atomic Operations

The recommended approach to maintain atomicity would be to keep all the related information, which is frequently updated together in a single document using **embedded documents**. This would make sure that all the updates for a single document are atomic.

Assume we have created a collection with name productDetails and inserted a documents in it as shown below –

```
>db.createCollection("products")
{ "ok" : 1 }
> db.productDetails.insert(
        {
                "_id":1,
                "product_name": "Samsung S3",
                "category": "mobiles",
                "product_total": 5,
                "product_available": 3,
                "product_bought_by": [
                        {
                                "customer": "john",
                                "date": "7-Jan-2014"
                        },
                        {
                                "customer": "mark",
                                "date": "8-Jan-2014"
                        }
                ]
        }
)
WriteResult({ "nInserted" : 1 })
>
```

In this document, we have embedded the information of the customer who buys the product in the **product_bought_by** field. Now, whenever a new customer buys the product, we will first check if the product is still available

using **product_available** field. If available, we will reduce the value of product_available field as well as insert the new customer's embedded document in the product_bought_by field. We will use **findAndModify** command for this functionality because it searches and updates the document in the same go.

```
>db.products.findAndModify({
  query:{_id:2,product_available:{$gt:0}},
  update:{
    $inc:{product_available:-1},
    $push:{product_bought_by:{customer:"rob",date:"9-Jan-2014"}}
  }
})
```

Our approach of embedded document and using findAndModify query makes sure that the product purchase information is updated only if it the product is available. And the whole of this transaction being in the same query, is atomic.

In contrast to this, consider the scenario where we may have kept the product availability and the information on who has bought the product, separately. In this case, we will first check if the product is available using the first query. Then in the second query we will update the purchase information. However, it is possible that between the executions of these two queries, some other user has purchased the product and it is no more available. Without knowing this, our second query will update the purchase information based on the result of our first query. This will make the database inconsistent because we have sold a product which is not available.

MongoDB - Advanced Indexing

we have inserted the following document in the collection named users as shown below –

```
db.users.insert(
        {
                "address": {
                        "city": "Los Angeles",
                        "state": "California",
                        "pincode": "123"
                },
                "tags": [
                        "music",
                        "cricket",
                        "blogs"
                ],
                "name": "Tom Benzamin"
        }
)
```

The above document contains an **address sub-document** and a **tags array**.

Indexing Array Fields

Suppose we want to search user documents based on the user's tags. For this, we will create an index on tags array in the collection.

Creating an index on array in turn creates separate index entries for each of its fields. So in our case when we create an index on tags array, separate indexes will be created for its values music, cricket and blogs.

To create an index on tags array, use the following code –

```
>db.users.createIndex({"tags":1})
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 2,
"numIndexesAfter" : 3,
"ok" : 1
}
>
```

After creating the index, we can search on the tags field of the collection like this –

```
> db.users.find({tags:"cricket"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
>
```

To verify that proper indexing is used, use the following **explain** command –

```
>db.users.find({tags:"cricket"}).explain()
```

This gives you the following result –

```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydb.users",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "tags" : {
                                "$eq" : "cricket"
```

```
                        }
                },
                "queryHash" : "9D3B61A7",
                "planCacheKey" : "04C9997B",
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "tags" : 1
                                },
                                "indexName" : "tags_1",
                                "isMultiKey" : false,
                                "multiKeyPaths" : {
                                        "tags" : [ ]
                                },
                                "isUnique" : false,
                                "isSparse" : false,
                                "isPartial" : false,
                                "indexVersion" : 2,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "tags" : [
                                                "[\"cricket\", \"cricket\"]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "Krishna",
                "port" : 27017,
                "version" : "4.2.1",
                "gitVersion" : "edf6d45851c0b9ee15548f0f847df141764a317e"
        },
        "ok" : 1
}
>
```

The above command resulted in "cursor" : "BtreeCursor tags_1" which confirms that proper indexing is used.

Indexing Sub-Document Fields

Suppose that we want to search documents based on city, state and pincode fields. Since all these fields are part of address sub-document field, we will create an index on all the fields of the sub-document.

