MONGODB HANDS-ON

After downloading MongoDB community server setup once done, head over to the C drive in which you have installed MongoDB. Go to program files and select the MongoDB directory.

```
C: -> Program Files -> MongoDB -> Server -> 4.0(version) -> bin
```

In the bin directory, you will find an interesting couple of executable files.

- mongod
- mongo

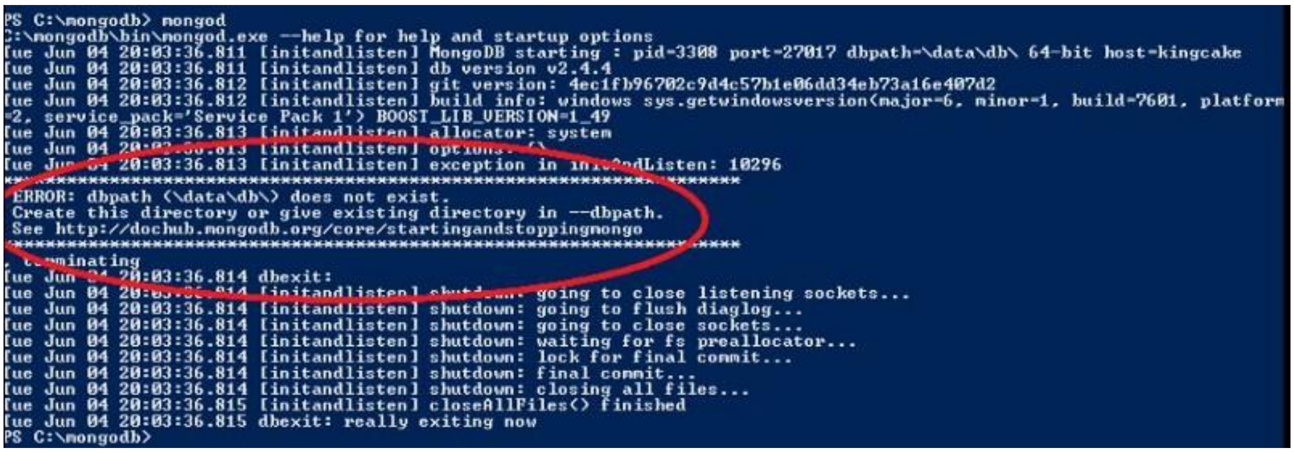
mongod stands for “Mongo Daemon”. mongod is a background process used by MongoDB. The main purpose of mongod is to manage all the MongoDB server tasks. For instance, accepting requests, responding to client, and memory management.

mongo is a command line shell that can interact with the client (for example, system administrators and developers).

Open up your command prompt inside your C drive and do the following:

```
C:\> mkdir data/dbC:\> cd dataC:\> mkdir db
```

The purpose of these directories is MongoDB requires a folder to store all data. MongoDB’s default data directory path is `/data/db` on the drive. Therefore, it is necessary that we provide those directories like so.
If you start the MongoDB server without those directories, you’ll probably see this following error:



```
PS C:\mongodb> mongod
C:\mongodb\bin\mongod.exe --help for help and startup options
Tue Jun 04 20:03:36.811 [initandlisten] MongoDB starting : pid=3308 port=27017 dbpath=\data\db\ 64-bit host=kingcake
Tue Jun 04 20:03:36.811 [initandlisten] db version v2.4.4
Tue Jun 04 20:03:36.812 [initandlisten] git version: 4ec1fb96702c9d4c57b1e06dd34eb73a16e407d2
Tue Jun 04 20:03:36.812 [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform
=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
Tue Jun 04 20:03:36.813 [initandlisten] allocator: system
Tue Jun 04 20:03:36.813 [initandlisten] options: {}
Tue Jun 04 20:03:36.813 [initandlisten] exception in initAndListen: 10296
*********************************************************************
 ERROR: dbpath (\data\db\) does not exist.
 Create this directory or give existing directory in --dbpath.
 See http://dochub.mongodb.org/core/startingandstoppingmongo
*********************************************************************
, terminating
Tue Jun 04 20:03:36.814 dbexit:
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: going to close listening sockets...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: going to flush diaglog...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: going to close sockets...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: waiting for fs preallocator...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: lock for final commit...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: final commit...
Tue Jun 04 20:03:36.814 [initandlisten] shutdown: closing all files...
Tue Jun 04 20:03:36.815 [initandlisten] closeAllFiles() finished
Tue Jun 04 20:03:36.815 dbexit: really exiting now
PS C:\mongodb>
```

Trying to start mongodb server without \data\db directories

After creating those two files, head over again to the bin folder you have in your mongodb directory and open up your shell inside it. Run the following command:

```
mongod
```

Now our MongoDB server is up and running! ?

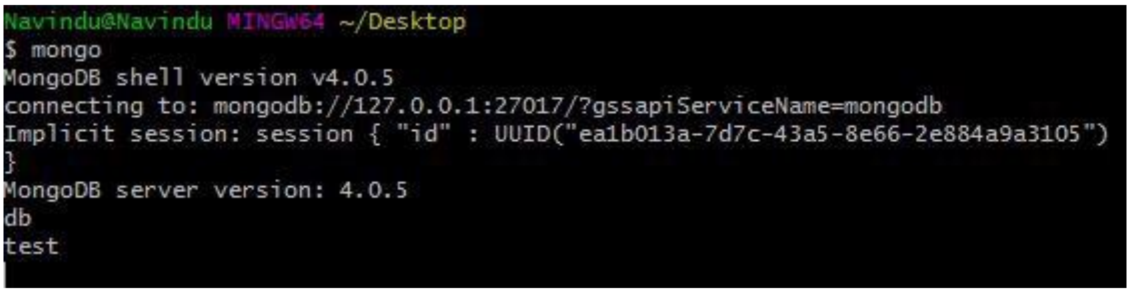
In order to work with this server, we need a mediator. So open another command window inside the bind folder and run the following command:

```
mongo
```

After running this command, navigate to the shell which we ran mongod command (which is our server). You'll see a 'connection accepted' message at the end. That means our installation and configuration is successful!

Just simply run in the mongo shell:

```
db
```



```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("ea1b013a-7d7c-43a5-8e66-2e884a9a3105")
}
MongoDB server version: 4.0.5
db
test
```

Setting up Environment Variables

To save time, you can set up your environment variables. In Windows, this is done by following the menus below:

```
Advanced System Settings -> Environment Variables -> Path(Under System
Variables) -> Edit
```

Simply copy the path of our bin folder and hit OK! In my case it's `C:\Program Files\MongoDB\Server\4.0\bin`
Now you're all set!

Working with MongoDB

There's a bunch of GUIs (Graphical User Interface) to work with MongoDB server such as MongoDB Compass, Studio 3T and so on.

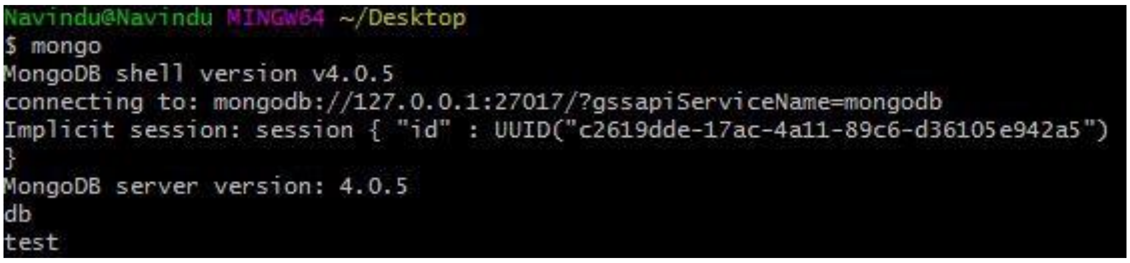
They provide a graphical interface so you can easily work with your database and perform queries instead of using a shell and typing queries manually.

1. Open up your command prompt and type `mongod` to start the MongoDB server.
2.

2. Open up another shell and type `mongo` to connect to MongoDB database server.

1. Finding the current database you're in

```
db
```

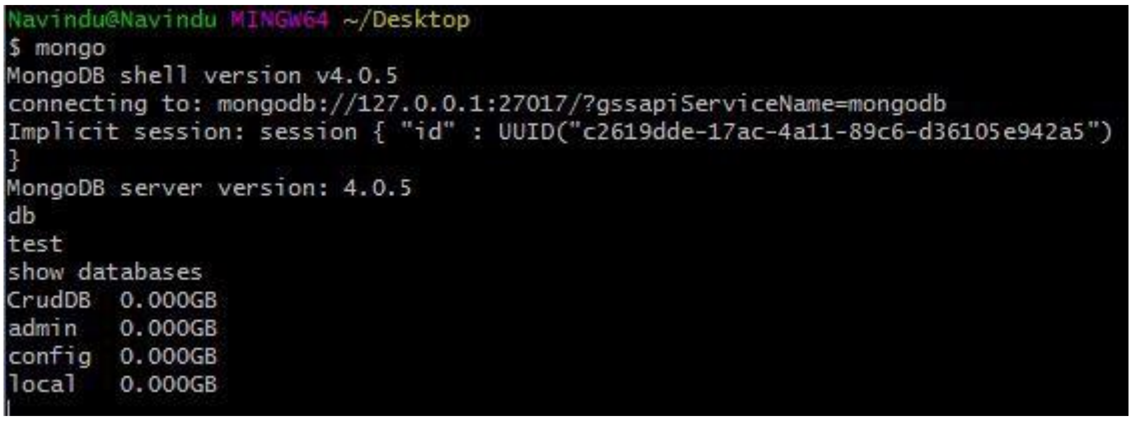


```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c2619dde-17ac-4a11-89c6-d36105e942a5")
}
MongoDB server version: 4.0.5
db
test
```

This command will show the current database you are in. `test` is the initial database that comes by default.

