

Introduction to MongoDB- Contents

About	1
Session 1: Getting started with MongoDB	2
Section 1.1: Execution of a JavaScript file in MongoDB	2
Section 1.2: Making the output of find readable in shell	2
Section 1.3: Complementary Terms	3
Section 1.4: Installation	3
Section 1.5: Basic commands on mongo shell	6
Section 1.6: Hello World	6
Session 2: CRUD Operation	7
Section 2.1: Create	7
Section 2.2: Update	7
Section 2.3: Delete	8
Section 2.4: Read	8
Section 2.5: Update of embedded documents	9
Section 2.6: More update operators	10
Section 2.7: "multi" Parameter while updating multiple documents	10
Session 3: Getting database information	11
Section 3.1: List all collections in database	11
Section 3.2: List all databases	11
Session 4: Querying for Data (Getting Started)	12
Section 4.1: Find()	12
Section 4.2: FindOne()	12
Section 4.3: limit, skip, sort and count the results of the find() method	12
Section 4.4: Query Document - Using AND, OR and IN Conditions	14
Section 4.5: find() method with Projection	16
Section 4.6: Find() method with Projection	16
Session 5: Update Operators	18
Section 5.1: \$set operator to update specified field(s) in document(s)	18
Session 6: Upserts and Inserts	20
Section 6.1: Insert a document	20
Session 7: Collections	21
Section 7.1: Create a Collection	21
Section 7.2: Drop Collection	22
Session 8: Aggregation	23
Section 8.1: Count	23
Section 8.2: Sum	23
Section 8.3: Average	24
Section 8.4: Operations with arrays	25
Section 8.5: Aggregate query examples useful for work and learning	25
Section 8.6: Match	29
Section 8.7: Get sample data	30
Section 8.8: Remove docs that have a duplicate field in a collection (dedupe)	30
Section 8.9: Left Outer Join with aggregation (\$Lookup)	30
Section 8.10: Server Aggregation	31
Section 8.11: Aggregation in a Server Method	31
Section 8.12: Java and Spring example	32

<u>Session 9: Indexes</u>	34
<u>Section 9.1: Index Creation Basics</u>	34
<u>Section 9.2: Dropping/Deleting an Index</u>	36
<u>Section 9.3: Sparse indexes and Partial indexes</u>	36
<u>Section 9.4: Get Indices of a Collection</u>	37
<u>Section 9.5: Compound</u>	38
<u>Section 9.6: Unique Index</u>	38
<u>Section 9.7: Single field</u>	38
<u>Section 9.8: Delete</u>	38
<u>Section 9.9: List</u>	39
<u>Session 10: Bulk Operations</u>	40
<u>Section 10.1: Converting a field to another type and updating the entire collection in Bulk</u>	40

Introduction to No-SQL Database Development with MongoDB

By coding-bootcamps.com

About

Although there is no prerequisite for this course, taking our [Intro to Database design](#) course is highly recommended. Also, if you wish to use MongoDB for web development projects, taking our [Intro to Node.JS, Express.JS with MongoDB](#) is highly recommended.

After finishing this course, you can explore more training on database development by taking the below courses:

- [Learn SQL Programming by Examples](#)
- [Learn Graph Database Design by Examples](#)

Live Private Tutoring Sessions

While reading this course materials or watching our videos, if you like to receive extra assistance from our experienced instructors, you can take our Live Private Tutoring sessions via one of the below links:

- [Private tutoring sessions for system administrator management- Weekly and monthly plans](#)
- [Database design and SQL coding- Private tutoring sessions](#)

Session 1: Getting started with MongoDB

Version Release Date

3.6.1	2017-12-26
3.4	2016-11-29
3.2	2015-12-08
3.0	2015-03-03
2.6	2014-04-08
2.4	2013-03-19
2.2	2012-08-29
2.0	2011-09-12
1.8	2011-03-16
1.6	2010-08-31
1.4	2010-03-25
1.2	2009-12-10

Section 1.1: Execution of a JavaScript file in MongoDB

```
./mongo localhost:27017/mydb myjsfile.js
```

Explanation: This operation executes the `myjsfile.js` script in a `mongo` shell that connects to the `mydb` database on the `mongod` instance accessible via the `localhost` interface on port `27017`. `localhost:27017` is not mandatory as this is the default port `mongodb` uses.

Also, you can run a `.js` file from within `mongo` console.

```
>load("myjsfile.js")
```

Section 1.2: Making the output of find readable in shell

We add three records to our collection test as:

```
> db.test.insert({"key":"value1","key2":"Val2","key3":"val3"})
WriteResult({ "nInserted" : 1 })
> db.test.insert({"key":"value2","key2":"Val21","key3":"val31"})
WriteResult({ "nInserted" : 1 })
> db.test.insert({"key":"value3","key2":"Val22","key3":"val33"})
WriteResult({ "nInserted" : 1 })
```

If we see them via `find`, they will look very ugly.

```
> db.test.find()
{ "_id" : ObjectId("5790c5cecae25b3d38c3c7ae"), "key" : "value1", "key2" : "Val2", "key3" : "val3" }
{ "_id" : ObjectId("5790c5d9cae25b3d38c3c7af"), "key" : "value2", "key2" : "Val21", "key3" : "val31" }
{ "_id" : ObjectId("5790c5e9cae25b3d38c3c7b0"), "key" : "value3", "key2" : "Val22", "key3" : "val33" }
```

To work around this and make them readable, use the `pretty()` function.

