

Birla Vishvakarma Mahavidyalaya
Engineering College, Vallabh Vidyanagar

4CP02: DATA ANALYTICS AND VISUALIZATION

Unit 2

The Big Data Technologies

Humongous = Enormous

- **Humongous**

- MongoDB

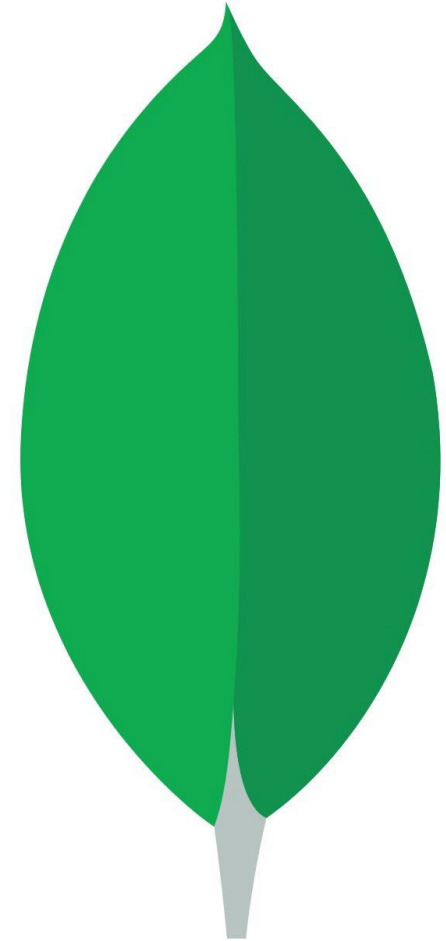


MongoDB

- MongoDB comes from the word humongous.
- Its founders built large Internet companies like DoubleClick.
- They consistently ran into the same problems with databases, one of the biggest problems being scalability.
- When they set out to build MongoDB, they wanted a database that scaled.
- Thus, “a humongous database,” or MongoDB.

MongoDB

- Founders believe that coding should be natural, and so should using a database.
- They want the experience of using MongoDB to be simple and natural.
- Thus, the leaf.



Advantages of MongoDB over RDBMS

- Schema less: MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Ease of scale-out: MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

When to go for MongoDB

- **Data Insert Consistency** : If there is a need for huge load of data to be written, without the worry of losing some data, then MongoDB should be preferred and really it's best suited.
- **Data Corruption Recovery**: When data recovery process needs to be faster, safe and automatic, MongoDB is preferred. In MySQL if a database (a few tables) become corrupt, you can repair them individually by deleting/updating the data. In MongoDB, you have to repair on a database level. But there is a command to do this automatically, but it reads all the data and re-writes it to a new set of files.

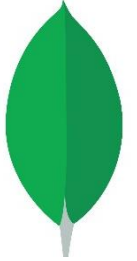
- Load Balancing: When data grows infinitely and proper load balancing of the same is required, MongoDB is the best solution. Because, it supports, faster replica setting options and its built in Sharding feature.

Documents



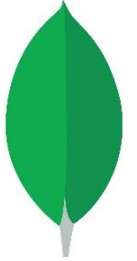
- When you see the term document, you might think of a word processing or spreadsheet file or perhaps even a paper document.
- They have nothing to do with document databases, at least with respect to the NoSQL type of database.
- Ex: HTML Documents, JSON Documents, XML Documents

Document Database



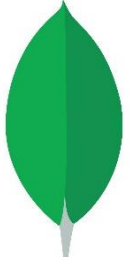
- Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.
- These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values.
- The documents stored are similar to each other but do not have to be exactly the same.
- Document databases store documents in the value part of the key-value store.

Document Database



- Let's look at how terminology compares in any RDBMS like Oracle and MongoDB.

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

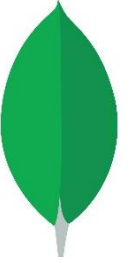


Document Database

- Document:

```
{ "firstname": "Martin",  
  "likes": [ "Biking",  
             "Photography" ],  
  "lastcity": "Boston",  
  "lastVisited":  
}
```

- The above document can be considered a row in a traditional RDBMS.