2. Listing databases

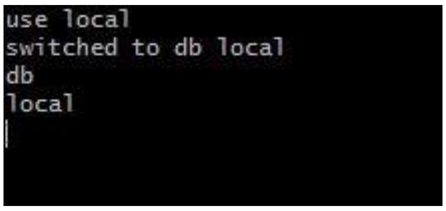
```
show databases
```



```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c2619dde-17ac-4a11-89c6-d36105e942a5")
}
MongoDB server version: 4.0.5
db
test
show databases
CrudDB   0.000GB
admin    0.000GB
config   0.000GB
local    0.000GB
```

3. Go to a particular database

```
use <your_db_name>
```



```
use local
switched to db local
db
local
```

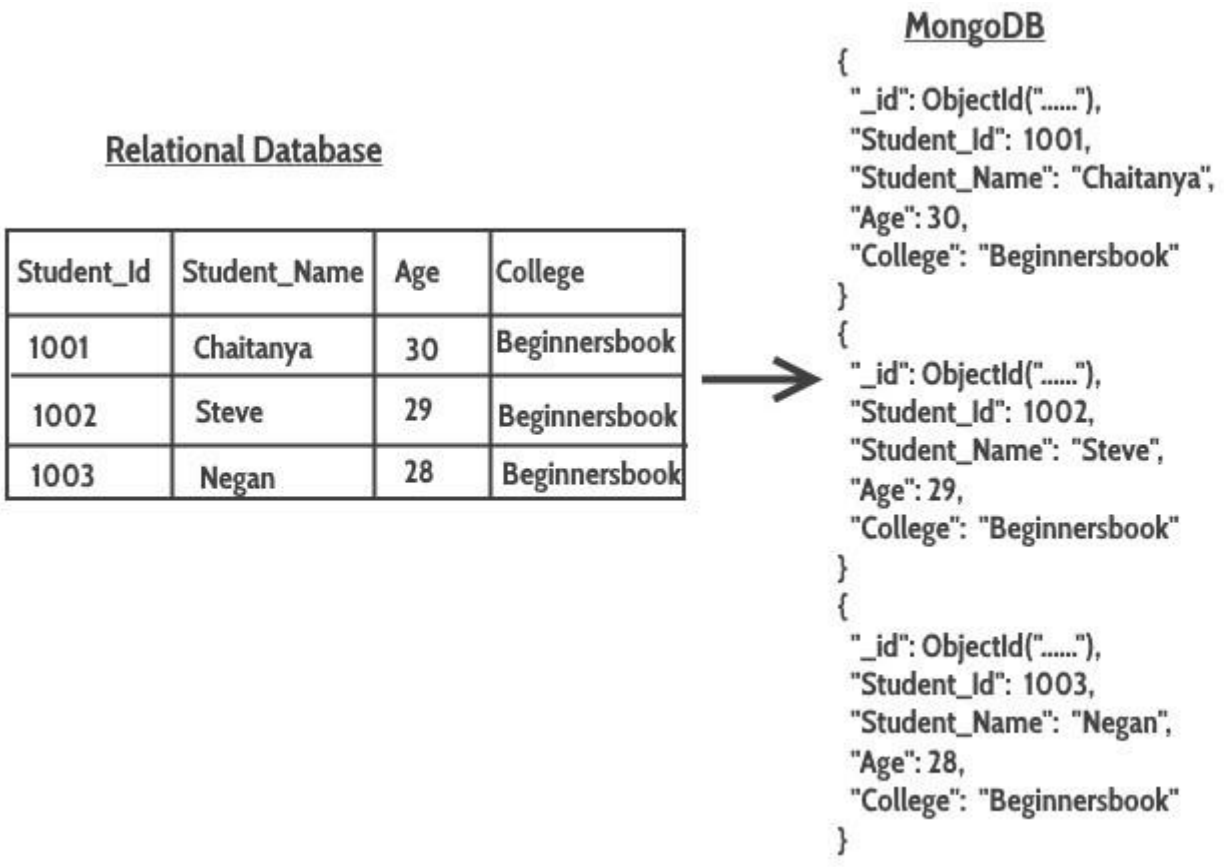
You can check this if you try the command `db` to print out the current database name

4. Creating a Database

With RDBMS (Relational Database Management Systems) we have Databases, Tables, Rows and Columns.

But in NoSQL databases, such as MongoDB, data is stored in BSON format (a binary version of JSON). They are stored in structures called “collections”.

In SQL databases, these are similar to Tables.

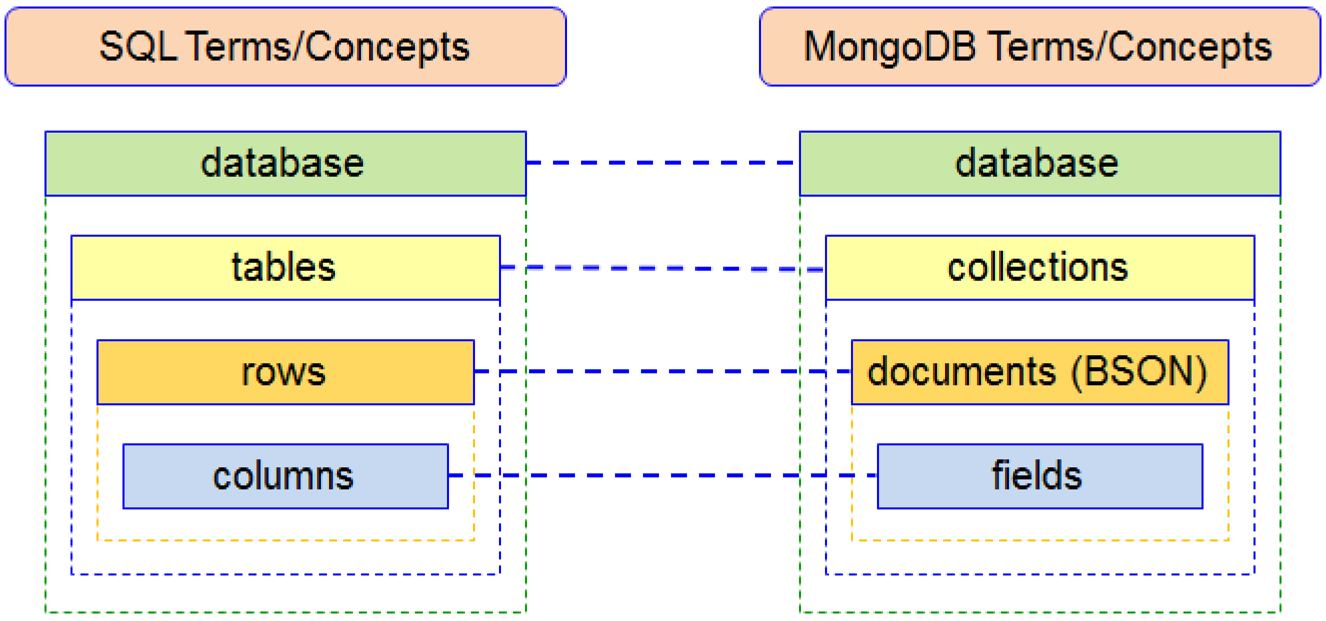


Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

MongoDB

```
{
 "_id": ObjectId("......"),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("......"),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("......"),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```



```
use <your_db_name>
```

In MongoDB server, if your database is present already, using that command will navigate into your database.

But if the database is not present already, then MongoDB server is going to create the database for you. Then, it will navigate into it.

After creating a new database, running the `show database` command will not show your newly created database. This is because, until it has any data (documents) in it, it is not going to show in your db list.

How Are Their Queries Different?

MySQL:

```
SELECT *

FROM customer
```

MongoDB:

```
db.customer.find()
```

Inserting records into the customer table:

MySQL:

```
INSERT INTO customer (cust_id, branch, status)

VALUES ('appl01', 'main', 'A')
```

MongoDB:

```
db.customer.insert({
        cust_id: 'appl01',

        branch: 'main',

        status: 'A'

})
```

Updating records in the customer table:

MySQL:

```
UPDATE customer

SET branch = 'main'

WHERE custage > 2
```

MongoDB:

```
db.customer.update({
      custage: { $gt: 2 }
  },
  {
      $set: { branch:'main' }
  },
  {
      multi: true
})
```

MySQL can be subject to SQL injection attacks, making it vulnerable. Since MongoDB uses object querying, where documents are passed to explain what is being queried, it reduces the risk of attack as MongoDB doesn't have a language to parse.

5. Creating a Collection

Navigate into your newly created database with the `use` command.

Actually, there are two ways to create a collection. One way is to insert data into the collection:

```
db.myCollection.insert({"name": "john", "age" : 22, "location": "colombo"})
```

This is going to create your collection `myCollection` even if the collection does not exist. Then it will insert a document with `name` and `age`. These are non-capped collections.
The second way is shown below:

2.1 Creating a Non-Capped Collection

```
db.createCollection("myCollection")
```

2.2 Creating a Capped Collection

```
db.createCollection("mySecondCollection", {capped : true, size : 2, max : 2})
```

In this way, you’re going to create a collection without inserting data.

The `size : 2` means a limit of two megabytes, and `max: 2` sets the maximum number of documents to two.

6. Inserting Data

We can insert data to a new collection, or to a collection that has been created before.

There are three methods of inserting data.

1. `insertOne()` is used to insert a single document only.
2. `insertMany()` is used to insert more than one document.
3. `insert()` is used to insert documents as many as you want.

Below are some examples:

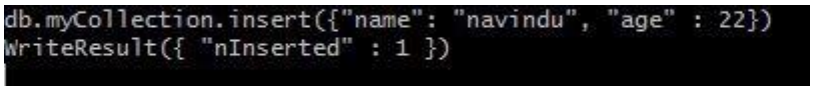
- **insertOne()**

```
db.myCollection.insertOne(
  {
    "name": "navindu",
    "age": 22
  }
)
```

- **insertMany()**

```
db.myCollection.insertMany([
  {
    "name": "navindu",
    "age": 22
  },
  {
    "name": "kavindu",
    "age": 20
  },

  {
    "name": "john doe",
    "age": 25,
    "location": "colombo"
  }
])
```

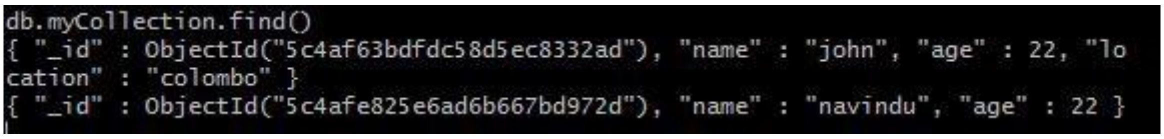
The `insert()` method is similar to the `insertMany()` method.



