Date:
Practical No 2
Aim: Practical of Data collection, Data curation and management for Large-scale Data system Using MongoDB

Theory:

Data curation is management of the data lifecycle. The term is typically applied to efforts to collect and preserve historical data. The following are common elements of data curation.

Overview: Data Curation	
Type	Data
Definition	Management of the data lifecycle especially in regards to historical data.
Related Concepts	Data Governance Data Management Data Profiling Data Backup Data Lineage Retention Schedule Compliance Data Quality Data Availability Data Disposal

1. MongoDB Create database
2. MongoDB Drop Database
3. MongoDB Create collection
4. MongoDB Drop collection
5. MongoDB Insert Document
6. MongoDB Query Document
7. MongoDB Update Document
8. Delete document in MongoDB
9. MongoDB Projection
10. limit() and skip() method in MongoDB
11. Sorting of Documents in MongoDB
12. MongoDB Indexing

Practical slip may contain experiment on above topics. Please be sure that c:\data directory is created.

By default, MongoDB listens for connections from clients on port 27017, and stores data in the C:/data/db directory.

```
C:\>net start MongoDB
```

```
The MongoDB service is starting..
The MongoDB service was started successfully.
```

```
C:\>mongo
```

To display the name of current database:

```
>db
test
>
```

Create Database in MongoDB

Once you are in the MongoDB shell, create the database in MongoDB by typing this command:

```
use database_name
```

For example: create a database “beginnersbook”:

```
use beginnersbook
> use beginnersbook
switched to db beginnersbook
>
```

Note: If the database name you mentioned is already present then this command will connect you to the database. However if the database doesn’t exist then this will create the database with the given name and connect you to it.

At any point if you want to check the currently connected database just type the command **db**. This command will show the database name to which you are currently connected. This is really helpful command when you are working with several databases so that before creating a collection or inserting a document in database, you may want to ensure that you are in the right database.

```
> db
beginnersbook
>
```

To list down all the databases, use the command **show dbs**. This command lists down all the databases and their size on the disk.

```
> show dbs
EmployeeDB      0.000GB
Testdb          0.000GB
admin           0.000GB
beginnersbookdb 0.000GB
config          0.000GB
local           0.000GB
```

```
test           0.000GB
>
```

As you can see that the database “beginnersbook” that we have created is not present in the list of all the databases. This is because a database is not created until you save a document in it.

Now let us create a collection **user** and insert a document in it.

```
> db.user.insert({name: "Chaitanya", age: 30})
WriteResult({ "nInserted" : 1 })
> show dbs
EmployeeDB      0.000GB
Testdb          0.000GB
admin           0.000GB
beginnersbook   0.000GB
beginnersbookdb 0.000GB
config          0.000GB
local           0.000GB
test            0.000GB
>
```

Drop Database in MongoDB

We use db.dropDatabase() command to delete a database. You should be very careful when deleting a database as this will remove all the data inside that database, including collections and documents stored in the database.

MongoDB Drop Database

The syntax to drop a Database is:

```
db.dropDatabase()
```

We do not specify any database name in this command, because this command deletes the currently selected database. Lets see the steps to drop a database in MongoDB.

1. See the list of databases using **show dbs** command.

```
> show dbs
admin          0.000GB
beginnersbook  0.000GB
local          0.000GB
```

2. Switch to the database that needs to be dropped by typing this command.

```
use database_name
```

For example delete the database “beginnersbook”.

```
> use beginnersbook
switched to db beginnersbook
```

Note: Change the database name in the above command, from beginnersbook to the database that needs to be deleted.

3. Now, the currently selected database is beginnersbook so the command db.dropDatabase() would delete this database.

```
> db.dropDatabase()
{ "dropped" : "beginnersbook", "ok" : 1 }
```

The command executed successfully and showing the operation "dropped" and status "ok" which means that the database is dropped.

```
> use beginnersbook
switched to db beginnersbook

>  db.dropDatabase()
{ "dropped" : "beginnersbook", "ok" : 1 }

 > show dbs
EmployeeDB      0.000GB
Testdb          0.000GB
admin           0.000GB
beginnersbookdb  0.000GB
config          0.000GB
local           0.000GB
test            0.000GB
>
```

Create Collection in MongoDB

We know that the data in MongoDB is stored in form of documents. These documents are stored in Collection and Collection is stored in Database.