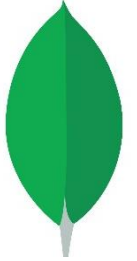


Document Database

- Document:

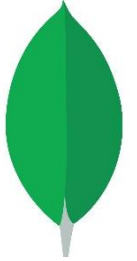
```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
  ],
  "lastcity": "Chicago"
}
```

Document Database



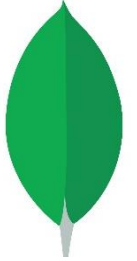
- Looking at the documents, we can see that they are similar, but have differences in attribute names.
- This is allowed in document databases.
- The schema of the data can differ across documents, but these documents can still belong to the same collection—unlike an RDBMS where every row in a table has to follow the same schema.
- We represent a list of *citiesvisited* as an array, or a list of addresses as list of documents embedded inside the main document.

Document Database



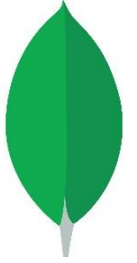
- Embedding child documents as sub-objects inside documents provides for easy access and better performance.
- If you look at the documents, you will see that some of the attributes are similar, such as *firstname* or *city*.
- At the same time, there are attributes in the second document which do not exist in the first document, such as *addresses*, while *likes* is in the first document but not the second.

Document Database

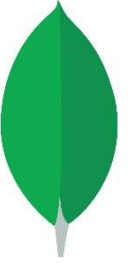


- This different representation of data is not the same as in RDBMS where every column has to be defined, and if it does not have data it is marked as empty or set to null.
- In documents, there are no empty attributes; if a given attribute is not found, we assume that it was not set or not relevant to the document.
- Documents allow for new attributes to be created without the need to define them or to change the existing documents.

Document Database



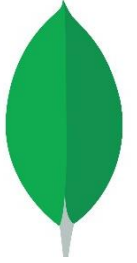
- Some of the popular document databases apart from MongoDB are
 - CouchDB
 - Terrastore
 - OrientDB
 - RavenDB
 - Lotus Notes



Document Database

- Databases, Collections, Documents
- A Database holds multiple Collections where each Collection can then hold multiple documents.
- Databases and Collections are created lazily (i.e. when a Document is inserted)
- A Document cannot be inserted a database, you need to use collections.

MongoDB Practice



- Get Ready
- Create Database
- Insert document
- Display document

Install MongoDB on Windows

- To install MongoDB on Windows, first download the latest release of MongoDB from <http://www.mongodb.org/downloads>.
- Now extract your downloaded file to c:\ drive or any other location.
- MongoDB requires a data folder to store its files. The default location for the MongoDB data directory is **c:\data\db**. So you need to create this folder using the Command Prompt. Execute the following command sequence.
- C:\>md data
- C:\>md data\db

- In the command prompt, navigate to the bin directory present in the MongoDB installation folder. Suppose my installation folder is

D:\set up\mongodb

- C:\Users\XYZ>d:
- D:\>cd "set up"
- D:\set up>cd mongodb
- D:\set up\mongodb>cd bin
- D:\set up\mongodb\bin>mongod.exe -dbpath "d:\setup\mongodb\data"

- Now to run the MongoDB, you need to open another command prompt and issue the following command.

```
D:\set up\mongodb\bin>mongo.exe
MongoDB shell version: 2.4.6
connecting to: test
>db.test.save( { a: 1 } )
>db.test.find()
{ "_id" : ObjectId(5879b0f65a56a454), "a" : 1 }
>
```

MongoDB – Create Database

- The use Command

Syntax

Basic syntax of **use DATABASE** statement is as follows:

```
use DATABASE_NAME
```

Example

If you want to create a database with name **<mydb>**, then **use DATABASE** statement would be as follows:

```
>use mydb  
switched to db mydb
```

To check your currently selected database, use the command **db**

```
>db  
mydb
```

- If you want to check your databases list, use the command show dbs.

```
>show dbs
```

- local 0.78125GB
- test 0.23012GB

- Your created database (mydb) is not present in list. To display database, you need to insert

at least one document into it.

```
>db.movie.insert({"name":"my name"})
```

```
>show dbs
```