```
db.myCollection.insert({"name": "navindu", "age" : 22})
WriteResult({ "nInserted" : 1 })
```

7. Querying Data

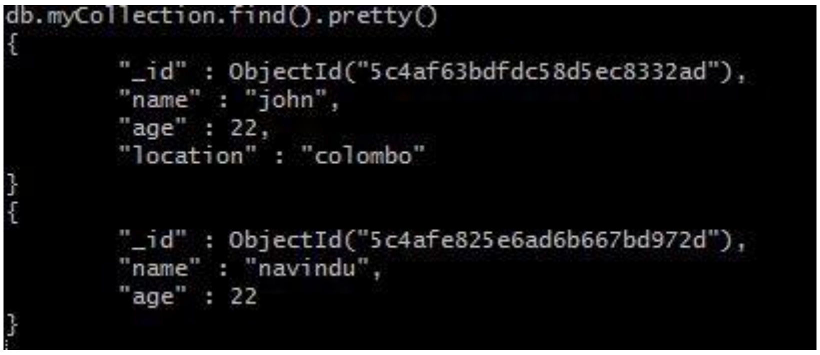
```
db.myCollection.find()
```



```
db.myCollection.find()
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "name" : "john", "age" : 22, "lo
cation" : "colombo" }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "name" : "navindu", "age" : 22 }
```

If you want to see this data in a cleaner, way just add `.pretty()` to the end of it. This will display document in pretty-printed JSON format.

```
db.myCollection.find().pretty()
```



```
db.myCollection.find().pretty()
{
        "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"),
        "name" : "john",
        "age" : 22,
        "location" : "colombo"
}
{
        "_id" : ObjectId("5c4afe825e6ad6b667bd972d"),
        "name" : "navindu",
        "age" : 22
}
```

`_id`?

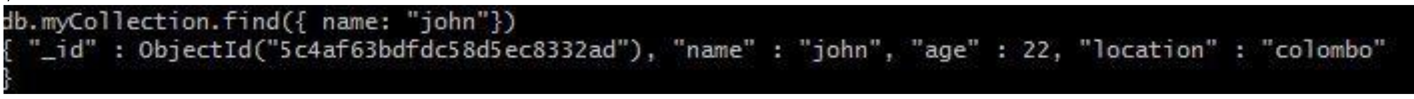
How did that get there?

Well, whenever you insert a document, MongoDB automatically adds an `_id` field which uniquely identifies each document. If you do not want it to display, just simply run the following command

```
db.myCollection.find({}, _id: 0).pretty()
```

If you want to display some specific document, you could specify a single detail of the document which you want to be displayed.

```
db.myCollection.find(
  {
    name: "john"
  }
)
```



```
db.myCollection.find({ name: "john"})
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "name" : "john", "age" : 22, "location" : "colombo"
}
```

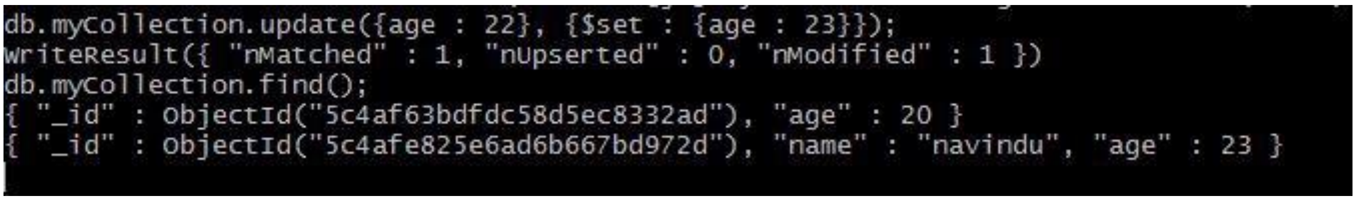
Display people whose age is less than 25. You can use `$lt` to filter for this.

```
db.myCollection.find(
  {
    age : {$lt : 25}
  }
)
```

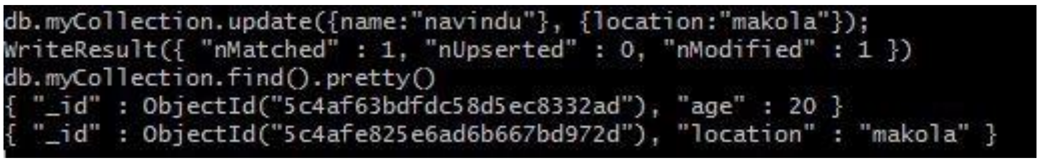
Similarly, `$gt` stands for greater than, `$lte` is “less than or equal to”, `$gte` is “greater than or equal to” and `$ne` is “not equal”.

8. Updating documents

```
db.myCollection.update({age: 20}, {$set: {age: 23}})
```



```
db.myCollection.update({age : 22}, {$set : {age : 23}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.myCollection.find();
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "age" : 20 }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "name" : "navindu", "age" : 23 }
```



```
db.myCollection.update({name:"navindu"}, {location:"makola"});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.myCollection.find().pretty()
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "age" : 20 }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "location" : "makola" }
```

```
// db.myCollection.update({name: "navindu"}, {$unset: age});
```

Removing a document

```
db.myCollection.remove({name: "navindu"});
```

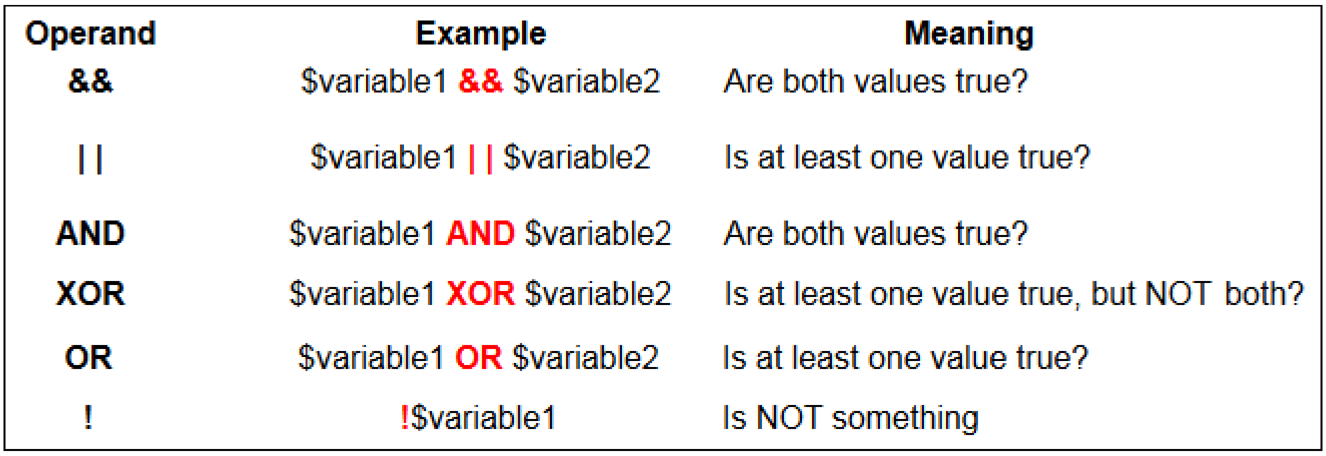
10. Removing a collection

```
// db.myCollection.remove({});
```

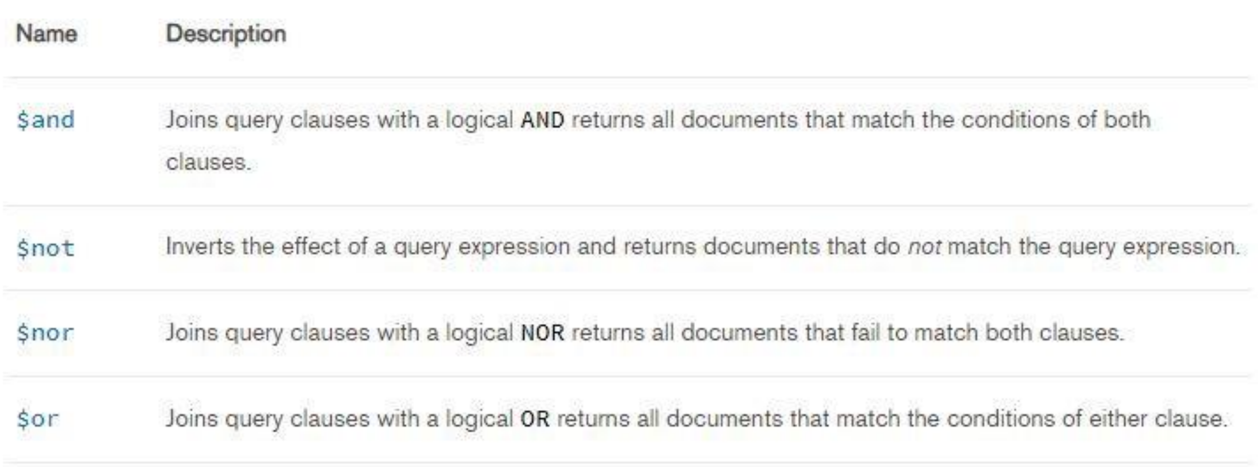
Note, this is not equal to the `drop()` method. The difference is `drop()` is used to remove all the documents inside a collection, but the `remove()` method is used to delete all the documents along with the collection itself.

Logical Operators

MongoDB provides logical operators. The picture below summarizes the different types of logical operators.



Operand	Example	Meaning
&&	$variable1 && $variable2	Are both values true?
||	$variable1 || $variable2	Is at least one value true?
AND	$variable1 AND $variable2	Are both values true?
XOR	$variable1 XOR $variable2	Is at least one value true, but NOT both?
OR	$variable1 OR $variable2	Is at least one value true?
!	!$variable1	Is NOT something



Name	Description
`$and`	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
`$nor`	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
`$or`	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

Display people whose age is less than 25, and also whose location is Colombo. What we could do?

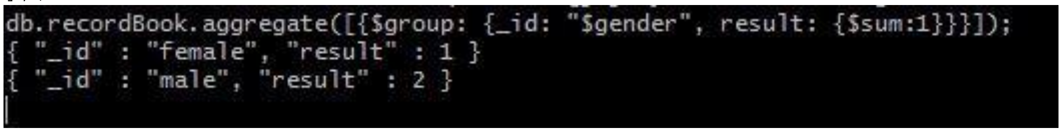
We can use the `$and` operator!