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

```
{
  "_id" : ObjectId("5790c5cecae25b3d38c3c7ae"),
  "key" : "value1",
  "key2" : "val2",
  "key3" : "val3"
}

{
  "_id" : ObjectId("5790c5d9cae25b3d38c3c7af"),
  "key" : "value2",
  "key2" : "val21",
  "key3" : "val31"
}

{
  "_id" : ObjectId("5790c5e9cae25b3d38c3c7b0"),
  "key" : "value3",
  "key2" : "val22",
  "key3" : "val33"
}
>
```

Section 1.3: Complementary Terms

SQL Terms

Database Database

Table Collection

Entity / Row Document

Column Key / Field

Table Join [Embedded Documents](#)

Primary Key [Primary Key](#) (Default key `_id` provided by mongodb itself)

MongoDB Terms

Section 1.4: Installation

To install MongoDB, follow the steps below:

- **For Mac OS:**

There are two options for Mac OS: manual install or [homebrew](#).

Installing with [homebrew](#):

- Type the following command into the terminal:

```
$ brew install mongodb
```

Installing manually:

- Download the latest release [here](#). Make sure that you are downloading the appropriate file, specially check whether your operating system type is 32-bit or 64-bit. The downloaded file is in format `tgz`.
- Go to the directory where this file is downloaded. Then type the following command:

```
$ tar xvf mongodb-osx-xyz.tgz
```

Instead of `xyz`, there would be some version and system type information. The extracted folder would be same name as the `tgz` file. Inside the folder, there would be a subfolder named `bin` which would contain several binary file along with `mongod` and `mongo`.

- By default server keeps data in folder `/data/db`. So, we have to create that directory and then

run the server having the following commands:

```
$ sudo bash
# mkdir -p /data/db
# chmod 777 /data
# chmod 777 /data/db
# exit
```

- To start the server, the following command should be given from the current location:

```
$ ./mongod
```

It would start the server on port 27017 by default.

- To start the client, a new terminal should be opened having the same directory as before. Then the following command would start the client and connect to the server.

```
$ ./mongo
```

By default it connects to the `test` database. If you see the line like `connecting to: test`. Then you have successfully installed MongoDB. Congrats! Now, you can test Hello World to be sure.

- **For Windows:**

Download the latest release [here](#). Make sure that you are downloading the appropriate file, specially check whether your operating system type is 32-bit or 64-bit.

The downloaded binary file has extension `exe`. Run it. It will prompt an installation wizard.

Click **Next**.

Accept the license agreement and click **Next**.

Select **Complete** Installation.

Click on **Install**. It might prompt a window for asking administrator's permission. Click **Yes**.

After installation click on **Finish**.

Now, the MongoDB is installed on the path `C:/Program Files/MongoDB/Server/3.2/bin`. Instead of version 3.2, there could be some other version for your case. The path name would be changed accordingly.

`bin` directory contain several binary file along with `mongod` and `mongo`. To run it from other folder, you could add the path in system path. To do it:

- Right click on **My Computer** and select **Properties**.
- Click on **Advanced system setting** on the left pane.
- Click on **Environment Variables...** under the **Advanced** tab.
- Select **Path** from **System variables** section and click on **Edit...**
- Before Windows 10, append a semi-colon and paste the path given above. From Windows 10, there is a **New** button to add new path.
- Click **OKs** to save changes.

Now, create a folder named `data` having a sub-folder named `db` where you want to run the server.

Start command prompt from there. Either changing the path in cmd or clicking on **Open command window here** which would be visible after right clicking on the empty space of the folder GUI pressing

the Shift and Ctrl key together.

Write the command to start the server:

```
> mongod
```

It would start the server on port 27017 by default.

Open another command prompt and type the following to start client:

```
> mongo
```

By default it connects to the `test` database. If you see the line like `connecting to: test`. Then you have successfully installed MongoDB. Congrats! Now, you can test Hello World to be more confident.

- **For Linux:** Almost same as Mac OS except some equivalent command is needed.

For Debian-based distros (using `apt-get`):

- Import MongoDB Repository key.

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
gpg: Total number processed: 1\
gpg: imported: 1 (RSA: 1)
```

- Add repository to package list on **Ubuntu 16.04**.

```
$ echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse"
| sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

- on **Ubuntu 14.04**.

```
$ echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse"
| sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

- Update package list.

```
$ sudo apt-get update
```

- Install MongoDB.

```
$ sudo apt-get install mongodb-org
```

For Red Hat based distros (using `yum`):

- use a text editor which you prefer.

```
$ vi /etc/yum.repos.d/mongodb-org-3.4.repo
```

- Paste following text.

```
[mongodb-org-3.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

- Update package list.

```
$ sudo yum update
```

- Install MongoDB

```
$ sudo yum install mongodb-org
```

Section 1.5: Basic commands on mongo shell

Show all available databases:

```
show dbs;
```

Select a particular database to access, e.g. `mydb`. This will create `mydb` if it does not already exist:

```
use mydb;
```

Show all collections in the database (be sure to select one first, see above):

```
show collections;
```

Show all functions that can be used with the database:

```
db.mydb.help();
```

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

```
> db  
mydb
```

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

```
db.dropDatabase()
```

Section 1.6: Hello World

After installation process, the following lines should be entered in mongo shell (client terminal).

```
> db.world.insert({ "speech" : "Hello World!" });  
> cur = db.world.find(); x=cur.next(); print(x["speech"]);
```

```
Hello World!
```

Explanation:

- In the first line, we have inserted a `{ key : value }` paired document in the default database `test` and in the collection named `world`.
- In the second line we retrieve the data we have just inserted. The retrieved data is kept in a JavaScript variable named `cur`. Then by the `next()` function, we retrieved the first and only document and kept it in another js variable named `x`. Then printed the value of the document providing the key.