- local 0.78125GB
- mydb 0.23012GB
- test 0.23012GB

MongoDB – Drop Database

- 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>
```

Collections in MongoDB

- Collection is a set of MongoDB documents. These documents are equivalent to the row of data in tables in RDBMS.
- Collections are a way of storing related data. Being schemaless, any type of Document can be saved in a collection, although similarity is recommended for index efficiency. Document's can have a maximum size of 4MB.

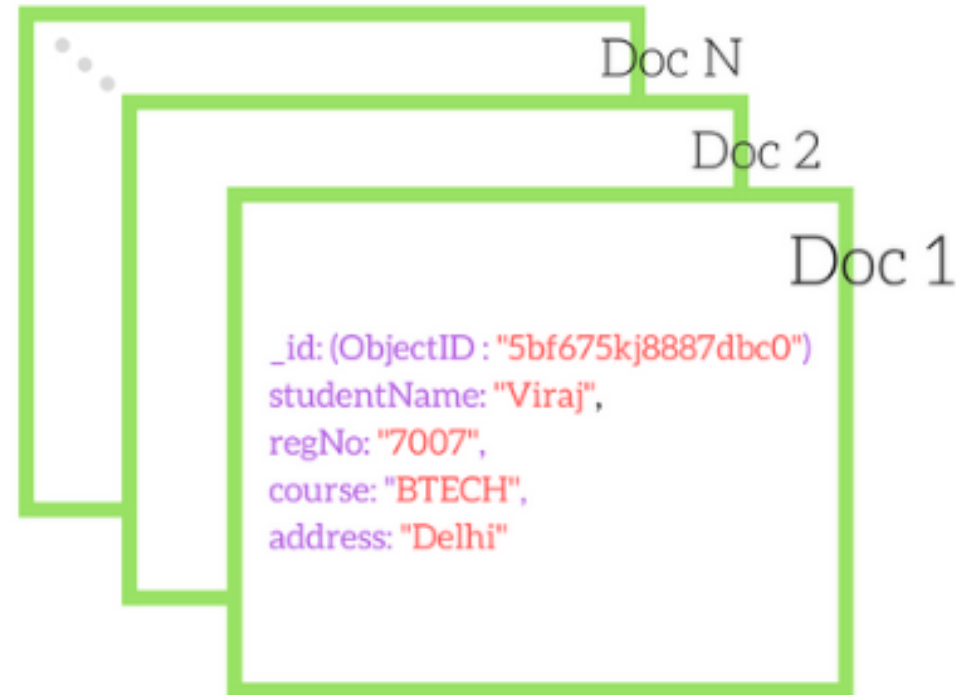
• We can use **namespace** to logically group and nest collections. **For example**
: There can be one collection named `db.student.users` to save user informations, then there can be others like `db.student.forum.questions` and `db.student.forum.answers` to store forum questions and answers respectively.

Sample Data in MongoDB

In the above figure, the field `_id` represents the primary key identifier of the given document.

Collection

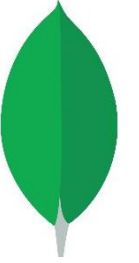
A collection can have multiple documents



you do not have to predefine the type of data to be stored, you can store anything you want. Remember, MongoDB is schema-less.

Example of Array as value

```
{ _id : 112233,  
  name : "Viraj",  
  education : [  
    {  
      year : 2029,  
      course : "BTECH",  
      college : "IIT, Delhi"  
    },  
    {  
      year : 2031,  
      course : "MS",  
      college : "Harvard College"  
    }  
  ]  
}
```



MongoDB Data Types

- Text
- Boolean
- Number
 - Int32
 - Int64
 - Decimal
- ObjectId
- ISODate
- Timestamp
- Embedded Documents & Arrays

Example of Different Datatypes in one Document

```
var mydoc = {  
  _id : ObjectId("5099803df3f4948bd2f98391"),  
  name : { first: "Alan", last: "Turing" },  
  birth : new Date('Jun 23, 1912'),  
  death : new Date('Jun 07, 1954'),  
  contribs : [ "Turing machine", "Turing test", "Turingery" ],  
  view : NumberLong(1250000)  
}
```

The createCollection() Method

- Syntax
- Basic syntax of createCollection() command is as follows –
- `db.createCollection(name, options)`
- **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

- Basic syntax of **createCollection()** method without options is as follows –
- `db.createCollection("mycollection")`
- `{ "ok" : 1 }`

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use –

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Create a Capped Collection

- When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection. The size of the capped collection includes a small amount of space for internal overhead.
- `db.createCollection("log", { capped: true, size: 100000 })`
- `db.createCollection("log", { capped : true, size : 5242880, max : 5000 })`

Check if a Collection is Capped

- Use the `isCapped()` method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

Convert a Collection to Capped

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

The insert() Method

- To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.
- The basic syntax of insert() command is as follows –

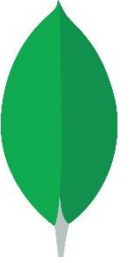
>db.COLLECTION_NAME.insert(document)