For creating an index on all the three fields of the sub-document, use the following code –

```
>db.users.createIndex({"address.city":1,"address.state":1,"address.pincode":1})
{
        "numIndexesBefore" : 4,
        "numIndexesAfter" : 4,
        "note" : "all indexes already exist",
        "ok" : 1
}
>
```

Once the index is created, we can search for any of the sub-document fields utilizing this index as follows –

```
> db.users.find({"address.city":"Los Angeles"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
```

Remember that the query expression has to follow the order of the index specified. So the index created above would support the following queries –

```
>db.users.find({"address.city":"Los Angeles","address.state":"California"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
>
```

MongoDB - Indexing Limitations

Previous Page

Next Page

In this chapter, we will learn about Indexing Limitations and its other components.

Extra Overhead

Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage

Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

Query Limitations

Indexing can't be used in queries which use –

- Regular expressions or negation operators like $nin, $not, etc.
- Arithmetic operators like $mod, etc.
- $where clause

Hence, it is always advisable to check the index usage for your queries.

Index Key Limits

Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Inserting Documents Exceeding Index Key Limit

MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.

MongoDB - Indexing Limitations

Previous Page

Next Page

In this chapter, we will learn about Indexing Limitations and its other components.

Extra Overhead

Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage

Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

Query Limitations

Indexing can't be used in queries which use –

- Regular expressions or negation operators like $nin, $not, etc.
- Arithmetic operators like $mod, etc.
- $where clause

Hence, it is always advisable to check the index usage for your queries.

Index Key Limits

Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Inserting Documents Exceeding Index Key Limit

MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.

MongoDB - Map Reduce

Previous Page

Next Page

As per the MongoDB documentation, **Map-reduce** is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses **mapReduce** command for map-reduce operations. MapReduce is generally used for processing large data sets.

MapReduce Command

Following is the syntax of the basic mapReduce command –

```
>db.collection.mapReduce(
  function() {emit(key,value);},  //map function
  function(key,values) {return reduceFunction}, {   //reduce function
    out: collection,
    query: document,
    sort: document,
    limit: number
  }
)
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.

In the above syntax –

- **map** is a javascript function that maps a value with a key and emits a key-value pair
- **reduce** is a javascript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria
- **limit** specifies the optional maximum number of documents to be returned

Using MapReduce

Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
  "post_text": "tutorialspoint is an awesome website for tutorials",
```

```
  "user_name": "mark",
  "status":"active"
}
```

Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code –

```
>db.posts.mapReduce(
  function() { emit(this.user_id,1); },

  function(key, values) {return Array.sum(values)}, {
    query:{status:"active"},
    out:"post_total"
  }
)
```

The above mapReduce query outputs the following result –

```
{
  "result" : "post_total",
  "timeMillis" : 9,
  "counts" : {
    "input" : 4,
    "emit" : 4,
    "reduce" : 2,
    "output" : 2
  },
  "ok" : 1,
}
```

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

To see the result of this mapReduce query, use the find operator –

```
>db.posts.mapReduce(
  function() { emit(this.user_id,1); },
  function(key, values) {return Array.sum(values)}, {
    query:{status:"active"},
    out:"post_total"
  }

).find()
```

The above query gives the following result which indicates that both users **tom** and **mark** have two posts in active states –

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

In a similar manner, MapReduce queries can be used to construct large complex aggregation queries. The use of custom Javascript functions make use of MapReduce which is very flexible and powerful.

MongoDB - Text Search

Previous Page

Next Page

Starting from version 2.4, MongoDB started supporting text indexes to search inside string content. The **Text Search** uses stemming techniques to look for specified words in the string fields by dropping stemming stop words like **a, an, the,** etc. At present, MongoDB supports around 15 languages.

Enabling Text Search

Initially, Text Search was an experimental feature but starting from version 2.6, the configuration is enabled by default.

Creating Text Index

Consider the following document under **posts** collection containing the post text and its tags –

```
> db.posts.insert({
  "post_text": "enjoy the mongodb articles on tutorialspoint",
  "tags": ["mongodb", "tutorialspoint"]
}
{
        "post_text" : "writing tutorials on mongodb",
        "tags" : [ "mongodb", "tutorial" ]
})
WriteResult({ "nInserted" : 1 })
```

We will create a text index on post_text field so that we can search inside our posts' text –

```
>db.posts.createIndex({post_text:"text"})
{
        "createdCollectionAutomatically" : true,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Using Text Index

Now that we have created the text index on post_text field, we will search for all the posts having the word **tutorialspoint** in their text.