```
db.myCollection.find({$and:[{age : {$lt : 25}}, {location: "colombo"}]});
```

Aggregation

Imagine if we had male and female students in a `recordBook` collection and we want a total count on each of them. In order to get the sum of males and females, we could use the `$group` aggregate function.

```
db.recordBook.aggregate([
  {
    $group : {_id : "$gender", result: {$sum: 1}}
  }
]);
```



```
db.recordBook.aggregate([{$group: {_id: "$gender", result: {$sum:1}}}]);
{ "_id" : "female", "result" : 1 }
{ "_id" : "male", "result" : 2 }
```

```
use LVCDB
db
db.LVCCollection.insert({"RegNo":"R003", "Name":"Anuj Singh",
"Location":"Bangalore", "Course":"Full Stack", "Mobile":443333333})

db.LVCCollection.insertMany( [
   {"RegNo":"R004", "Name":"Komal Singh", "Location":"Madurai", "Course":"Java",
"Mobile":12345432},
   {"RegNo":"R005", "Name":"Amit Singh", "Location":"Shimla", "Course":"Full Stack",
"Mobile":34567898},
   {"RegNo":"R006", "Name":"Reshma Singh", "Location":"UP", "Course":"Full Stack",
"Mobile":8766544332}
]);

show collections
show databases
db.LVCCollection.find( {} )
```

SHOW ALL RECORDS FROM COLLECTION

```
db.LVCCollection.find( {} )
```

```
db.LVCCollection.update({"RegNo": "R001"}, {$set: {"RegNo": "R009"}})
db.LVCCollection.find()
```

```
db.LVCCollection.aggregate([
  {
    $group : {_id : "RegNo", result: {$sum: 1}}
  }
]);
```

```
use LVCDB
db

db.LVCCollection.insert({"RegNo":"R009", "Name":"Kamal Singh",
"Location":"Bangalore", "Course":"Full Stack", "Mobile":54333333})

db.LVCCollection.insertMany( [
   {"RegNo":"R004", "Name":"Komal Singh", "Location":"Madurai", "Course":"Java",
"Mobile":12345432},
   {"RegNo":"R005", "Name":"Amit Singh", "Location":"Shimla", "Course":"Full Stack",
"Mobile":34567898},
   {"RegNo":"R006", "Name":"Reshma Singh", "Location":"UP", "Course":"Full Stack",
"Mobile":8766544332}
]);

db.createCollection("myCollection")
db.myCollection.insert({"RegNo":"R003", "Name":"Anuj Singh", "Location":"Bangalore",
"Course":"Full Stack", "Mobile":443333333})
db.myCollection.find( {} )


db.createCollection("mySecondCollection", {capped : true, size : 2, max : 3})
db.mySecondCollection.insert({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})
db.mySecondCollection.find( {} )
db.createCollection("myThirdCollection", {capped : true, size : 2, max : 4})
db.myThirdCollection.insert({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})
db.myThirdCollection.insertone({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})

db.myThirdCollection.find( {} )

show collections
show databases
db.LVCCollection.find( {} )

db.myCollection5.insertOne(
  {
    "name": "navindu",
    "age": 22
  }
)

db.myCollection5.find({})
db.myCollection.find({})
db.myCollection.find()

db

db.LVCCollection.find()
db.LVCCollection.find().pretty()
db.LVCCollection.find({}, _id: 0).pretty()
db.LVCCollection.find() {"_id":ObjectId("5f5baad8572a1ad446f3060c")}

db.LVCCollection.find(
  {
    age : {$lt:12345678}
  }
)


db.LVCCollection.update({"RegNo": "R001"}, {$set: {"RegNo": "R009"}})
db.LVCCollection.find()

db.LVCCollection.remove({Name:"Raj Kumar"});

db.mySecondCollection.remove({});

db.mySecondCollection.find({$and:[{Mobile : {$lt : 25}}, {location: "Mumbai"}]});

db.LVCCollection.aggregate([
  {
    $group : {_id : "RegNo", result: {$sum: 1}}
  }
]);
```

```
db.inventory.updateOne(
  { item: "paper" },
  {
    $set: { "size.uom": "cm", status: "P" },
    $currentDate: { lastModified: true }
  }
)
```

`db.collection.update()`	Either updates or replaces a single document that match a specified filter or updates all documents that match a specified filter. By default, the `db.collection.update()` method updates a **single** document. To update multiple documents, use the multi option.

Update a Single Document

The following example uses the `db.collection.updateOne()` method on the `inventory` collection to update the *first* document where `item` equals `"paper"`:

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" }
   }
)
```

The following example uses the `db.collection.updateMany()` method on the `inventory` collection to update all documents where `qty` is less than `50`:

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" }

   }
)
```

- db.collection.deleteMany()
- db.collection.deleteOne()

Delete All Documents

To delete all documents from a collection, pass an empty filter document `{}` to the `db.collection.deleteMany()` method.

The following example deletes *all* documents from the `inventory` collection:

```
db.inventory.deleteMany({})
```

Delete Only One Document that Matches a Condition

To delete at most a single document that matches a specified filter (even though multiple documents may match the specified filter) use the `db.collection.deleteOne()` method.

The following example deletes the *first* document where `status` is `"D"`:

```
db.inventory.deleteOne( { status: "D" } )
```

INDEX IN MONGODB

Indexes are very important in any database, and with MongoDB it's no different. With the use of Indexes, performing queries in MongoDB becomes more efficient.

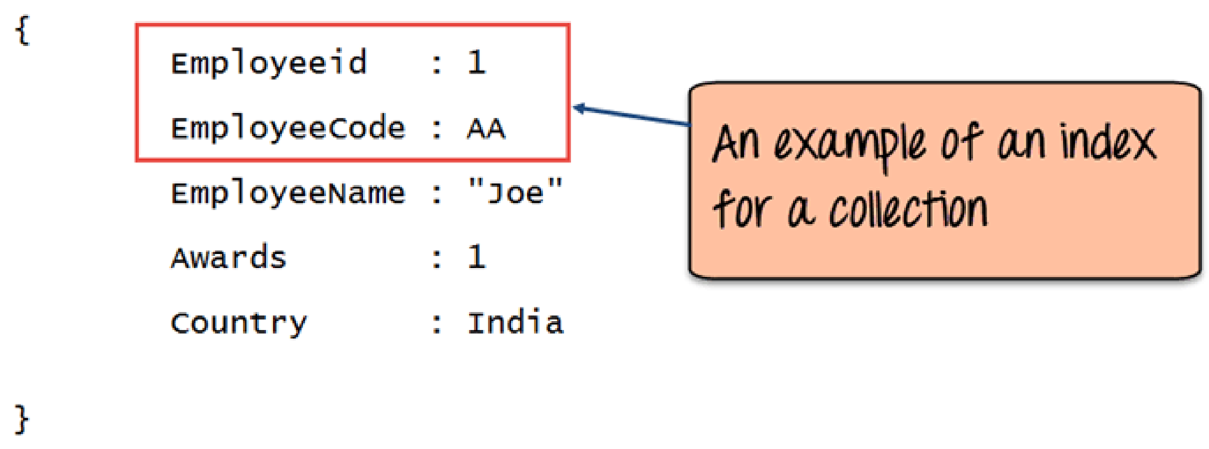
If you had a collection with thousands of documents with no indexes, and then you query to find certain documents, then in such case MongoDB would need to scan the entire collection to find the documents. But if you had indexes, MongoDB would use these indexes to limit the number of documents that had to be searched in the collection.

Indexes are special data sets which store a partial part of the collection's data. Since the data is partial, it becomes easier to read this data. This partial set stores the value of a specific field or a set of fields ordered by the value of the field.

Indexes are good for queries, but having too many indexes can slow down other operations such as the Insert, Delete and Update operation.

If there are frequent insert, delete and update operations carried out on documents, then the indexes would need to change that often, which would just be an overhead for the collection.

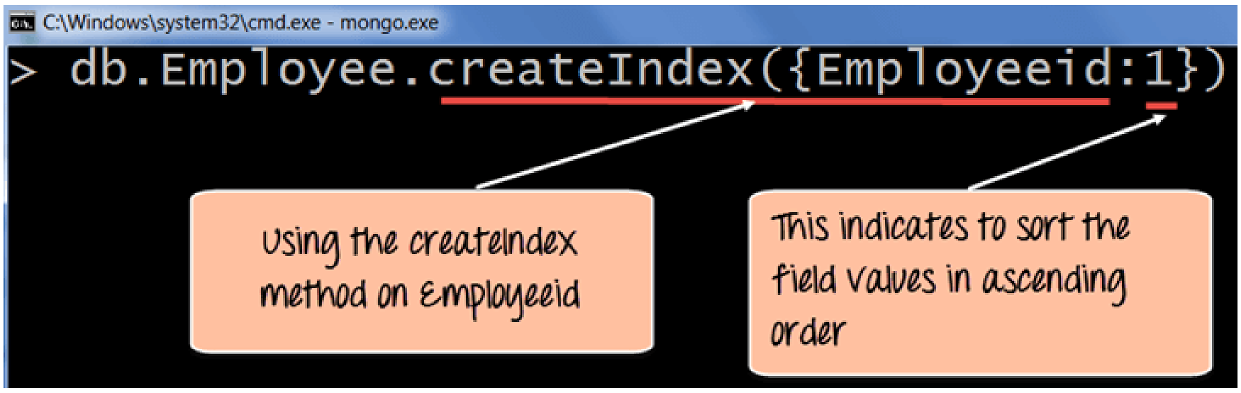
The below example shows an example of what field values could constitute an index in a collection. An index can either be based on just one field in the collection, or it can be based on multiple fields in the collection.