```
db.users.insert({
  _id : ObjectId("507f191e810c19729de860ea"),
  title: "MongoDB Overview",
  description: "MongoDB is no sql database",
  by: "tutorial",
  url: "http://www.tutorialspoint.com",
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
})
```

- In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.

- `_id : ObjectId("507f191e810c19729de860ea"),`
- `_id` is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows –
- `_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)`

- Do practice by taking any other example.



Document Database

- CRUD Operations CRUD = Create, Read, Update, Delete
- MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. `insertOne()`, `insertMany()`)
- Some methods require an argument (e.g. `insertOne()`), others don't (e.g. `find()`) inserted)
- `find()` returns a cursor, NOT a list of documents.
- Use filters to find specific documents



Document Database

- CRUD Operations CRUD = Create, Read, Update, Delete
- We can omit double quotes for keys while inserting
- custom _id can be inserted/created

Create Operations –

The create or insert operations are used to insert or add new documents in the collection. If a collection does not exist, then it will create a new collection in the database. You can perform, create operations using the following methods provided by the MongoDB:

Method	Description
db.collection.insertOne()	It is used to insert a single document in the collection.
db.collection.insertMany()	It is used to insert multiple documents in the collection.
db.createCollection()	It is used to create an empty collection.

insertOne()

```
db.RecordsDB.insertOne({  
  name: "Marsh",  
  age: "6 years",  
  species: "Dog",  
  ownerAddress: "380 W. Fir Ave",  
  chipped: true  
})
```

If the create operation is successful, a new document is created. The function will return an object where “acknowledged” is “true” and “insertID” is the newly created “ObjectId.”

```
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")  
}
```

insertMany():- It's possible to insert multiple items at one time by calling the insertMany() method on the desired collection.

```
db.RecordsDB.insertMany([  
  {  
    name: "Marsh",  
    age: "6 years",  
    species: "Dog",  
    ownerAddress: "380 W. Fir Ave",  
    chipped: true},  
  {name: "Kitana",  
    age: "4 years",  
    species: "Cat",  
    ownerAddress: "521 E. Cortland",  
    chipped: true}])
```

```
{  
  "acknowledged" : true,  
  "insertedIds" : [  
    ObjectId("5fd98ea9ce6e8850d88270b4"),  
    ObjectId("5fd98ea9ce6e8850d88270b5")  
  ]  
}
```

Read Operations

- The [read](#) operations allow you to supply special query filters and criteria that let you specify which documents you want.
- MongoDB has two methods of reading documents from a collection:
 - [db.collection.find\(\)](#)
 - [db.collection.findOne\(\)](#)

find()

- In order to get all the documents from a collection, we can simply use the find() method on our chosen collection.
- Executing just the find() method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" :  
"4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" :  
true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" :  
"6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" :  
true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3  
years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true  
}
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" :  
"8 years", "species" : "Dog", "ownerAddress" : "900 W. Wood Way",  
"chipped" : true }
```

search by value:-

- If you want to get more specific with a read operation and find a desired subsection of the records, you can use the filtering criteria to choose what results should be returned.
- One of the most common ways of filtering the results is to search by value.


```
db.RecordsDB.find({"species":"Cat"})
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana",  
  "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland",  
  "chipped" : true }
```

findOne()

- In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection.
- If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.
- If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

db.{collection}.findOne({query}, {projection})

```
db.RecordsDB.find({"age":"8 years"})
```

- We would get the following result:
- { "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }

Update Operations –

- The update operations are used to update or modify the existing document in the collection.

Method	Description
db.collection.updateOne()	It is used to update a single document in the collection that satisfy the given criteria.
db.collection.updateMany()	It is used to update multiple documents in the collection that satisfy the given criteria.
db.collection.replaceOne()	It is used to replace single document in the collection that satisfy the given criteria.

- Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.
- You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

- For MongoDB CRUD, there are three different methods of updating documents:
- [db.collection.updateOne\(\)](#)
- [db.collection.updateMany\(\)](#)
- [db.collection.replaceOne\(\)](#)

updateOne()

- We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection.
- The update filter defines which items we want to update, and the update action defines how to update those items.
- We first pass in the update filter. Then, we use the “\$set” key and provide the fields we want to update as a value.
- This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Output:-

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
```

```
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```


updateMany()

- *updateMany()* allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Dog" }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog" }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog" }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" : "Dog" }
```

replaceOne()

- The replaceOne() method is used to replace a single document in the specified collection.
- replaceOne() replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat" }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog" }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog" }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

- [Delete](#) operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.
- MongoDB has two different methods of deleting records from a collection:
- [db.collection.deleteOne\(\)](#)
- [db.collection.deleteMany\(\)](#)

deleteOne()

- *deleteOne()* is used to remove a document from a specified collection on the MongoDB server.
- A filter criteria is used to specify the item to delete.
- It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()  
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat"  
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog"  
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog"
```

deleteMany()

- deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation.
- A list is passed into the method and the individual items are defined with filter criteria as in deleteOne().


```
db.RecordsDB.deleteMany({species:"Dog"})
```

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
```

```
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Practice

Try all this function for student collection.

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

AND in MongoDB

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

```
>db.mycol.find({$and:[{"by":"tutorialpoint"}, {"title": "MongoDB Overview"}]})
```

OR in MongoDB

```
>db.mycol.find( { $or: [ {key1: value1}, {key2:value2} ] }).pretty()
```

```
>db.mycol.find({$or:[{"by":"tutorials point"}, {"title": "MongoDB  
Overview"}]}).pretty()
```

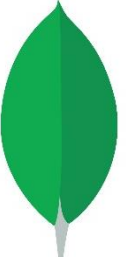
AND and OR Together

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},  
    {"title": "MongoDB Overview"}]}).pretty()  
r
```

NOT in MongoDB

```
>db.COLLECTION_NAME.find( { $NOT: [ {key1: value1}, {key2:value2} ] }).pretty()  
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
```

MongoDB Query Exercise



- count
- limit
- sort
- skip

Count

Count will convert our result set into a number. We can use it in two ways. We can either chain it:

```
db.people.find({sharks: 3}).count()
```

or we can use it in place of find:

```
db.people.count({sharks: 3})
```

To count the people who have exactly three sharks.

Limit and Skip

Limit will allow us to limit the results in the output set. Skip will allow us to offset the start. Between them they give us pagination.

For example

```
db.biscuits.find().limit(5)
```

will give us the first 5 biscuits. If we want the next 5 we can skip the first 5.

```
db.biscuits.find().limit(5).skip(5)
```

Sort

We can sort the results using the sort operator, like so:

```
db.spiders.find().sort({hairiness: 1})
```

This will sort the spiders in ascending order of hairiness. You can reverse the sort by passing -1.

```
db.spiders.find().sort({hairiness: -1})
```

This will get the most hairy spiders first.

We can sort by more than one field:

```
db.spiders.find().sort({  
    hairiness: -1,  
    scariness: -1  })
```

We might also sort by nested fields:

```
db.spiders.find().sort({'web.size': -1})
```

will give the spiders with the largest webs.

update + insert = upsert

```
db.employee.findAndModify({query:{name:"Ram"},  
                           update:{$set:{department:"Development"}},  
                           upsert:true})
```

```
db.employee.update({name:"Priya"}, {$set: {department:  
"HR"}},{upsert:true})
```

```
db.example.update({Name: "Rekha"}, // Query parameter  
                  {$set: {Phone: '7841235468 '}, // Update document  
                  $setOnInsert: {Gender: 'Female'}},  
                  {upsert: true})
```

Save()

Save a New Document without Specifying an `_id` Field

In the following example, `save()` method performs an insert since the document passed to the method does not contain the `_id` field:

```
db.products.save( { item: "book", qty: 40 } )
```

During the insert, the shell will create the `_id` field with a unique ObjectId value, as verified by the inserted document:

```
{ "_id" : ObjectId("50691737d386d8fadbd6b01d"), "item" : "book", "qty" :  
40 }
```

Aggregate operation

- Aggregations operations process data records and return computed results.
- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- In SQL `count(*)` and `with group by` is an equivalent of MongoDB aggregation.
- Basic syntax of `aggregate()` method is as follows –
- `>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)`

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
  description: 'No sql database is very fast',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 10
},
{
  _id: ObjectId(7df78ad8902e)
  title: 'Neo4j Overview',
  description: 'Neo4j is no sql database',
  by_user: 'Neo4j',
  url: 'http://www.neo4j.com',
  tags: ['neo4j', 'database', 'NoSQL'],
  likes: 750
},
```