Method 1: Creating the Collection in MongoDB on the fly

The cool thing about MongoDB is that you need not to create collection before you insert document in it. With a single command you can insert a document in the collection and the MongoDB creates that collection on the fly.
Syntax: **db.collection_name.insert({key:value, key:value…})**
For example:
We don't have a collection beginnersbook in the database beginnersbookdb. This command will create the collection named "beginnersbook" on the fly and insert a document in it with the specified key and value pairs.

```
> use beginnersbookdb
switched to db beginnersbookdb
> db.beginnersbook.insert({
...   name: "Chaitanya",
...   age: 30,
...   website: "beginnersbook.com"
... })
WriteResult({ "nInserted" : 1 })
>
```

To check whether the document is successfully inserted, type the following command. It shows all the documents in the given collection.
Syntax: **db.collection_name.find()**

```
> db.beginnersbook.find()
{ "_id" : ObjectId("5c281a2dc23e08d1515fd9ca"), "name" : "Chaitanya", "age" : 30,
"website" : "beginnersbook.com" }
>
```

To check whether the collection is created successfully, use the following command.

```
show collections
```

This command shows the list of all the collections in the currently selected database.

```
> show collections
beginnersbook
students
>
```

Method 2: Creating collection with options before inserting the documents

We can also create collection before we actually insert data in it. This method provides you the options that you can set while creating a collection.

Syntax:

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

name is the collection name
and **options** is an optional field that we can use to specify certain parameters such as size, max number of documents etc. in the collection.

First lets see how this command is used for creating collection without any parameters:

```
> db.createCollection("students")
{
	"ok" : 0,
	"errmsg" : "a collection 'beginnersbookdb.students' already exists",
	"code" : 48,
	"codeName" : "NamespaceExists"
}

 > db.createCollection("students11")
{ "ok" : 1 }
```

```
>
```

Lets see the options that we can provide while creating a collection:

capped: type: boolean.
This parameter takes only true and false. This specifies a cap on the max entries a collection can have. Once the collection reaches that limit, it starts overwriting old entries.
The point to note here is that when you set the capped option to true you also have to specify the size parameter.

size: type: number.
This specifies the max size of collection (capped collection) in bytes.

max: type: number.
This specifies the max number of documents a collection can hold.

autoIndexId: type: boolean
The default value of this parameter is false. If you set it true then it automatically creates index field _id for each document.

Lets see an example of capped collection:

```
> db.createCollection("teachers", { capped : true, size : 9232768} )
{ "ok" : 1 }
>
```

This command will create a collection named “teachers” with the max size of 9232768 bytes. Once this collection reaches that limit it will start overwriting old entries.

```
> show collections
beginnersbook
students
students11
teachers
>
```

Drop collection in MongoDB

To drop a collection , first connect to the database in which you want to delete collection and then type the following command to delete the collection:

```
db.collection_name.drop()
```

Note: Once you drop a collection all the documents and the indexes associated with them will also be dropped. To preserve the indexes we use remove() function that only removes the documents in the collection but doesn't remove the collection itself and the indexes created on it.

MongoDB drop collection Example

For example delete a collection names "teachers" in my database "beginnersbookdb". Write the following commands in the given sequence.

```
> use beginnersbookdb
switched to db beginnersbookdb
> show collections
beginnersbook
students
students11
teachers
>  db.teachers.drop()
true
> show collections
beginnersbook
students
students11
>
```

As you can see that the command db.teachers.drop() returned true which means that the collection is deleted successfully.

MongoDB Insert Document

Syntax to insert a document into the collection:

```
db.collection_name.insert()
```

Lets take an example to understand this .