Session 2: CRUD Operation

Section 2.1: Create

```
db.people.insert({name: 'Tom', age: 28});
```

Or

```
db.people.save({name: 'Tom', age: 28});
```

The difference with `save` is that if the passed document contains an `_id` field, if a document already exists with that `_id` it will be updated instead of being added as new.

Two new methods to insert documents into a collection, in MongoDB 3.2.x:

Use `insertOne` to insert only one record:

```
db.people.insertOne({name: 'Tom', age: 28});
```

Use `insertMany` to insert multiple records:

```
db.people.insertMany([{name: 'Tom', age: 28}, {name: 'John', age: 25}, {name: 'Kathy', age: 23}])
```

Note that `insert` is highlighted as deprecated in every official language driver since version 3.0. The full distinction being that the shell methods actually lagged behind the other drivers in implementing the method. The same thing applies for all other CRUD methods

Section 2.2: Update

Update the **entire** object:

```
db.people.update({name: 'Tom'}, {age: 29, name: 'Tom'})
```

// New in MongoDB 3.2

```
db.people.updateOne({name: 'Tom'}, {age: 29, name: 'Tom'}) //Will replace only first matching document.
```

```
db.people.updateMany({name: 'Tom'}, {age: 29, name: 'Tom'}) //Will replace all matching documents.
```

Or just update a single field of a document. In this case `age`:

```
db.people.update({name: 'Tom'}, {$set: {age: 29}})
```

You can also update multiple documents simultaneously by adding a third parameter. This query will update all documents where the name equals `Tom`:

```
db.people.update({name: 'Tom'}, {$set: {age: 29}}, {multi: true})
```

// New in MongoDB 3.2

```
db.people.updateOne({name: 'Tom'}, {$set: {age: 30}}) //Will update only first matching document.
```

```
db.people.updateMany({name: 'Tom'}, {$set: {age: 30}}) //Will update all matching documents.
```

If a new field is coming for update, that field will be added to the document.

```
db.people.updateMany({name: 'Tom'}, {$set: {age: 30, salary: 50000}}) // Document will have `salary` field as well.
```

If a document is needed to be replaced,

```
db.collection.replaceOne({name: 'Tom'}, {name: 'Lakmal', age: 25, address: 'Sri Lanka'})
```

can be used.

Note: Fields you use to identify the object will be saved in the updated document. Field that are not defined in the update section will be removed from the document.

Section 2.3: Delete

Deletes all documents matching the query parameter:

```
// New in MongoDB 3.2
db.people.deleteMany({name: 'Tom'})

// All versions
db.people.remove({name: 'Tom'})
```

Or just one

```
// New in MongoDB 3.2
db.people.deleteOne({name: 'Tom'})

// All versions
db.people.remove({name: 'Tom'}, true)
```

MongoDB's `remove()` method. If you execute this command without any argument or without empty argument it will remove all documents from the collection.

```
db.people.remove();
```

or

```
db.people.remove({});
```

Section 2.4: Read

Query for all the docs in the `people` collection that have a `name` field with a value of `'Tom'`:

```
db.people.find({name: 'Tom'})
```

Or just the first one:

```
db.people.findOne({name: 'Tom'})
```

You can also specify which fields to return by passing a field selection parameter. The following will exclude the `_id` field and only include the `age` field:

```
db.people.find({name: 'Tom'}, {_id: 0, age: 1})
```

Note: by default, the `_id` field will be returned, even if you don't ask for it. If you would like not to get the `_id` back, you can just follow the previous example and ask for the `_id` to be excluded by specifying `_id: 0` (or `_id: false`). If you want to find sub record like address object contains country, city, etc.

```
db.people.find({'address.country': 'US'})
```

& specify field too if required

```
db.people.find({'address.country': 'US'}, {'name': true, 'address.city': true})
```

Remember that the result has a `.pretty()` method that pretty-prints resulting JSON:

```
db.people.find().pretty()
```

Section 2.5: Update of embedded documents

For the following schema:

```
{name: 'Tom', age: 28, marks: [50, 60, 70]}
```

Update Tom's marks to 55 where marks are 50 (Use the positional operator \$):

```
db.people.update({'name': 'Tom', marks: 50}, {"$set": {"marks.$": 55}})
```

For the following schema:

```
{name: 'Tom', age: 28, marks: [{subject: "English", marks: 90}, {subject: "Maths", marks: 100}, {subject: "Computes", marks: 20}]}
```

Update Tom's English marks to 85 :

```
db.people.update({'name': 'Tom', "marks.subject": "English"}, {"$set": {"marks.$.marks": 85}})
```

Explaining above example:

By using `{name: "Tom", "marks.subject": "English"}` you will get the position of the object in the marks array, where subject is English. In `"marks.$.marks"`, `$` is used to update in that position of the marks array

Update Values in an Array

The positional `$` operator identifies an element in an array to update without explicitly specifying the position of the element in the array.