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}
{ "_id" : "tutorials point", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>
```

Sql equivalent query for the above use case will be

select by_user, count(*) from mycol group by by_user.

Expression	Description	Example
\$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}])
\$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}])
\$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}])
\$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}])

\$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push: "\$url"}}}])
\$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}])
\$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : {_id : "\$by_user", first_url : {\$first : "\$url"}}}])
\$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : {_id : "\$by_user", last_url : {\$last : "\$url"}}}])

Pipeline Concept

- In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on.
- MongoDB also supports same concept in aggregation framework.
- There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline).

Following are the possible stages in aggregation framework

- **\$project** – Used to select some specific fields from a collection.
- **\$match** – This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **\$group** – This does the actual aggregation as discussed above.
- **\$sort** – Sorts the documents.

- **\$skip** – With this, it is possible to skip forward in the list of documents for a given amount of documents.
- **\$limit** – This limits the amount of documents to look at, by the given number starting from the current positions.
- **\$unwind** – This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

- **Displaying the total number of students in one section only**

```
db.students.aggregate([{$match:{sec:"B"}},{ $count:"Total student in  
sec:B"}])
```

In this example, for taking a count of the number of students in section B we first filter the documents using the `$match` operator, and then we use the `$count` accumulator to count the total number of documents that are passed after filtering from the `$match`.

```
> db.students.aggregate([{$match:{sec:"B"}},{ $count:"Total student in sec:B"}])  
{ "Total student in sec:B" : 3 }  
> _
```

Displaying the total number of students in both the sections and maximum age from both section

```
> db.students.aggregate([{$group: {_id:"$sec", total_st: {$sum:1}, max_age:{$max:"$age"} } }])
{ "_id" : "A", "total_st" : 4, "max_age" : 37 }
{ "_id" : "B", "total_st" : 3, "max_age" : 40 }
>
```

- we use **\$group** to group, so that we can count for every other section in the documents, here **\$sum** sums up the document in each group and **\$max** accumulator is applied on age expression which will find the maximum age in each document.

- **Displaying details of students whose age is greater than 30 using match stage**

```
db.students.aggregate([{$match:{age:{$gt:30}}}] )
```

In this example, we display students whose age is greater than 30. So we use the **\$match** operator to filter out the documents.

- **Sorting the students on the basis of age**

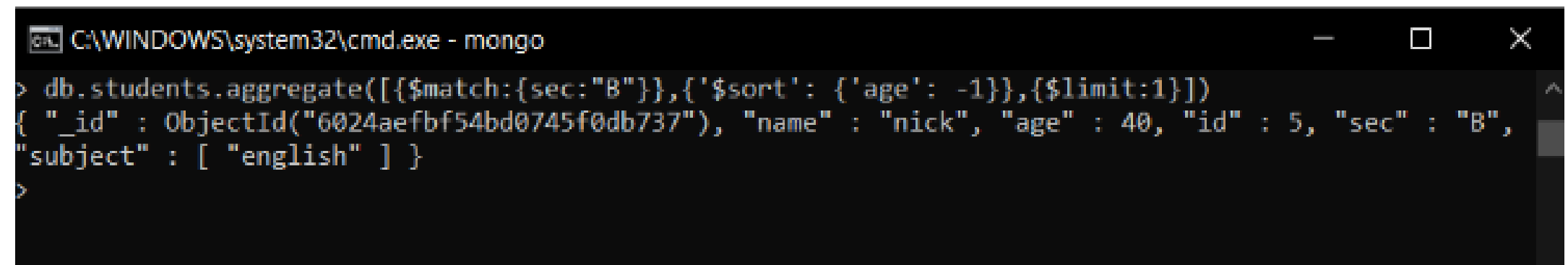
```
db.students.aggregate([{'$sort': {'age': 1}}])
```

In this example, we are using the **\$sort** operator to sort in ascending order we provide 'age':1 if we want to sort in descending order we can simply change 1 to -1 i.e. 'age':-1.

- Displaying details of a student having the largest age in the section – B

```
students.aggregate([{$match:{sec:"B"}},{ '$sort': {'age': -1}},{ $limit:1}])
```

In this example, first, we only select those documents that have section B, so for that, we use the **\$match** operator then we sort the documents in descending order using **\$sort** by setting 'age': -1 and then to only show the topmost result we use **\$limit**.