MongoDB Insert Document using insert() Example

Here we are inserting a document into the collection named "beginnersbook". The field "course" in the example below is an array that holds the several key-value pairs.

You should see a successful write message like this:

```
WriteResult({ "nInserted" : 1 })
```

The insert() method creates the collection if it doesn’t exist but if the collection is present then it inserts the document into it

```
> db.beginnersbook.insert(
...    {
...      name: "Chaitanya",
...      age: 30,
...      email: "admin@beginnersbook.com",
...      course: [ { name: "MongoDB", duration: 7 }, { name: "Java", duration: 30
} ]
...    }
... )
WriteResult({ "nInserted" : 1 })
>
```

Verification:
You can also verify whether the document is successfully inserted by typing following command:

```
db.collection_name.find()
```

In the above example, we inserted the document in the collection named “beginnersbook” so the command should be:

```
> db.beginnersbook.find()
{ "_id" : ObjectId("5c2d37734fa204bd77e7fc1c"), "name" : "Chaitanya", "age" : 30, "email" :
"admin@beginnersbook.com", "course" : [ { "name" : "MongoDB", "duration" : 7 }, { "name"
: "Java", "duration" : 30 } ] }
>
```

MongoDB Example: Insert Multiple Documents in collection

To insert multiple documents in collection, we define an array of documents and later we use the insert() method on the array variable as shown in the example below. Here we are inserting three documents in the collection named “students”. This command will insert the data in “students” collection, if the collection is not present then it will create the collection and insert these documents.

```
> var beginners =
...  [
...    {
... "StudentId" : 1001,
... "StudentName" : "Steve",
...        "age": 30
...    },
...    {
... "StudentId" : 1002,
... "StudentName" : "Negan",
...        "age": 42
...    },
...    {
... "StudentId" : 3333,
... "StudentName" : "Rick",
...        "age": 35
...    },
... ];
> db.students.insert(beginners);
Output:
BulkWriteResult({
       "writeErrors" : [ ],
       "writeConcernErrors" : [ ],
       "nInserted" : 3,
       "nUpserted" : 0,
       "nMatched" : 0,
       "nModified" : 0,
       "nRemoved" : 0,
       "upserted" : [ ]
})
>
```

As you can see that it shows number 3 in front of **nInserted**. this means that the 3 documents have been inserted by this command.

```
> db.students.find()
{ "_id" : ObjectId("5c281c90c23e08d1515fd9cc"), "StudentId" : 1001, "StudentName" :
"Steve", "age" : 30 }
{ "_id" : ObjectId("5c281c90c23e08d1515fd9cd"), "StudentId" : 1002, "StudentName" :
"Negan", "age" : 42 }
{ "_id" : ObjectId("5c281c90c23e08d1515fd9ce"), "StudentId" : 3333, "StudentName" :
"Rick", "age" : 35 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"), "StudentId" : 1001, "StudentName" :
"Steve", "age" : 30 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"), "StudentId" : 1002, "StudentName" :
"Negan", "age" : 42 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"), "StudentId" : 3333, "StudentName" :
"Rick", "age" : 35 }
>
```

You can print the output data in a JSON format so that you can read it easily. To print the data in JSON format run the command **db.collection_name.find().forEach(printjson)**

So in our case the command would be this:

```
> db.students.find().forEach(printjson)
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
        "StudentId" : 3333,
        "StudentName" : "Rick",
        "age" : 35
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
        "StudentId" : 3333,
        "StudentName" : "Rick",
        "age" : 35
}
>
```

MongoDB Query Document using find() method

Querying all the documents in JSON format

Lets say we have a collection `students` in a database named `beginnersbookdb`. To get all the documents we use this command:

```
db.students.find()
```

However the output we get is not in any format and less-readable. To improve the readability, we can format the output in JSON format with this command:

```
db.students.find().forEach(printjson);
```

OR simply use pretty() – It does the same thing.

```
> db.students.find().pretty()
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
        "StudentId" : 3333,
        "StudentName" : "Rick",
        "age" : 35
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
	"age" : 42
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
	"StudentId" : 3333,
	"StudentName" : "Rick",
	"age" : 35
}
```

Query Document based on the criteria

Instead of fetching all the documents from collection, we can fetch selected documents based on a criteria.