Consider a collection students with the following documents:

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

To update 80 to 82 in the grades array in the first document, use the positional `$` operator if you do not know the position of the element in the array:

```
db.students.update(
  { _id: 1, grades: 80 },
  { $set: { "grades.$" : 82 } })
```

```
)
```

Section 2.6: More update operators

You can use other operators besides `$set` when updating a document. The `$push` operator allows you to push a value into an array, in this case we will add a new nickname to the `nicknames` array.

```
db.people.update({name: 'Tom'}, {$push: {nicknames: 'Tommy'}})
// This adds the string 'Tommy' into the nicknames array in Tom's document.
```

The `$pull` operator is the opposite of `$push`, you can pull specific items from arrays.

```
db.people.update({name: 'Tom'}, {$pull: {nicknames: 'Tommy'}})
// This removes the string 'Tommy' from the nicknames array in Tom's document.
```

The `$pop` operator allows you to remove the first or the last value from an array. Let's say Tom's document has a property called `siblings` that has the value `['Marie', 'Bob', 'Kevin', 'Alex']`.

```
db.people.update({name: 'Tom'}, {$pop: {siblings: -1}})
// This will remove the first value from the siblings array, which is 'Marie' in this case.
```

```
db.people.update({name: 'Tom'}, {$pop: {siblings: 1}})
// This will remove the last value from the siblings array, which is 'Alex' in this case.
```

Section 2.7: "multi" Parameter while updating multiple documents

To update multiple documents in a collection, set the `multi` option to `true`.

```
db.collection.update(
  query,
  update,
  {
    upsert: boolean,
    multi: boolean,
    writeConcern: document
  }
)
```

`multi` is optional. If set to `true`, updates multiple documents that meet the query criteria. If set to `false`, updates one document. The default value is `false`.

```
db.mycol.find() { "_id" : ObjectId(598354878df45ec5), "title":"MongoDB Overview" } { "_id" :
ObjectId(59835487adf45ec6), "title":"NoSQL Overview" } { "_id" : ObjectId(59835487adf45ec7),
"title":"Tutorials Point Overview" }
```

```
db.mycol.update({'title':'MongoDB Overview'}, {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

Session 3: Getting database information

Section 3.1: List all collections in database

```
show collections
```

or

```
show tables
```

or

```
db.getCollectionNames()
```

Section 3.2: List all databases

```
show dbs
```

or

```
db.adminCommand('listDatabases')
```

or

```
db.getMongo().getDBNames()
```

Session 4: Querying for Data (Getting Started)

Basic querying examples

Section 4.1: Find()

retrieve all documents in a collection

```
db.collection.find({});
```

retrieve documents in a collection using a condition (similar to WHERE in MYSQL)

```
db.collection.find({key: value});  
example  
db.users.find({email:"sample@email.com"});
```

retrieve documents in a collection using Boolean conditions (Query Operators)

```
//AND  
db.collection.find( {  
  $and: [  
    { key: value }, { key: value }  
  ]  
})  
//OR  
db.collection.find( {  
  $or: [  
    { key: value }, { key: value }  
  ]  
})  
//NOT  
db.inventory.find( { key: { $not: value } } )
```

more boolean operations and examples can be found [here](#)

NOTE: *find()* will keep on searching the collection even if a document match has been found , therefore it is inefficient when used in a large collection , however by carefully modeling your data and/or using indexes you can increase the efficiency of *find()*

Section 4.2: FindOne()

```
db.collection.findOne({});
```

the querying functionality is similar to *find()* but this will end execution the moment it finds one document matching its condition , if used with an empty object , it will fetch the first document and return it . [findOne\(\) mongodb api documentation](#)

Section 4.3: limit, skip, sort and count the results of the find() method

Similar to aggregation methods also by the *find()* method you have the possibility to limit, skip, sort and count the results. Let say we have following collection:

```
db.test.insertMany([
  {name:"Any", age:"21", status:"busy"},
  {name:"Tony", age:"25", status:"busy"},
  {name:"Bobby", age:"28", status:"online"},
  {name:"Sonny", age:"28", status:"away"},
  {name:"Cher", age:"20", status:"online"}
])
```

To list the collection:

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

Will return:

```
{ "_id" : ObjectId("592516d7fbd5b591f53237b0"), "name" : "Any", "age" : "21", "status" : "busy" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b1"), "name" : "Tony", "age" : "25", "status" : "busy" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b2"), "name" : "Bobby", "age" : "28", "status" : "online" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b3"), "name" : "Sonny", "age" : "28", "status" : "away" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b4"), "name" : "Cher", "age" : "20", "status" : "online" }
```

To skip first 3 documents:

```
db.test.find({}).skip(3)
```

Will return:

```
{ "_id" : ObjectId("592516d7fbd5b591f53237b3"), "name" : "Sonny", "age" : "28", "status" : "away" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b4"), "name" : "Cher", "age" : "20", "status" : "online" }
```

To sort descending by the field name:

```
db.test.find({}).sort({ "name" : -1 })
```

Will return:

```
{ "_id" : ObjectId("592516d7fbd5b591f53237b1"), "name" : "Tony", "age" : "25", "status" : "busy" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b3"), "name" : "Sonny", "age" : "28", "status" : "away" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b4"), "name" : "Cher", "age" : "20", "status" : "online" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b2"), "name" : "Bobby", "age" : "28", "status" : "online" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b0"), "name" : "Any", "age" : "21", "status" : "busy" }
```

If you want to sort ascending just replace -1 with 1

To count the results:

```
db.test.find({}).count()
```

Will return:

```
5
```

Also combinations of this methods are allowed. For example get 2 documents from descending sorted collection skipping the first 1:

```
db.test.find({}).sort({ "name" : -1 }).skip(1).limit(2)
```

Will return:

```
{ "_id" : ObjectId("592516d7fbd5b591f53237b3"), "name" : "Sonny", "age" : "28", "status" : "away" }
{ "_id" : ObjectId("592516d7fbd5b591f53237b4"), "name" : "Cher", "age" : "20", "status" : "online"
}
```

Section 4.4: Query Document - Using AND, OR and IN Conditions

All documents from `students` collection.

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

{
  "_id" : ObjectId("58f29a694117d1b7af126dca"),
  "studentNo" : 1,
  "firstName" : "Prosen",
  "lastName" : "Ghosh",
  "age" : 25
}
{
  "_id" : ObjectId("58f29a694117d1b7af126dcb"),
  "studentNo" : 2,
  "firstName" : "Rajib",
  "lastName" : "Ghosh",
  "age" : 25
}
{
  "_id" : ObjectId("58f29a694117d1b7af126dcc"),
  "studentNo" : 3,
  "firstName" : "Rizve",
  "lastName" : "Amin",
  "age" : 23
}
{
  "_id" : ObjectId("58f29a694117d1b7af126dcd"),
  "studentNo" : 4,
  "firstName" : "Jabed",
  "lastName" : "Bangali",
  "age" : 25
}
{
  "_id" : ObjectId("58f29a694117d1b7af126dce"),
  "studentNo" : 5,
  "firstName" : "Gm",
  "lastName" : "Anik",
  "age" : 23
}
```

Similar `mysql` Query of the above command.

```
SELECT * FROM students;
```



```
db.students.find({firstName:"Prosen"});
```

```
{ "_id" : ObjectId("58f2547804951ad51ad206f5"), "studentNo" : "1", "firstName" : "Prosen",  
"lastName" : "Ghosh", "age" : "23" }
```

Similar `mysql` Query of the above command.

```
SELECT * FROM students WHERE firstName = "Prosen";
```

AND Queries

```
db.students.find({  
  "firstName": "Prosen",  
  "age": {  
    "$gte": 23  
  }  
});
```

```
{ "_id" : ObjectId("58f29a694117d1b7af126dca"), "studentNo" : 1, "firstName" : "Prosen", "lastName"  
: "Ghosh", "age" : 25 }
```

Similar `mysql` Query of the above command.

```
SELECT * FROM students WHERE firstName = "Prosen" AND age >= 23
```

Or Queries

```
db.students.find({  
  "$or": [{  
    "firstName": "Prosen"  
  }, {  
    "age": {  
      "$gte": 23  
    }  
  }]  
});
```

```
{ "_id" : ObjectId("58f29a694117d1b7af126dca"), "studentNo" : 1, "firstName" : "Prosen", "lastName"  
: "Ghosh", "age" : 25 }  
{ "_id" : ObjectId("58f29a694117d1b7af126dcb"), "studentNo" : 2, "firstName" : "Rajib", "lastName"  
: "Ghosh", "age" : 25 }  
{ "_id" : ObjectId("58f29a694117d1b7af126dcc"), "studentNo" : 3, "firstName" : "Rizve", "lastName"  
: "Amin", "age" : 23 }  
{ "_id" : ObjectId("58f29a694117d1b7af126dcd"), "studentNo" : 4, "firstName" : "Jabed", "lastName"  
: "Bangali", "age" : 25 }  
{ "_id" : ObjectId("58f29a694117d1b7af126dce"), "studentNo" : 5, "firstName" : "Gm", "lastName" :  
"Anik", "age" : 23 }
```

Similar `mysql` Query of the above command.

```
SELECT * FROM students WHERE firstName = "Prosen" OR age >= 23
```

And OR Queries

```
db.students.find({
```

```

    firstName : "Prosen",
    $or : [
      { age : 23 },
      { age : 25 }
    ]
  });

{ "_id" : ObjectId("58f29a694117d1b7af126dca"), "studentNo" : 1, "firstName" : "Prosen", "lastName" : "Ghosh", "age" : 25 }

```

Similar mySql Query of the above command.

```
SELECT * FROM students WHERE firstName = "Prosen" AND age = 23 OR age = 25;
```

IN Queries This queries can improve multiple use of OR Queries

```

db.students.find(lastName:{$in:["Ghosh", "Amin"]})

{ "_id" : ObjectId("58f29a694117d1b7af126dca"), "studentNo" : 1, "firstName" : "Prosen", "lastName" : "Ghosh", "age" : 25 }
{ "_id" : ObjectId("58f29a694117d1b7af126dcb"), "studentNo" : 2, "firstName" : "Rajib", "lastName" : "Ghosh", "age" : 25 }
{ "_id" : ObjectId("58f29a694117d1b7af126dcc"), "studentNo" : 3, "firstName" : "Rizve", "lastName" : "Amin", "age" : 23 }

```

Similar mySql query to above command

```
SELECT * FROM students WHERE lastName IN ('Ghosh', 'Amin')
```

Section 4.5: find() method with Projection

The basic syntax of `find()` method with projection is as follows

```
> db.COLLECTION_NAME.find({}, {KEY:1});
```

If you want to show all documents without the age field then the command is as follows

```
db.people.find({}, {age : 0});
```

If you want to show all documents the age field then the command is as follows

Section 4.6: Find() method with Projection

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document.