```
{
        Employeeid    : 1
        EmployeeCode  : AA
        EmployeeName  : "Joe"
        Awards        : 1
        Country       : India
}
```

How to Create Indexes: createIndex()

Creating an Index in MongoDB is done by using the "**createIndex**" method.



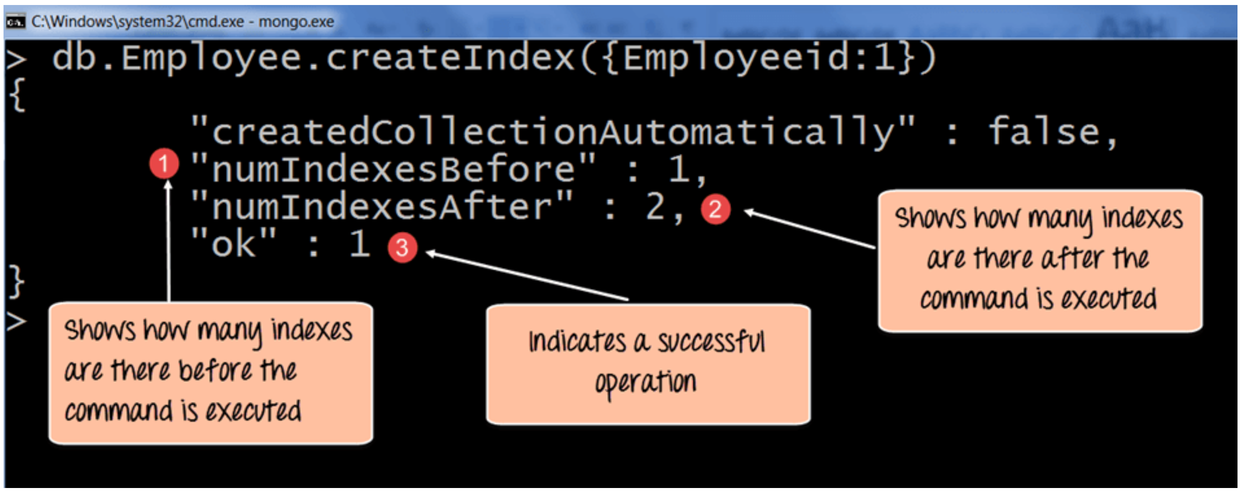
```
> db.Employee.createIndex({Employeeid:1})
```

C:\Windows\system32\cmd.exe - mongo.exe
Using the createIndex method on Employeeid
This indicates to sort the field values in ascending order

```
db.Employee.createIndex({Employeeid:1})
```

1. The **createIndex** method is used to create an index based on the "Employeeid" of the document.
2. The '1' parameter indicates that when the index is created with the "Employeeid" Field values, they should be sorted in ascending order. Please note that this is different from the _id field (The id field is used to uniquely identify each document in the collection) which is created automatically in the collection by MongoDB. The documents will now be sorted as per the Employeeid and not the _id field.

3. If the command is executed successfully, the following Output will be shown:
4. **Output:**



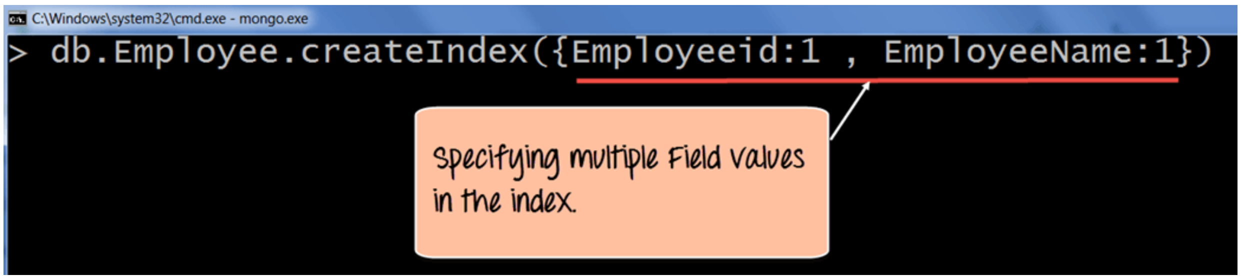
```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.createIndex({Employeeid:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

1
2
3
Shows how many indexes are there after the command is executed
Indicates a successful operation
Shows how many indexes are there before the command is executed

1. The numIndexesBefore: 1 indicates the number of Field values (The actual fields in the collection) which were there in the indexes before the command was run. Remember that each collection has the _id field which also counts as a Field value to the index. Since the _id index field is part of the collection when it is initially created, the value of numIndexesBefore is 1.
2. The numIndexesAfter: 2 indicates the number of Field values which were there in the indexes after the command was run.
3. Here the "ok: 1" output specifies that the operation was successful, and the new index is added to the collection.

The above code shows how to create an index based on one field value, but one can also create an index based on multiple field values.

The following example shows how this can be done;



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.createIndex({Employeeid:1 , EmployeeName:1})
```

```
db.Employee.createIndex({Employeeid:1, EmployeeName:1])
```

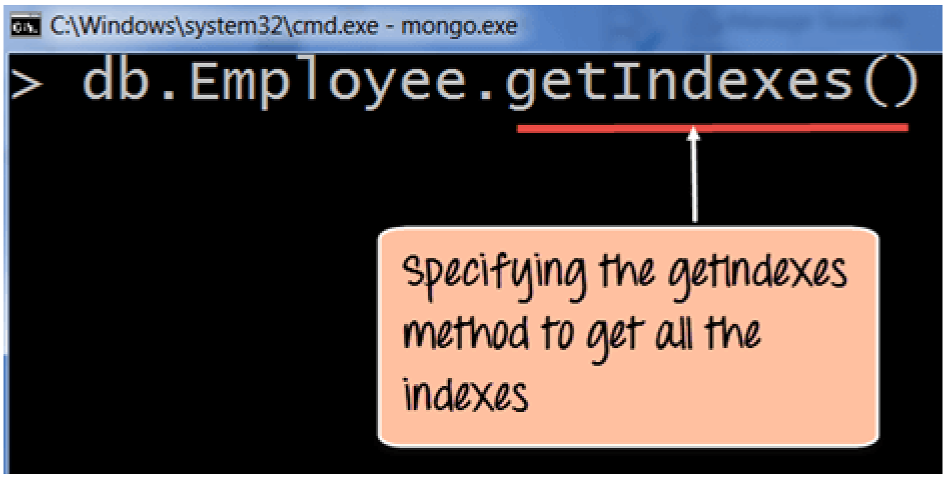
Code Explanation:

1. The createIndex method now takes into account multiple Field values which will now cause the index to be created based on the "Employeeid" and "EmployeeName". The Employeeid:1 and EmployeeName:1 indicates that the index should be created on these 2 field values with the :1 indicating that it should be in ascending order.

How to Find Indexes: getindexes()

Finding an Index in MongoDB is done by using the **"getIndexes"** method.

The following example shows how this can be done;


C:\Windows\system32\cmd.exe - mongo.exe
Specifying the getIndexes method to get all the indexes

```
> db.Employee.getIndexes()
```

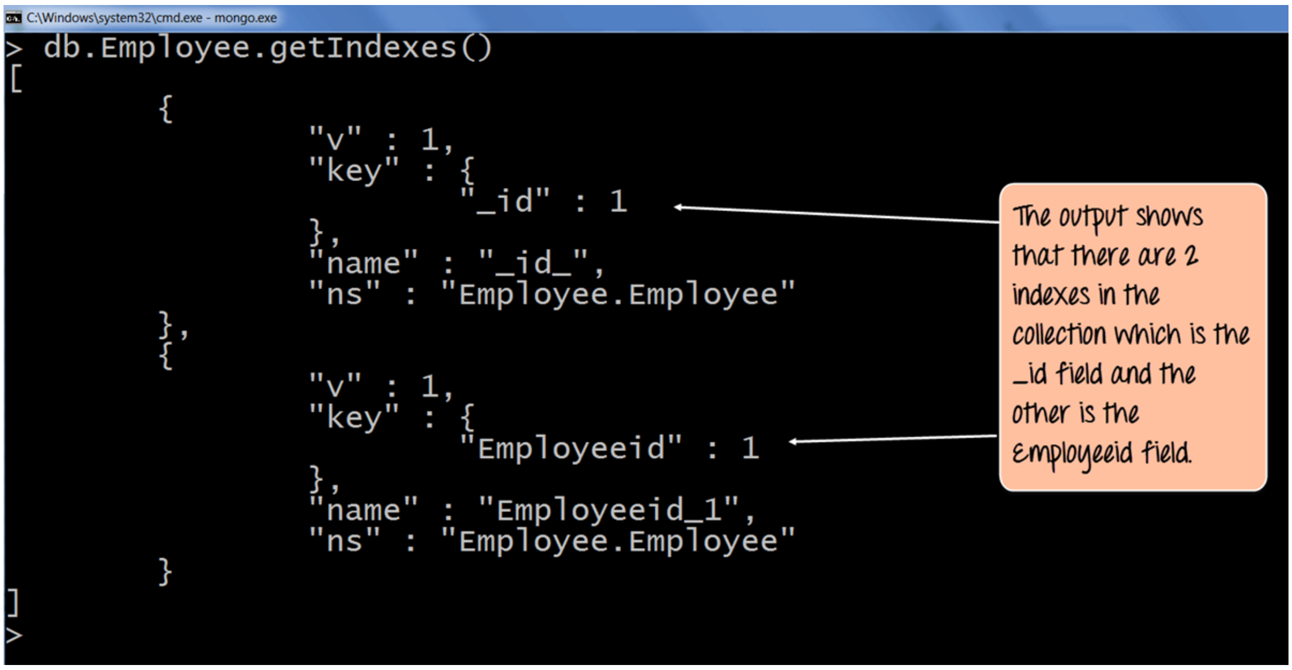
```
db.Employee.getIndexes()
```

Code Explanation:

1. The getIndexes method is used to find all of the indexes in a collection.

If the command is executed successfully, the following Output will be shown:

Output:



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "Employee.Employee"
        },
        {
                "v" : 1,
                "key" : {
                        "Employeeid" : 1
                },
                "name" : "Employeeid_1",
                "ns" : "Employee.Employee"
        }
]
>
```