Equality Criteria:
For example: To fetch the data of “Steve” from students collection. The command for this should be:

```
> db.students.find({StudentName : "Steve"}).pretty()
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
	"StudentId" : 1001,
	"StudentName" : "Steve",
	"age" : 30
}
>
```

Greater Than Criteria:
Syntax:

```
db.collection_name.find({"field_name":{$gt:criteria_value}}).pretty()
```

For example: To fetch the details of students having age > 32 then the query should be:

```
> db.students.find({"age":{$gt:32}}).pretty()
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
	"StudentId" : 1002,
	"StudentName" : "Negan",
	"age" : 42
}
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
	"StudentId" : 3333,
	"StudentName" : "Rick",
        "age" : 35
}
>
```

Less than Criteria:
Syntax:

```
db.collection_name.find({"field_name":{$lt:criteria_value}}).pretty()
```

Example: Find all the students having id less than 3000. The command for this criteria would be:

```
> db.students.find({"StudentId":{$lt:3000}}).pretty()
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
>
```

Not Equals Criteria:
Syntax:

```
db.collection_name.find({"field_name":{$ne:criteria_value}}).pretty()
```

Example: Find all the students where id is not equal to 1002. The command for this criteria would be:

```
> db.students.find({"StudentId":{$ne:1002}}).pretty()
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
        "StudentId" : 3333,
        "StudentName" : "Rick",
        "age" : 35
}
```

```
>
```

Greater than equals Criteria:

```
db.collection_name.find({"field_name":{$gte:criteria_value}}).pretty()
```

Less than equals Criteria:

```
db.collection_name.find({"field_name":{$lte:criteria_value}}).pretty()
```

The pretty() method that we have added at the end of all the commands is not mandatory. It is just used for formatting purposes.

MongoDB - Update Document in a Collection

In MongoDB, we have two ways to update a document in a collection. 1) update() method 2) save() method. Although both the methods update an existing document, they are being used in different scenarios. The update() method is used when we need to update the values of an existing document while save() method is used to replace the existing document with the document that has been passed in it.

To update a document in MongoDB, we provide a criteria in command and the document that matches that criteria is updated.

Updating Document using update() method

Syntax:

```
db.collection_name.update(criteria, update_data)
```

Example:

For example: Use a collection named “got” in the database “beginnersbookdb”. The documents inside “got” are:

```
> use beginnersbookdb
switched to db beginnersbookdb
> show collections
beginnersbook
students
students11
> db.createCollection("got")
{ "ok" : 1 }
>

> var abc = [
... {
...       "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
...        "name" : "Jon Snow",
...        "age" : 32
... },
... {
...        "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
...        "name" : "Khal Drogo",
...        "age" : 36
... },
... {
...        "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
...        "name" : "Sansa Stark",
...        "age" : 20
... },
... {
...        "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
...        "name" : "Lord Varys",
...        "age" : 42
... },
... ];
> db.got.insert(abc);
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 4,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})

>  db.got.find().pretty()
{
        "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
        "name" : "Jon Snow",
        "age" : 32
}
{
        "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
        "name" : "Khal Drogo",
        "age" : 36
}
{
        "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
        "name" : "Sansa Stark",
        "age" : 20
}
```

```
{
	"_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
	"name" : "Lord Varys",
	"age" : 42
}
>
```

To update the name of Jon Snow with the name “Kit Harington”. The command for this would be:

```
db.got.update({"name":"Jon Snow"},{$set:{"name":"Kit Harington"}})
```

```
> db.got.update({"name":"Jon Snow"},{$set:{"name":"Kit Harington"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.got.find().pretty()
{
	"_id" : ObjectId("59bd2e73ce524b733f14dd65"),
	"name" : "Kit Harington",
	"age" : 32
}
{
	"_id" : ObjectId("59bd2e8bce524b733f14dd66"),
	"name" : "Khal Drogo",
	"age" : 36
}
{
	"_id" : ObjectId("59bd2e9fce524b733f14dd67"),
	"name" : "Sansa Stark",
	"age" : 20
}
{
	"_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
	"name" : "Lord Varys",
	"age" : 42
}
>
```

By default the update method updates a single document. In the above example we had only one document matching with the criteria, however if there were more then also only one document would have been updated. To enable update() method to update multiple documents you have to set “multi” parameter of this method to true as shown below.