The basic syntax of `find()` method with projection is as follows

```
> db.COLLECTION_NAME.find({}, {KEY:1});
```

If you want to to show all document without the age field then the command is as follows

```
> db.people.find({}, {age:0});
```

If you want to show only the age field then the command is as follows

```
> db.people.find({}, {age:1});
```

Note: `_id` field is always displayed while executing `find()` method, if you don't want this field, then you need to set it as `0`.

```
> db.people.find({}, {name:1, _id:0});
```

Note: `1` is used to show the field while `0` is used to hide the fields.

Session 5: Update Operators

parameters

Meaning

fieldName Field will be updated :**{name: 'Tom'}**

targetVaule Value will be assigned to the field :**{name: 'Tom'}**

Section 5.1: \$set operator to update specified field(s) in document(s)

I. Overview

A significant difference between MongoDB & RDBMS is MongoDB has many kinds of operators. One of them is update operator, which is used in update statements.

II. What happen if we don't use update operators?

Suppose we have a **student** collection to store student information (Table view):

age	name	sex
20	Tom	M
25	Billy	M
18	Mary	F
40	Ken	M

One day you get a job that need to change Tom's gender from "M" to "F". That's easy, right? So you write below statement very quickly based on your RDBMS experience:

```
db.student.update(  
  {name: 'Tom'}, // query criteria  
  {sex: 'F'} // update action  
);
```

Let's see what is the result:

age	name	sex
		F
25	Billy	M
18	Mary	F
40	Ken	M

We lost Tom's age & name! From this example, we can know that **the whole document will be overridden** if without any update operator in update statement. This is the default behavior of MongoDB.

III. \$set operator

If we want to change only the 'sex' field in Tom's document, we can use `$set` to specify which field(s) we want to update:

```
db.student.update(  
  {name: 'Tom'}, // query criteria  
  {$set: {sex: 'F'}} // update action  
);
```

The value of `$set` is an object, its fields stands for those fields you want to update in the documents, and the values of these fields are the target values.

So, the result is correct now:

age	name	sex
20	Tom	F
25	Billy	M
18	Mary	F
40	Ken	M

Also, if you want to change both 'sex' and 'age' at the same time, you can append them to `$set` :

```
db.student.update(  
  {name: 'Tom'}, // query criteria  
  {$set: {sex: 'F', age: 40}} // update action  
);
```

Session 6: Upserts and Inserts

Section 6.1: Insert a document

`_id` is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide `_id` while inserting the document. **If you didn't provide then MongoDB provide a unique id for every document.** These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental value.

```
db.mycol.insert({
  _id: ObjectId(7df78ad8902c),
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
})
```

Here *mycol* is a collection name, if the collection doesn't exist in the database, then MongoDB will create this collection and then insert document into it. In the inserted document if we don't specify the `_id` parameter, then MongoDB assigns an unique ObjectId for this document.

Session 7: Collections

Section 7.1: Create a Collection

First Select Or Create a database.

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

Using `db.createCollection("yourCollectionName")` method you can explicitly create Collection.

```
> db.createCollection("newCollection1")
{ "ok" : 1 }
```

Using `show collections` command see all collections in the database.

```
> show collections
newCollection1
system.indexes
>
```

The `db.createCollection()` method has the following parameters:

Parameter	Type	Description
name	string	The name of the collection to create.
options	document	<i>Optional.</i> Configuration options for creating a capped collection or for preallocating space in a new collection.

The following example shows the syntax of `createCollection()` method with few important options

```
>db.createCollection("newCollection4", {capped :true, autoIndexId : true, size : 6142800, max : 10000})
{ "ok" : 1 }
```

Both the `db.collection.insert()` and the `db.collection.createIndex()` operations create their respective collection if they do not already exist.

```
> db.newCollection2.insert({name : "XXX"})
> db.newCollection3.createIndex({accountNo : 1})
```

Now, Show All the collections using `show collections` command

```
> show collections
newCollection1
newCollection2
newCollection3
newCollection4
system.indexes
```

If you want to see the inserted document, use the `find()` command.

```
> db.newCollection2.find()
{ "_id" : ObjectId("58f26876cabafaeb509e9c1f"), "name" : "XXX" }
```

Section 7.2: Drop Collection

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

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

```
> use mydb
switched to db mydb

> show collections
newCollection1
newCollection2
newCollection3
system.indexes
```

Now drop the collection with the name `newCollection1`.

```
> db.newCollection1.drop()
true
```

Note: If the collection dropped successfully then the method will return `true` otherwise it will return

`false`. Again check the list of collections into database.

```
> show collections
newCollection2
newCollection3
system.indexes
```

Reference: MongoDB [drop\(\)](#) Method.

Session 8: Aggregation

Parameter

Details

- pipeline** array(A sequence of data aggregation operations or stages)
- options** document(optional, available only if pipeline present as an array)

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. MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

From Mongo manual <https://docs.mongodb.com/manual/aggregation/>

Section 8.1: Count

How do you get the number of Debit and Credit transactions? One way to do it is by using `count()` function as below.

```
> db.transactions.count({cr_dr : "D"});
```

or

```
> db.transactions.find({cr_dr : "D"}).length();
```

But what if you do not know the possible values of `cr_dr` upfront. Here Aggregation framework comes to play. See the below Aggregate query.

```
> db.transactions.aggregate([
  {
    $group : {
      _id : '$cr_dr', // group by type of transaction
      // Add 1 for each document to the count for this type of transaction
      count : {$sum : 1}
    }
  }
]);
```

And the result is

```
{
  "_id" : "C",
  "count" : 3
}
{
  "_id" : "D",
  "count" : 5
}
```