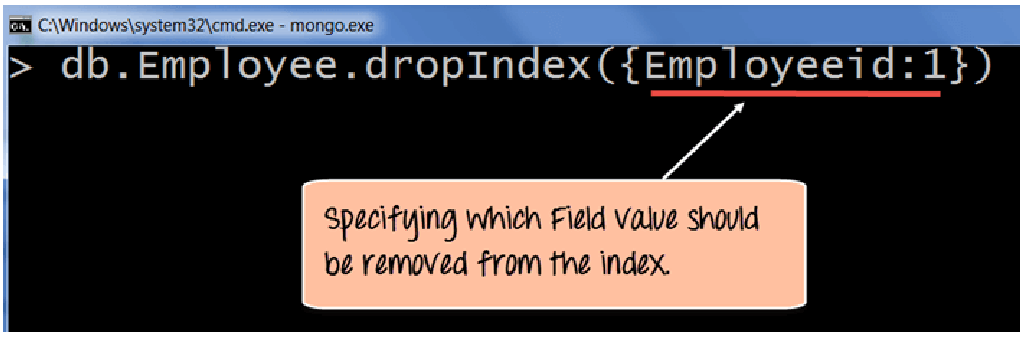
The output shows that there are 2 indexes in the collection which is the _id field and the other is the Employeeid field.

1. The output returns a document which just shows that there are 2 indexes in the collection which is the _id field, and the other is the Employee id field. The :1 indicates that the field values in the index are created in ascending order.

How to Drop Indexes: dropindex()

Removing an Index in MongoDB is done by using the dropIndex method.

The following example shows how this can be done;



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.dropIndex({Employeeid:1})
```

Specifying which Field value should be removed from the index.

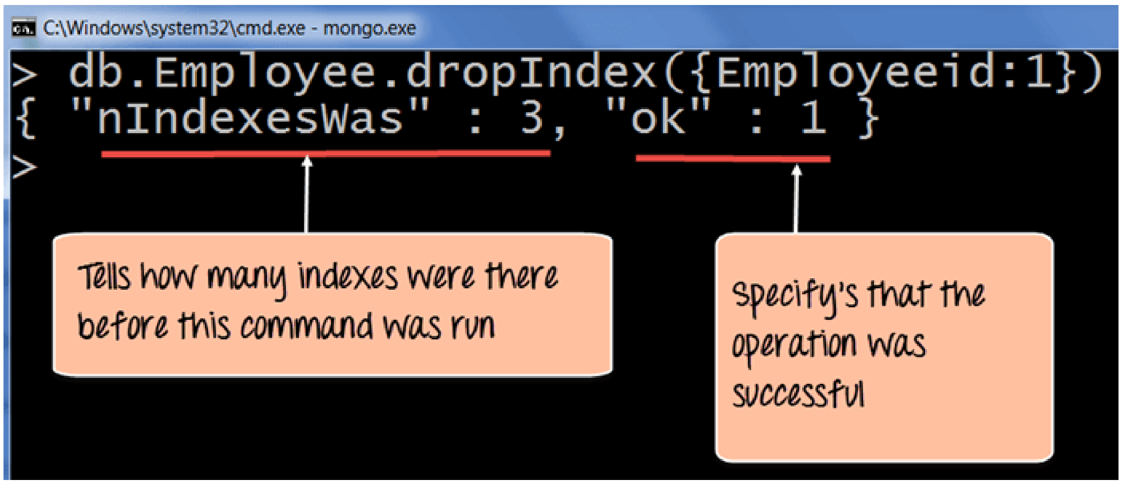
```
db.Employee.dropIndex(Employeeid:1)
```

Code Explanation:

1. The dropIndex method takes the required Field values which needs to be removed from the Index.

If the command is executed successfully, the following Output will be shown:

Output:



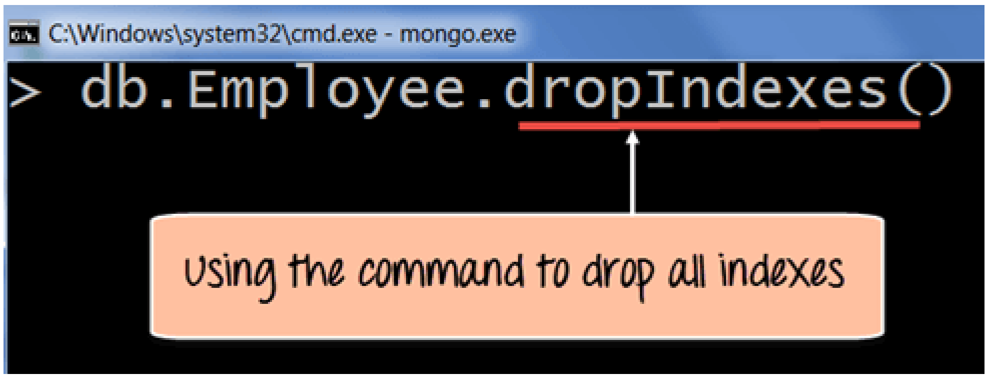
```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.dropIndex({Employeeid:1})
{ "nIndexesWas" : 3, "ok" : 1 }
>
```

Tells how many indexes were there before this command was run
Specify's that the operation was successful

1. The nIndexesWas: 3 indicates the number of Field values which were there in the indexes before the command was run. Remember that each collection has the _id field which also counts as a Field value to the index.
2. The ok: 1 output specifies that the operation was successful, and the "Employeeid" field is removed from the index.

To remove all of the indexes at once in the collection, one can use the dropIndexes command.

The following example shows how this can be done.



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.dropIndexes()
```

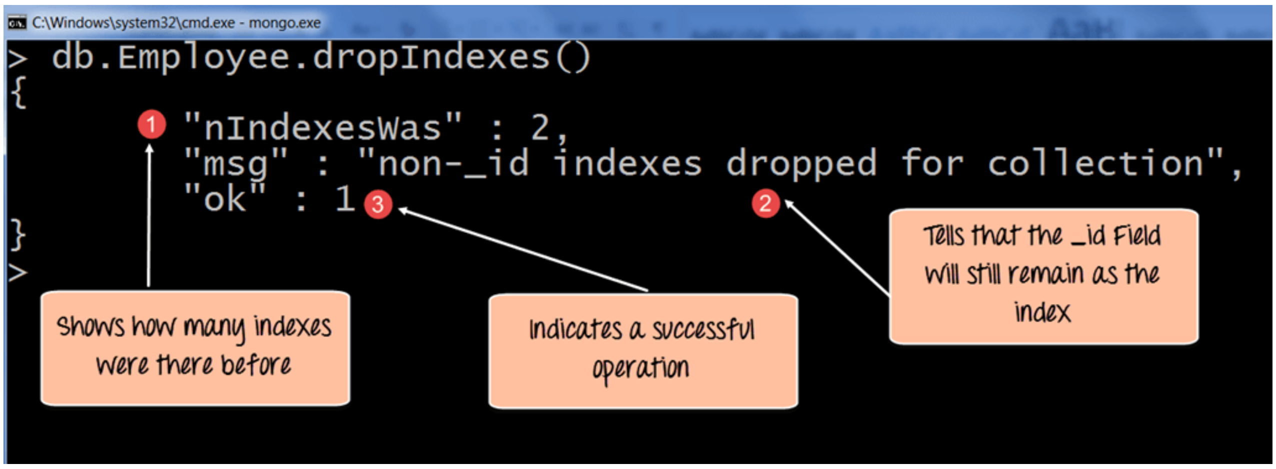
```
db.Employee.dropIndex()
```

Code Explanation:

1. The dropIndexes method will drop all of the indexes except for the _id index.

If the command is executed successfully, the following Output will be shown:

Output:



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.dropIndexes()
{
        "nIndexesWas" : 2,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
>
```

1
2
3
Shows how many indexes were there before
Indicates a successful operation
Tells that the _id Field Will still remain as the index

1. The nIndexesWas: 2 indicates the number of Field values which were there in the indexes before the command was run.
2. Remember again that each collection has the _id field which also counts as a Field value to the index, and that will not be removed by MongoDB and that is what this message indicates.
3. The ok: 1 output specifies that the operation was successful.

Summary

- Defining indexes are important for faster and efficient searching of documents in a collection.
- Indexes can be created by using the createIndex method. Indexes can be created on just one field or multiple field values.
- Indexes can be found by using the getIndexes method.
- Indexes can be removed by using the dropIndex for single indexes or dropIndexes for dropping all indexes.

Document Field Order

MongoDB preserves the order of the document fields following write operations *except* for the following cases:

- The `_id` field is always the first field in the document.
- Updates that include `renaming` of field names may result in the reordering of fields in the document.

The `_id` Field

In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the `_id` field, the MongoDB driver automatically generates an ObjectId for the `_id` field.

This also applies to documents inserted through update operations with upsert: true.

The `_id` field has the following behavior and constraints:

- By default, MongoDB creates a unique index on the `_id` field during the creation of a collection.
- The `_id` field is always the first field in the documents. If the server receives a document that does not have the `_id` field first, then the server will move the field to the beginning.
- The `_id` field may contain values of any BSON data type, other than an array.

PRACTICAL :

Use mydatabase