To update multiple documents with the update() method:

```
db.got.update({"name":"Jon Snow"},
 {$set:{"name":"Kit Harington"}},{multi:true})
```

Updating Document using save() method

Syntax:

```
db.collection_name.save( {_id:ObjectId(), new_document} )
```

Lets take the same example that we have seen above. Now we want to update the name of “Kit Harington” to “Jon Snow”. To work with save() method you should know the unique _id field of that document.

A very important **point to note** is that when you do not provide the **_id** field while using save()
method, it calls insert() method and the passed document is inserted into the collection as a new document

To get the _id of a document, you can either type this command:

```
db.got.find().pretty()
```

Here got is a collection name. This method of finding unique _id is only useful when you have few documents, otherwise scrolling and searching for that _id in huge number of documents is tedious.

If you have huge number of documents then, to get the _id of a particular document use the criteria in find() method. For example: To get the _id of the document that we want to update using save() method, type this command:

```
> db.got.find({"name": "Kit Harington"}).pretty()
{
	"_id" : ObjectId("59bd2e73ce524b733f14dd65"),
	"name" : "Kit Harington",
	"age" : 32
}
> db.got.save({"_id" : ObjectId("59bd2e73ce524b733f14dd65"), "name":
...  "Jon Snow", "age": 30})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.got.find().pretty()
{
	"_id" : ObjectId("59bd2e73ce524b733f14dd65"),
	"name" : "Jon Snow",
	"age" : 30
}
{
	"_id" : ObjectId("59bd2e8bce524b733f14dd66"),
	"name" : "Khal Drogo",
	"age" : 36
}
```

```
{
        "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
        "name" : "Sansa Stark",
        "age" : 20
}
{
        "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
        "name" : "Lord Varys",
        "age" : 42
}
>
```

MongoDB Delete Document from a Collection

In this experiment we will learn how to delete documents from a collection. The remove() method is used for removing the documents from a collection in MongoDB.

Syntax of remove() method:

```
db.collection_name.remove(delete_criteria)
```

Delete Document using remove() method

Lets use a collection `students` in my MongoDB database named `beginnersbookdb`. The documents in `students` collection are:

```
> db.students.find().pretty()
{
        "_id" : ObjectId("59bcecc7668dcce02aaa6fed"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("59bcecc7668dcce02aaa6fef"),
        "StudentId" : 3333,
        "StudentName" : "Rick",
        "age" : 35
}
```

To remove the student from this collection who has a student id equal to 3333. Write a command using remove() method like this:

```
db.students.remove({"StudentId": 3333})
```

Output:

```
WriteResult({ "nRemoved" : 1 })
```

To verify whether the document is actually deleted. Type the following command:

```
db.students.find().pretty()
```

It will list all the documents of students collection.

```
> use beginnersbookdb
switched to db beginnersbookdb
> db.students.find().pretty()
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
	"StudentId" : 1001,
	"StudentName" : "Steve",
	"age" : 30
}
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
	"StudentId" : 1002,
	"StudentName" : "Negan",
	"age" : 42
}
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
	"StudentId" : 3333,
	"StudentName" : "Rick",
	"age" : 35
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
	"StudentId" : 1001,
	"StudentName" : "Steve",
	"age" : 30
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
	"StudentId" : 1002,
	"StudentName" : "Negan",
	"age" : 42
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
	"StudentId" : 3333,
	"StudentName" : "Rick",
	"age" : 35
}
```

```
> db.students.remove({"StudentId": 3333})
WriteResult({ "nRemoved" : 2 })

> db.students.find().pretty()
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
        "StudentId" : 1001,
        "StudentName" : "Steve",
        "age" : 30
}
{
        "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
>
```

How to remove only one document matching your criteria?