```
> db.posts.find({$text:{$search:"tutorialspoint"}}).pretty()
{
        "_id" : ObjectId("5dd7ce28f1dd4583e7103fe0"),
        "post_text" : "enjoy the mongodb articles on tutorialspoint",
        "tags" : [
                "mongodb",
                "tutorialspoint"
        ]
}
```

The above command returned the following result documents having the word **tutorialspoint** in their post text –

```
{
  "_id" : ObjectId("53493d14d852429c10000002"),
  "post_text" : "enjoy the mongodb articles on tutorialspoint",
  "tags" : [ "mongodb", "tutorialspoint" ]
}
```

Deleting Text Index

To delete an existing text index, first find the name of index using the following query –

```
>db.posts.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "mydb.posts"
        },
        {
                "v" : 2,
                "key" : {
                        "fts" : "text",
                        "ftsx" : 1
                },
                "name" : "post_text_text",
                "ns" : "mydb.posts",
                "weights" : {
                        "post_text" : 1
                },
                "default_language" : "english",
                "language_override" : "language",
```

```
                "textIndexVersion" : 3
        }
]
>
```

After getting the name of your index from above query, run the following command. Here, **post_text_text** is the name of the index.

```
>db.posts.dropIndex("post_text_text")
{ "nIndexesWas" : 2, "ok" : 1 }
```

MongoDB - Regular Expression

Previous Page

Next Page

Regular Expressions are frequently used in all languages to search for a pattern or word in any string. MongoDB also provides functionality of regular expression for string pattern matching using the **$regex** operator. MongoDB uses PCRE (Perl Compatible Regular Expression) as regular expression language.

Unlike text search, we do not need to do any configuration or command to use regular expressions.

Assume we have inserted a document in a database named **posts** as shown below –

```
> db.posts.insert(
{
  "post_text": "enjoy the mongodb articles on tutorialspoint",
  "tags": [
    "mongodb",
    "tutorialspoint"
  ]
}
WriteResult({ "nInserted" : 1 })
```

Using regex Expression

The following regex query searches for all the posts containing string **tutorialspoint** in it –

```
> db.posts.find({post_text:{$regex:"tutorialspoint"}}).pretty()
{
        "_id" : ObjectId("5dd7ce28f1dd4583e7103fe0"),
        "post_text" : "enjoy the mongodb articles on tutorialspoint",
        "tags" : [
                "mongodb",
                "tutorialspoint"
        ]
}
```

```
{
        "_id" : ObjectId("5dd7d111f1dd4583e7103fe2"),
        "post_text" : "enjoy the mongodb articles on tutorialspoint",
        "tags" : [
                "mongodb",
                "tutorialspoint"
        ]
}
>
```

The same query can also be written as –

```
>db.posts.find({post_text:/tutorialspoint/})
```

Using regex Expression with Case Insensitive

To make the search case insensitive, we use the **$options** parameter with value **$i**. The following command will look for strings having the word **tutorialspoint**, irrespective of smaller or capital case –

```
>db.posts.find({post_text:{$regex:"tutorialspoint",$options:"$i"}})
```

One of the results returned from this query is the following document which contains the word **tutorialspoint** in different cases –

```
{
  "_id" : ObjectId("53493d37d852429c10000004"),
  "post_text" : "hey! this is my post on TutorialsPoint",
  "tags" : [ "tutorialspoint" ]
}
```

Using regex for Array Elements

We can also use the concept of regex on array field. This is particularly very important when we implement the functionality of tags. So, if you want to search for all the posts having tags beginning from the word tutorial (either tutorial or tutorials or tutorialpoint or tutorialphp), you can use the following code –

```
>db.posts.find({tags:{$regex:"tutorial"}})
```

Optimizing Regular Expression Queries

- If the document fields are **indexed**, the query will use make use of indexed values to match the regular expression. This makes the search very fast as compared to the regular expression scanning the whole collection.
- If the regular expression is a **prefix expression**, all the matches are meant to start with a certain string characters. For e.g., if the regex expression is **^tut**, then the query has to search for only those strings that begin with **tut**.

Interview Questions

https://www.interviewbit.com/mongodb-interview-questions/