```
for (i=0; i<10000; ++i)
{
db.mypost.insert({"student_id" : 1, "Name" : "Raj"});
}
```

Open the Mongo Shell to check this

Use mydatabase

db.mypost.find()   // Show All records

db.mypost.find({"student_id" : 1000});

Observer the retrieval speed without Index

Make it faster :

Db.mypost.findOne{“student_id” : 1000});

NOW make more Faster :

Db.mypost.ensureIndex({“student_id” : 1});

Observer the retrieval speed with Index

Db.mypost.findOne{“student_id” : 1000});

// REMOVE INDEX

Db.mypost.dropIndex({“student_id” : 1});

Db.mypost.find({“student_id” : 1000});

Serializing and de-serializing data

In the context of data storage, **serialization** (or serialisation) is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later

SON is a format that encodes objects in a string. **Serialization means to convert an object into that string**, and **deserialization is its inverse operation (convert string -> object)**.

The opposite operation, extracting a data structure from a series of bytes, is **deserialization**

This process of serializing an object is also called marshalling an object in some situations.The opposite operation, extracting a data structure from a series of bytes, is **deserialization**, (also called **unserialization** or **unmarshalling**).

EXAMPLE

Say, you have an object:

```
{foo: [1, 4, 7, 10], bar: "baz"}
```

serializing into JSON will convert it into a string:

```
'{"foo":[1,4,7,10],"bar":"baz"}'
```

which can be stored or sent through wire to anywhere. The receiver can then deserialize this string to get back the original object. `{foo: [1, 4, 7, 10], bar: "baz"}`.

JSON AND BSON

BSON is just binary **JSON** (a superset of **JSON** with some more data types, most importantly binary byte array). It is a serialization format used in MongoDB.

JSON data contains its data basic in **JSON** format.

BSON gives extra datatypes over the **JSON** data.

Since its initial formulation, BSON has been extended to add some optional non-JSON-native data types, like dates and binary data, without which MongoDB would have been missing some valuable support.

Does MongoDB use BSON, or JSON?

MongoDB stores data in BSON format both internally, and over the network, but that doesn't mean you can't think of MongoDB as a JSON database. Anything you can represent in JSON can be natively stored in MongoDB, and retrieved just as easily in JSON.

Data Support :

Json :

String, Boolean, Number, Array

BSON

String, Boolean, Number (Integer, Float, Long, Decimal128...), Array, Date, Raw Binary

JSON vs BSON

#1. Type

JSON	BSON
Standard file format.	Binary file format

#2. Speed

JSON	BSON
Comparatively less fast.	Faster.

#3. Space

JSON	BSON
Consumes comparatively less space.	More space is consumed.

#4. Usage

JSON	BSON
Transmission of data.	Storage of data.

#5. Encoding and Decoding Technique

JSON	BSON
No such technique.	Faster encoding and decoding technique.

#6. Characteristics

JSON	BSON
Key value pair only used for transmission of data.	Lightweight, fast and traversable.

#7. Structure

JSON BSON

MONGODB SHELL

The MongoDB Shell, `mongosh`, is a fully functional JavaScript environment for interacting with MongoDB deployments. You can use the MongoDB Shell to test queries and operations directly with your database.

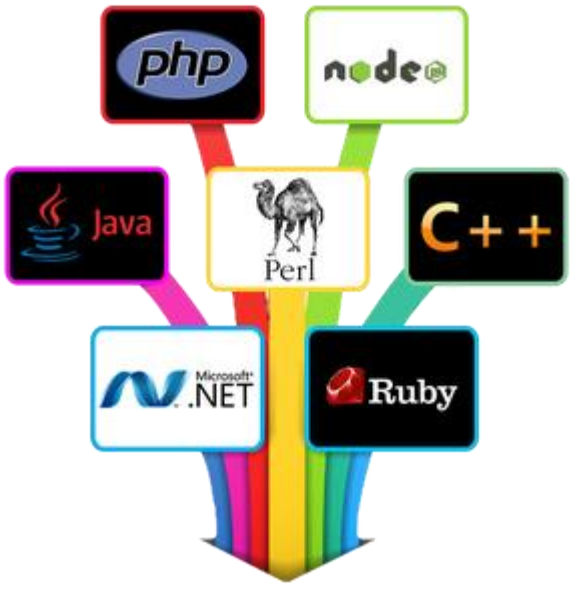
Install MongoDB Shell

https://docs.mongodb.com/mongodb-shell/install#mdb-shell-install

MongoDB Integration with Other languages

MongoDB is an open-source database, which is reliable, globally scalable and inexpensive to operate. MongoDB provides native drivers for all popular programming languages and frameworks to make development natural.

A list of supported drivers include:
Java, .NET, Ruby, PHP, JavaScript, node.js, Python, Perl, PHP, Scala etc.



An application communicates with MongoDB by way of a client library, called a driver is used for interacting with MongoDB in a particular language.
The popular MongoDB drivers and client libraries include:

PHPMoAdmin is MongoDB Administration tool for PHP built on a stripped down version of the Vork high-performance framework. It can help you discover the source of connection issues between PHP and MongoDB.
Mongobee is a java tool which can help you manage changes in your MongoDB database and keep them synchronized with your java application. It provides new approach for adding changes based on Java classes and methods with appropriate annotations.
PyMongo is a Python distribution containing tools for working with MongoDB.
MongoVUE is a .NET Graphical User Interface that gives an elegant and highly usable interface for working with MongoDB.
Node.js Driver is the officially supported node.js driver for MongoDB. It is written in pure JavaScript and provides a native asynchronous Node.js interface to MongoDB.
MongoDB also supports some Third-party GUI tools like: Robomongo, Fang of Mongo, Mongo3, UMongo etc.

Robomongo is a shell-centric cross-platform open source MongoDB management tool. It embeds the same JavaScript engine that powers MongoDB's mongo shells. Everything you can write in mongo shell, you can write in Robomongo. Robomongo provides you with syntax highlighting, auto-completion, different view modes (text, tree) & more. It has its own features like multiple shells and multiple results.
Fang of Mongo is a web-based User Interface built with Django and jQuery.
UMongo is a cross-platfrom Management-GUI, implemented in java.

MongoDB can be used with many programming languages and development environments as we have discussed above. With MongoDB, you can build applications that were never possible with traditional relational databases, as with the advanced features it provides an extensive driver support with the programming languages.

MONGODB CAPED COLLECTION

We can create **collections** in mongoDb on which we can apply size limit. These special **type** of **collections** are called **Capped Collections**. These are a kind of circular queues, in which if allocated size limit is reached, it makes space for new documents by overwriting the oldest documents in the **collection**.

FULL TEXT SEARCH IN MONGODB

MongoDB, one of the leading NoSQL databases, is well known for its fast performance, flexible schema, scalability and great indexing capabilities. At the core of this fast performance lies MongoDB indexes, which support efficient execution of queries by avoiding full-collection scans and hence limiting the number of documents MongoDB searches.

Introducing MongoDB Text Search

Using MongoDB full-text search, you can define a text index on any field in the document whose value is a string or an array of strings. When we create a text index on a field, MongoDB tokenizes and stems the indexed field's text content, and sets up the indexes accordingly.

Text Score

The `$text` operator assigns a score to each document that contains the search term in the indexed fields. The score represents the relevance of a document to a given text search query. The score can be part of a `sort()` method specification as well as part of the projection expression. The `{ $meta: "textScore" }` expression provides information on the processing of the `$text` operation. See `$meta` projection operator for details on accessing the score for projection or sort.

Examples

The following examples assume a collection `articles` that has a version 3 text index on the field `subject`:

```
db.articles.createIndex( { subject: "text" } )
```

Populate the collection with the following documents:

```
db.articles.insert(
   [
     { _id: 1, subject: "coffee", author: "xyz", views: 50 },
     { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
     { _id: 3, subject: "Baking a cake", author: "abc", views: 90  },
     { _id: 4, subject: "baking", author: "xyz", views: 100 },
     { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
     { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
     { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
     { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }
   ]
)
```

Search for a Single Word

The following query specifies a `$search` string of `coffee`:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

This query returns the documents that contain the term `coffee` in the indexed `subject` field, or more precisely, the stemmed version of the word:

copy

copied

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

INDEX LIMITATIONS:

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.

$explain

It provides information on the query, indexes used in the query and some statistics. It is good to use this if you want to know how well your indexes are optimized.

Now we will take an example to understand about this query. We will use the collection named examples.

```
{
"_id": ObjectId("53402597d852426020000002"),
"contact": "1234567809",
"dob": "01-01-1991",
"gender": "M",
"name": "ABC",
"user_name": "abcuser"
}
>db.examples.ensureIndex({gender:1,user_name:1})
```

Now we will use $explain query over it.

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

After executing the above line we will get the following output:

```
{
"cursor" : "BtreeCursor gender_1_user_name_1",
"isMultiKey" : false,
"n" : 1,
"nscannedObjects" : 0,
"nscanned" : 1,
"nscannedObjectsAllPlans" : 0,
"nscannedAllPlans" : 1,
"scanAndOrder" : false,
"indexOnly" : true,
"nYields" : 0,
"nChunkSkips" : 0,
"millis" : 0,
"indexBounds" : {
"gender" : [
[
"M",
"M"
]
],
"user_name" : [
[
{
"$minElement" : 1
},
{
"$maxElement" : 1
}
]
]
}
}
```

Database Profiler in MongoDB

The database profiler collects detailed information about Database Commands executed against a running mongod instance. This includes CRUD operations as well as configuration and administration commands. The profiler writes all the data it collects to the system.profile collection, a capped collection in the admin database

To enable profiling and set the profiling level, pass the profiling level to the `db.setProfilingLevel()` helper. For example, to enable profiling for all database operations, consider the following operation in the `mongo` shell:

```
db.setProfilingLevel(2)
```

Disable Profiling

To disable profiling, use the following helper in the `mongo` shell:

```
db.setProfilingLevel(0)
```

MongoDB Covered Query

The MongoDB covered query is one which uses an **index** and does not have to examine any documents. An index will cover a query if it satisfies the following conditions:

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

If we take an example of a **collection** named example which has 2 indexes type and item.

```
db.example.createIndex( { type: 1, item: 1 } )
```

This index will cover queries with operations on type and item fields and then return only item field.