When there are more than one documents present in collection that matches the criteria then all those documents will be deleted if you run the remove command. However there is a way to limit the deletion to only one document so that even if there are more documents matching the deletion criteria, only one document will be deleted.

```
db.collection_name.remove(delete_criteria, justOne)
```

Here justOne is a Boolean parameter that takes only 1 and 0, if you give 1 then it will limit the the document deletion to only 1 document. This is an optional parameters as we have seen above that we have used the remove() method without using this parameter.

Example:

```
db.walkingdead.remove({"age": 32}, 1)
```

Remove all Documents

If you want to remove all the documents from a collection but does not want to remove the collection itself then you can use remove() method like this:

```
db.collection_name.remove({})
```

MongoDB Projection

This is used when we want to get the selected fields of the documents rather than all fields.

For example, we have a collection where we have stored documents that have the fields: StudentId, StudentName, age but we want to see only the StudentId of all the students then in that case we can use projection to get only the StudentId.

Syntax:

```
db.collection_name.find({},{field_key:1 or 0})
```

```
> db.students.find().pretty()
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
	"StudentId" : 1001,
	"StudentName" : "Steve",
	"age" : 30
}
{
	"_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
	"StudentId" : 1002,
	"StudentName" : "Negan",
	"age" : 42
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
	"StudentId" : 1001,
	"StudentName" : "Steve",
	"age" : 30
}
{
	"_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
        "StudentId" : 1002,
        "StudentName" : "Negan",
        "age" : 42
}
> db.students.find({}, {"_id": 0, "StudentId" : 1})
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
>
```

Value 1 means show that field and 0 means do not show that field. When we set a field to 1 in Projection other fields are automatically set to 0, except _id, so to avoid the _id we need to specifically set it to 0 in projection. The vice versa is also true when we set few fields to 0, other fields set to 1 automatically.

```
> db.students.find({}, {"_id": 0, "StudentName" : 0, "age" : 0})
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
>

> db.students.find({}, {"_id": 0, "StudentName" : 0, "age" : 1})
Error: error: {
        "ok" : 0,
        "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
        "code" : 2,
        "codeName" : "BadValue"
}
>
```

This is because we have set student_name to 0 and other field student_age to 1. We can’t mix these. You either set those fields that you don’t want to display to 0 or set the fields to 1 that you want to display.

MongoDB - limit( ) and skip( ) method

The limit() method in MongoDB

This method limits the number of documents returned in response to a particular query.
Syntax:

```
db.collection_name.find().limit(number_of_documents)
db.studentdata.find({student_id : {$gt:2002}}).pretty()
db.studentdata.find({student_id : {$gt:2002}}).limit(1).pretty()
```

MongoDB Skip() Method

The skip() method is used for skipping the given number of documents in the Query result.

To understand the use of skip() method, lets take the same example that we have seen above. In the above example we can see that by using limit(1) we managed to get only one document, which is the first document that matched the given criteria. What if you do not want the first document matching your criteria. For example we have two documents that have student_id value greater than 2002 but when we limited the result to 1 by using limit(1), we got the first document, in order to get the second document matching this criteria we can use skip(1) here which will skip the first document.

```
db.studentdata.find({student_id : {$gt:2002}}).limit(1).skip(1).pretty()
```

MongoDB sort() method

Sorting Documents using sort() method

Using sort() method, you can sort the documents in ascending or descending order based on a particular field of document.

Syntax of sort() method:

```
db.collecttion_name.find().sort({field_key:1 or -1})
```

1 is for ascending order and -1 is for descending order. The default value is 1.