Section 8.2: Sum

How to get the summation of `amount`? See the below aggregate query.

```
> db.transactions.aggregate(
  [
    {
      $group : {
        _id : '$cr_dr',
        count : {$sum : 1},    //counts the number
        totalAmount : {$sum : '$amount'}    //sums the amount
      }
    }
  ]
);
```

And the result is

```
{
  "_id" : "C",
  "count" : 3.0,
  "totalAmount" : 120.0
}
{
  "_id" : "D",
  "count" : 5.0,
  "totalAmount" : 410.0
}
```

Another version that sums amount and fee.

```
> db.transactions.aggregate(
  [
    {
      $group : {
        _id : '$cr_dr',
        count : {$sum : 1},
        totalAmount : {$sum : { $sum : ['$amount', '$fee']}}
      }
    }
  ]
);
```

And the result is

```
{
  "_id" : "C",
  "count" : 3.0,
  "totalAmount" : 128.0
}
{
  "_id" : "D",
  "count" : 5.0,
  "totalAmount" : 422.0
}
```

Section 8.3: Average

How to get the average amount of debit and credit transactions?

```
> db.transactions.aggregate(
  [
    {
```

```

    $group : {
      _id : '$cr_dr', // group by type of transaction (debit or credit)
      count : {$sum : 1}, // number of transaction for each type
      totalAmount : {$sum : { $sum : ['$amount', '$fee']}}, // sum
      averageAmount : {$avg : { $sum : ['$amount', '$fee']}} // average
    }
  ]
)

```

The result is

```

{
  "_id" : "C", // Amounts for credit transactions
  "count" : 3.0,
  "totalAmount" : 128.0,
  "averageAmount" : 40.0
}
{
  "_id" : "D", // Amounts for debit transactions
  "count" : 5.0,
  "totalAmount" : 422.0,
  "averageAmount" : 82.0
}

```

Section 8.4: Operations with arrays

When you want to work with the data entries in arrays you first need to [unwind](#) the array. The unwind operation creates a document for each entry in the array. When you have lot's of documents with large arrays you will see an explosion in number of documents.

```

{ "_id" : 1, "item" : "myItem1", sizes: [ "S", "M", "L" ] }
{ "_id" : 2, "item" : "myItem2", sizes: [ "XS", "M", "XL" ] }

db.inventory.aggregate( [ { $unwind : "$sizes" } ] )

```

An important notice is that when a document doesn't contain the array it will be lost. From mongo 3.2 and up there are is an unwind option "preserveNullAndEmptyArrays" added. This option makes sure the document is preserved when the array is missing.

```

{ "_id" : 1, "item" : "myItem1", sizes: [ "S", "M", "L" ] }
{ "_id" : 2, "item" : "myItem2", sizes: [ "XS", "M", "XL" ] }
{ "_id" : 3, "item" : "myItem3" }

db.inventory.aggregate( [ { $unwind : { path: "$sizes", includeArrayIndex: "arrayIndex" } } ] )

```

Section 8.5: Aggregate query examples useful for work and learning

Aggregation is used to perform complex data search operations in the mongo query which can't be done in normal "find" query.

Create some dummy data:

```

db.employees.insert({"name":"Adma","dept":"Admin","languages":["german","french","english","hindi"],
,"age":30, "totalExp":10});
db.employees.insert({"name":"Anna","dept":"Admin","languages":["english","hindi"],"age":35,

```

```

"totalExp":11}});
db.employees.insert({"name":"Bob","dept":"Facilities","languages":["english","hindi"],"age":36,
"totalExp":14});
db.employees.insert({"name":"Cathy","dept":"Facilities","languages":["hindi"],"age":31,
"totalExp":4});
db.employees.insert({"name":"Mike","dept":"HR","languages":["english","hindi",
"spanish"],"age":26,"totalExp":3});
db.employees.insert({"name":"Jenny","dept":"HR","languages":["english","hindi",
"spanish"],"age":25,"totalExp":3});

```

Examples by topic:

1. Match: Used to match documents (like SQL where clause)

```

db.employees.aggregate([{$match:{dept:"Admin"}}]);
Output:
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "Admin", "languages" : [
"german", "french", "english", "hindi" ], "age" : 30, "totalExp" : 10 }
{ "_id" : ObjectId("54982fc92e9b4b54ec384a0e"), "name" : "Anna", "dept" : "Admin", "languages" : [
"english", "hindi" ], "age" : 35, "totalExp" : 11 }

```

2. Project: Used to populate specific field's value(s)

project stage will include `_id` field automatically unless you specify to disable.

```

db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{"name":1, "dept":1}}]);
Output:
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "Admin" }
{ "_id" : ObjectId("54982fc92e9b4b54ec384a0e"), "name" : "Anna", "dept" : "Admin" }

db.employees.aggregate({$project: {'_id':0, 'name': 1}})
Output:
{ "name" : "Adma" }
{ "name" : "Anna" }
{ "name" : "Bob" }
{ "name" : "Cathy" }
{ "name" : "Mike" }
{ "name" : "Jenny" }

```

3. Group: \$group is used to group documents by specific field, here documents are grouped by "dept" field's value. Another useful feature is that you can group by null, it means all documents will be aggregated into one.

```

db.employees.aggregate([{$group:{"_id":"$dept"}}]);

{ "_id" : "HR" }

{ "_id" : "Facilities" }

{ "_id" : "Admin" }

db.employees.aggregate([{$group:{"_id":null, "totalAge":{"$sum":"$age"}}}]);
Output:
{ "_id" : null, "noOfEmployee" : 183 }