```
db.example.find(
{ type: "game", item:/^c/ },
{ item: 1, _id: 0 }
)
```

Here, to cover the query, the **document** must explicitly specify _id: 0 to exclude the _id field from a result. However, this has changed in version 3.6. Over here an index can cover a query on fields within embedded documents.

MongoDB is a cross-platform document-oriented and a non relational (i.e NoSQL) database program. It is an open-source document database, that stores the data in the form of key-value pairs.

What is a MongoDB Query?

MongoDB query is used to specify the selection filter using query operators while retrieving the data from the collection by `db.find()` method. We can easily filter the documents using the query object. To apply the filter on the collection, we can pass the query specifying the condition for the required documents as a parameter to this method, which is an optional parameter for `db.find()` method.
Query Selectors:
Following is the list of some operators used in the queries in MongoDB.

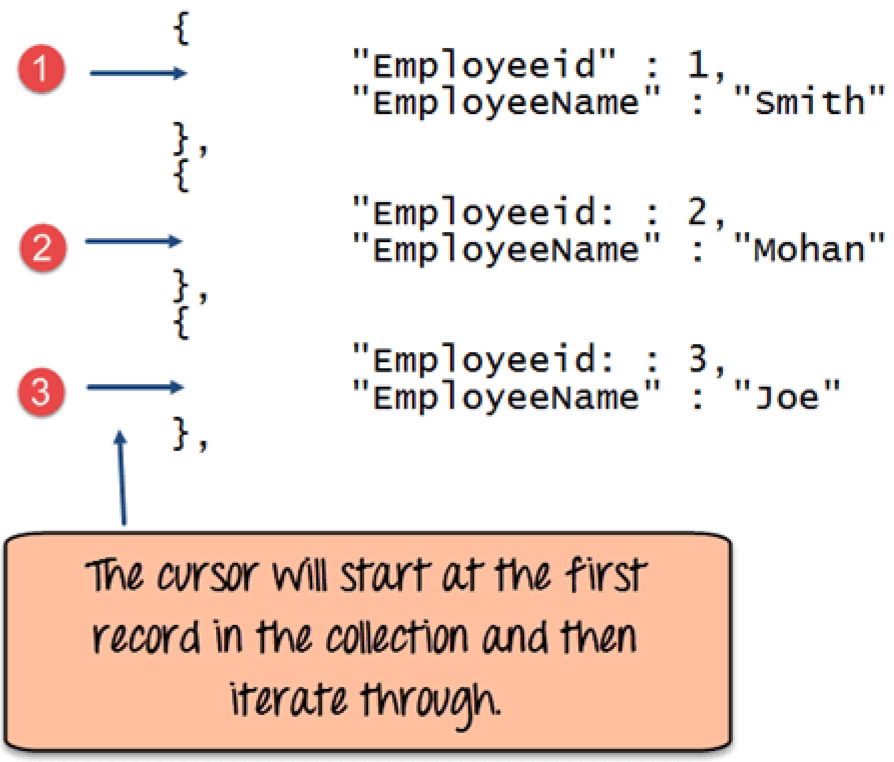
OPERATION	SYNTAX	DESCRIPTION
Equality	{“key” : “value”}	Matches values that are equal to a specified value.
Less Than	{“key” :{$lt:”value”}}	Matches values that are less than a specified value.
Greater Than	{“key” :{$gt:”value”}}	Matches values that are greater than a specified value.
Less Than Equal to	{“key” :{$lte:”value”}}	Matches values that are less than or equal to a specified value.
Greater Than Equal to	{“key” :{$lte:”value”}}	Matches values that are greater than or equal to a specified value.
Not Equal to	{“key”:{$ne: “value”}}	Matches all values that are not equal to a specified value.
Logical AND	{ “$and”:[{exp1}, {exp2}, …, {expN}] }	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
Logical OR	{ “$or”:[{exp1}, {<exp2}, …, {expN}] }	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

OPERATION	SYNTAX	DESCRIPTION
Logical NOT	{ “$not”:[{exp1}, {exp2}, …, {expN}] }	Inverts the effect of a query expression and returns documents that do not match the query expression.

What is Cursor in MongoDB?

When the **db.collection.find ()** function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.

By default, the cursor will be iterated automatically when the result of the query is returned. But one can also explicitly go through the items returned in the cursor one by one. If you see the below example, if we have 3 documents in our collection, the cursor object will point to the first document and then iterate through all of the documents of the collection.


1
2
3
The cursor will start at the first record in the collection and then iterate through.

```
{
        "Employeeid" : 1,
        "EmployeeName" : "Smith"
},
{
        "Employeeid: : 2,
        "EmployeeName" : "Mohan"
},
{
        "Employeeid: : 3,
        "EmployeeName" : "Joe"
},
```

The following example shows how this can be done.

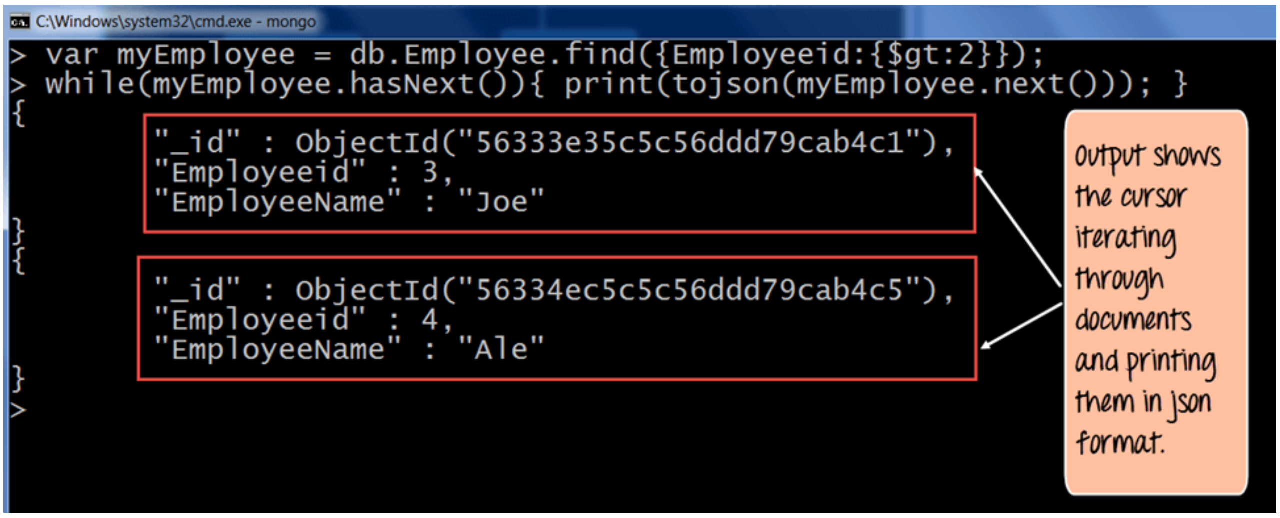
```
var myEmployee = db.Employee.find( { Employeeid : { $gt:2 }});

        while(myEmployee.hasNext())

        {

                print(tojson(myEmployee.next()));

        }
```

Code Explanation:

1. First we take the result set of the query which finds the Employee's whose id is greater than 2 and assign it to the JavaScript variable 'myEmployee'
2. Next we use the while loop to iterate through all of the documents which are returned as part of the query.
3. Finally for each document, we print the details of that document in JSON readable format.

If the command is executed successfully, the following Output will be shown

Output:



```
C:\Windows\system32\cmd.exe - mongo
> var myEmployee = db.Employee.find({Employeeid:{$gt:2}});
> while(myEmployee.hasNext()){ print(tojson(myEmployee.next())); }
{
        "_id" : ObjectId("56333e35c5c56ddd79cab4c1"),
        "Employeeid" : 3,
        "EmployeeName" : "Joe"
}
{
        "_id" : ObjectId("56334ec5c5c56ddd79cab4c5"),
        "Employeeid" : 4,
        "EmployeeName" : "Ale"
}
>
```

Output shows the cursor iterating through documents and printing them in json format.

MONGO QUERY LANGUAGE

MongoDB uses **the MongoDB Query Language** (MQL), designed for easy use by developers. **The** documentation compares MQL and SQL **syntax** for common database operations.

These **queries** are sent to the **MongoDB** server present in the Data Layer. Now, the **MongoDB** server receives the **queries** and passes the received **queries** to the storage engine. **MongoDB** server itself does not directly read or write the data to the files or disk or memory.

Example :

1. Sort

```
db.userdetails.find({"date_of_join" :
"16/10/2010","education":"M.C.A."}).sort({"profession":-
1}).pretty()
```

$query & $orderby

```
>db.userdetails.find({$query : {"date_of_join" : "16/10/2010","education":"M.C.A."},
$orderby : {"profession":-1}}).pretty();
```

The MongoDB server treats all the query parameters as a single object. Some object such as $query, $orderby etc can be used to get the same result as return simple mongo query language. The $query can be evaluated as the WHERE clause of SQL and $orderby sorts the results as specified order.

SQL TO MONGODB MAPPING CHART

https://docs.mongodb.com/manual/reference/sql-comparison/

MongoDB fetch documents using Dot notation

If we want to fetch documents from the collection "testtable" which contains the value of "community_name" is "MODERN MUSIC" under the "extra" of an JSON style object, the following mongodb command can be used :

```
>db.testtable.find({"extra.community_name" : "MODERN MUSIC"}).pretty();
```

Advance example of MongoDB Dot notation

If we want to fetch documents from the collection "testtable" which contain the value of "community_name" is "MODERN MUSIC" and "valued_friends_id" which is under the "friends" is "harry" and all the said is under "extra" of an JSON style object, the following mongodb command can be used :

```
> db.testtable.find({"extra.community_name" : "MODERN
MUSIC","extra.friends.valued_friends_id":"harry"}).pretty();
```

Display all documents from a collection with the help of find() method –

```
db.demo302.find();
```

Following is the query for field selection using dot notation –

```
>db.demo302.find({"details.Subject":"MongoDB"},{"details.Name":0,"d
etails.Age":0,_id:0,Id:0});
```

This will produce the following output −

```
{ "details" : [ { "Subject" : "MongoDB" } ] }
```