For example: collection studentdata contains following documents:

```
> db.studentdata.find().pretty()
{
        "_id" : ObjectId("59bf63380be1d7770c3982af"),
        "student_name" : "Steve",
        "student_id" : 2002,
        "student_age" : 22
}
{
        "_id" : ObjectId("59bf63500be1d7770c3982b0"),
        "student_name" : "Carol",
        "student_id" : 2003,
        "student_age" : 22
}
{
        "_id" : ObjectId("59bf63650be1d7770c3982b1"),
        "student_name" : "Tim",
        "student_id" : 2004,
        "student_age" : 23
}
```

Let's display the student_id of all the documents in **descending order**:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({"student_id": -1})
{ "student_id" : 2004 }
{ "student_id" : 2003 }
{ "student_id" : 2002 }
```

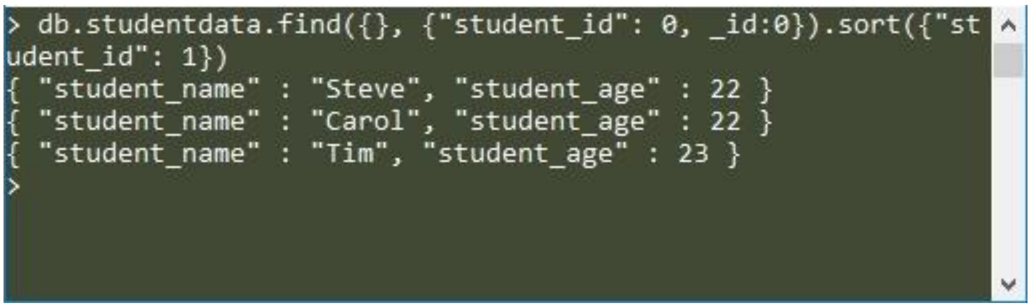
To display the student_id field of all the students in **ascending order**:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({"student_id": 1})
{ "student_id" : 2002 }
{ "student_id" : 2003 }
{ "student_id" : 2004 }
```

Default: The default is ascending order so if any value is not provided in the sort() method then it will sort the records in ascending order as shown below:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({})
{ "student_id" : 2002 }
{ "student_id" : 2003 }
{ "student_id" : 2004 }
```

You can also sort the documents based on the field that you don't want to display: For example, you can sort the documents based on student_id and display the student_age and student_name fields.



```
> db.studentdata.find({}, {"student_id": 0, _id:0}).sort({"st
udent_id": 1})
{ "student_name" : "Steve", "student_age" : 22 }
{ "student_name" : "Carol", "student_age" : 22 }
{ "student_name" : "Tim", "student_age" : 23 }
>
```

MongoDB Indexing

An **index** in MongoDB is a special data structure that holds the data of few fields of documents on which the index is created. Indexes improve the speed of search operations in database because instead of searching the whole document, the search is performed on the indexes that hold only few fields. On the other hand, having too many indexes can hamper the performance of insert, update and delete operations because of the additional write and additional data space used by indexes.

How to create index in MongoDB

```
db.collection_name.createIndex({field_name: 1 or -1})
```

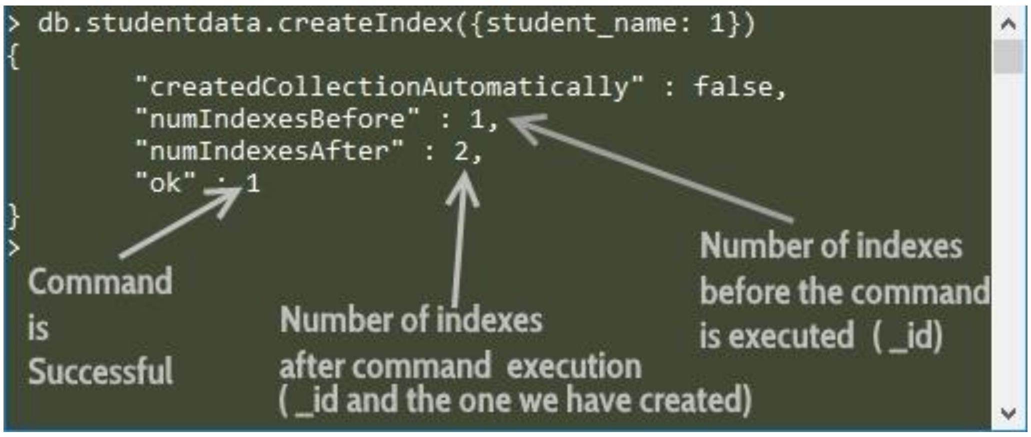
The value 1 is for ascending order and -1 is for descending order.

For example, use collection studentdata. The documents inside this collection have following fields:
student_name, student_id and student_age

Let’s create the index on student_name field in ascending order:

```
db.studentdata.createIndex({student_name: 1})
```
Output:

```
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```



```
> db.studentdata.createIndex({student_name: 1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Command is Successful
Number of indexes after command execution (_id and the one we have created)
Number of indexes before the command is executed (_id)

We have created the index on student_name which means when someone searches the document based on the student_name, the search will be faster because the index will be used for this search. So this is important to create the index on the field that will be frequently searched in a collection.

MongoDB - Finding the indexes in a collection

We can use getIndexes() method to find all the indexes created on a collection. The syntax for this method is:

```
db.collection_name.getIndexes()
```
So to get the indexes of studentdata collection, the command would be:

```
> db.studentdata.getIndexes()
```

```
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.studentdata"
        },
        {
                "v" : 2,
                "key" : {
                        "student_name" : 1
                },
                "name" : "student_name_1",
                "ns" : "test.studentdata"
        }
]
```

The output shows that we have two indexes in this collection. The default index created on _id and the index that we have created on student_name field.

MongoDB - Drop indexes in a collection

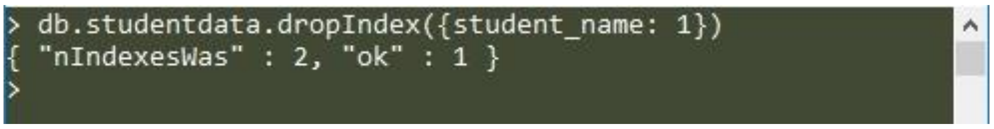
You can either drop a particular index or all the indexes.

Dropping a specific index:
For this purpose the dropIndex() method is used.

```
db.collection_name.dropIndex({index_name: 1})
```

Lets drop the index that we have created on `student_name` field in the collection `studentdata`. The command for this:

```
db.studentdata.dropIndex({student_name: 1})
```



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

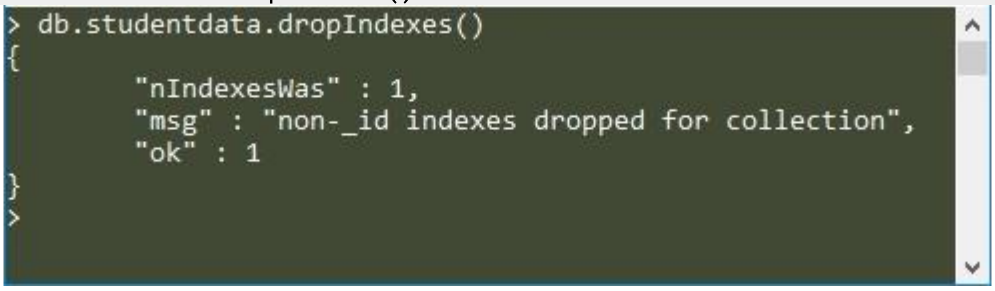
nIndexesWas: It shows how many indexes were there before this command got executed
ok: 1: This means the command is executed successfully.

Dropping all the indexes:
To drop all the indexes of a collection, we use dropIndexes() method.
Syntax of dropIndexs() method:

```
db.collection_name.dropIndexes()
```

Lets say we want to drop all the indexes of `studentdata` collection.

```
db.studentdata.dropIndexes()
```



```
> db.studentdata.dropIndexes()
{
        "nIndexesWas" : 1,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
>
```

The message “non-_id indexes dropped for collection” indicates that the default index _id will still remain and cannot be dropped. This means that using this method we can only drop indexes that we have created, we can’t drop the default index created on _id field.