```
C:\WINDOWS\system32\cmd.exe - mongo
> db.students.aggregate([{$match:{sec:"B"}},{ '$sort': {'age': -1}},{ $limit:1}])
{ "_id" : ObjectId("6024aefbf54bd0745f0db737"), "name" : "nick", "age" : 40, "id" : 5, "sec" : "B",
"subject" : [ "english" ] }
>
```

- **Unwinding students on the basis of subject**

Unwinding works on array here in our collection we have array of subjects (which consists of different subjects inside it like math, physics, English, etc) so unwinding will be done on that i.e. the array will be deconstructed and the output will have only one subject not an array of subjects which were there earlier.

```
db.students.aggregate([{$unwind:"$subject"}])
```

```
C:\WINDOWS\system32\cmd.exe - mongo
> db.students.aggregate([{$unwind:"$subject"}])
{ "_id" : ObjectId("6024aefbf54bd0745f0db733"), "name" : "tony", "age" : 17, "id" : 1, "sec" : "A", "subject" : "physics" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db733"), "name" : "tony", "age" : 17, "id" : 1, "sec" : "A", "subject" : "maths" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db735"), "name" : "natasha", "age" : 17, "id" : 3, "sec" : "B", "subject" : "physics" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db735"), "name" : "natasha", "age" : 17, "id" : 3, "sec" : "B", "subject" : "english" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db736"), "name" : "bruce", "age" : 21, "id" : 4, "sec" : "B", "subject" : "physics" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db736"), "name" : "bruce", "age" : 21, "id" : 4, "sec" : "B", "subject" : "maths" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db736"), "name" : "bruce", "age" : 21, "id" : 4, "sec" : "B", "subject" : "biology" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db736"), "name" : "bruce", "age" : 21, "id" : 4, "sec" : "B", "subject" : "Chemistry" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db737"), "name" : "nick", "age" : 40, "id" : 5, "sec" : "B", "subject" : "english" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db738"), "name" : "groot", "age" : 4, "id" : 6, "sec" : "A", "subject" : "english" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db739"), "name" : "thanos", "age" : 4, "id" : 7, "sec" : "A", "subject" : "maths" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db739"), "name" : "thanos", "age" : 4, "id" : 7, "sec" : "A", "subject" : "physics" }
{ "_id" : ObjectId("6024aefbf54bd0745f0db739"), "name" : "thanos", "age" : 4, "id" : 7, "sec" : "A", "subject" : "chemistry" }
>
```

Map Reduce

- Map reduce is used for aggregating results for the large volume of data. Map reduce has two main functions one is a map that groups all the documents and the second one is the reduce which performs operation on the grouped data.
 - **`db.collectionName.mapReduce(mappingFunction, reduceFunction, {out:'Result'});`**

```
var mapfunction = function(){emit(this.age, this.marks)}  
var reducefunction = function(key, values){return Array.sum(values)}  
db.studentsMarks.mapReduce(mapfunction, reducefunction, {'out':'Result'}
```

Now, we will group the documents on the basis of age and find total marks in each age group. So, we will create two variables first mapfunction which will emit age as a key (expressed as “_id” in the output) and marks as value this emitted data is passed to our reducefunction, which takes key and value as grouped data, and then it performs operations over it. After performing reduction the results are stored in a collection here in this case the collection is Results.

C:\WINDOWS\system32\cmd.exe - mongo

```
> var mapfunction = function(){emit(this.age,this.marks)}
> var reducefunction = function(key,values){return Array.sum(values)}
> db.studentsMark.mapReduce(mapfunction,reducefunction,{'out':'Results'})
{ "result" : "Results", "ok" : 1 }
> db.Results.find()
{ "_id" : 19, "value" : 30 }
{ "_id" : 27, "value" : 55 }
{ "_id" : 37, "value" : 50 }
{ "_id" : 17, "value" : 70 }
> _
```

- **Displaying distinct names and ages (non-repeating)**

```
db.studentsMarks.distinct("name")
```

Here, we use a `distinct()` method that finds distinct values of the specified field (i.e., name).

```
C:\WINDOWS\system32\cmd.exe - mongo
> db.studentsMark.distinct("name")
[ "bruce", "bucky", "groot", "loki", "nick", "steve", "tony" ]
> db.studentsMark.distinct("age")
[ 17, 19, 27, 37 ]
> _
```