```

4. Sum: \$sum is used to count or sum the values inside a group.

```

db.employees.aggregate([{$group:{"_id":"$dept", "noOfDept":{"$sum":1}}}]);
Output:

```

```
{ "_id" : "HR", "noOfDept" : 2 }
{ "_id" : "Facilities", "noOfDept" : 2 }
{ "_id" : "Admin", "noOfDept" : 2 }
```

5. Average: Calculates average of specific field's value per group.

```
db.employees.aggregate([{$group:{"_id":"$dept", "noOfEmployee":{$sum:1},
"avgExp":{$avg:"$totalExp"}}}]);
```

Output:

```
{ "_id" : "HR", "noOfEmployee" : 2, "totalExp" : 3 }
{ "_id" : "Facilities", "noOfEmployee" : 2, "totalExp" : 9 }
{ "_id" : "Admin", "noOfEmployee" : 2, "totalExp" : 10.5 }
```

6. Minimum: Finds minimum value of a field in each group.

```
db.employees.aggregate([{$group:{"_id":"$dept", "noOfEmployee":{$sum:1},
"minExp":{$min:"$totalExp"}}}]);
```

Output:

```
{ "_id" : "HR", "noOfEmployee" : 2, "totalExp" : 3 }
{ "_id" : "Facilities", "noOfEmployee" : 2, "totalExp" : 4 }
{ "_id" : "Admin", "noOfEmployee" : 2, "totalExp" : 10 }
```

7. Maximum: Finds maximum value of a field in each group.

```
db.employees.aggregate([{$group:{"_id":"$dept", "noOfEmployee":{$sum:1},
"maxExp":{$max:"$totalExp"}}}]);
```

Output:

```
{ "_id" : "HR", "noOfEmployee" : 2, "totalExp" : 3 }
{ "_id" : "Facilities", "noOfEmployee" : 2, "totalExp" : 14 }
{ "_id" : "Admin", "noOfEmployee" : 2, "totalExp" : 11 }
```

8. Getting specific field's value from first and last document of each group: Works well when document result is sorted.

```
db.employees.aggregate([{$group:{"_id":"$age", "lasts":{$last:"$name"},
"firsts":{$first:"$name"}}}]);
```

Output:

```
{ "_id" : 25, "lasts" : "Jenny", "firsts" : "Jenny" }
{ "_id" : 26, "lasts" : "Mike", "firsts" : "Mike" }
{ "_id" : 35, "lasts" : "Cathy", "firsts" : "Anna" }
{ "_id" : 30, "lasts" : "Adma", "firsts" : "Adma" }
```

9. Minimum with maximum:

```
db.employees.aggregate([{$group:{"_id":"$dept", "noOfEmployee":{$sum:1},
"maxExp":{$max:"$totalExp"}, "minExp":{$min:"$totalExp"}}}]);
```

Output:

```
{ "_id" : "HR", "noOfEmployee" : 2, "maxExp" : 3, "minExp" : 3 }
{ "_id" : "Facilities", "noOfEmployee" : 2, "maxExp" : 14, "minExp" : 4 }
{ "_id" : "Admin", "noOfEmployee" : 2, "maxExp" : 11, "minExp" : 10 }
```

10. Push and addToSet: Push adds a field's value from each document in group to an array used to project data in array format, addToSet is similar to push but it omits duplicate values.

```
db.employees.aggregate([{$group:{"_id":"dept", "arrPush":{$push:"$age"}, "arrSet":
{$addToSet:"$age"}}}]);
```

Output:

```
{ "_id" : "dept", "arrPush" : [ 30, 35, 35, 35, 26, 25 ], "arrSet" : [ 25, 26, 35, 30 ] }
```

11. Unwind: Used to create multiple in-memory documents for each value in the specified array type field, then we can do further aggregation based on those values.

```
db.employees.aggregate([{$match:{ "name": "Adma" }}, {$unwind: "$languages"}]);
```

Output:

```
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "HR", "languages" :  
"german", "age" : 30, "totalExp" : 10 }  
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "HR", "languages" :  
"french", "age" : 30, "totalExp" : 10 }  
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "HR", "languages" :  
"english", "age" : 30, "totalExp" : 10 }  
{ "_id" : ObjectId("54982fac2e9b4b54ec384a0d"), "name" : "Adma", "dept" : "HR", "languages" :  
"hindi", "age" : 30, "totalExp" : 10 }
```

12. Sorting:

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{ "name":1, "dept":1}}, {$sort: {name:  
1}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e553dedf0228d4862ac"), "name" : "Adma", "dept" : "Admin" }  
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin" }
```

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{ "name":1, "dept":1}}, {$sort: {name:  
-1}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin" }  
{ "_id" : ObjectId("57ff3e553dedf0228d4862ac"), "name" : "Adma", "dept" : "Admin" }
```

13. Skip:

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{ "name":1, "dept":1}}, {$sort: {name:  
-1}}, {$skip:1}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e553dedf0228d4862ac"), "name" : "Adma", "dept" : "Admin" }
```

14. Limit:

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{ "name":1, "dept":1}}, {$sort: {name:  
-1}}, {$limit:1}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin" }
```

15. Comparison operator in projection:

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{ "name":1, "dept":1, age: {$gt: ["$age",  
30]}}}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e553dedf0228d4862ac"), "name" : "Adma", "dept" : "Admin", "age" : false }  
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin", "age" : true }
```

16. Comparison operator in match:

```
db.employees.aggregate([{$match:{dept:"Admin", age: {$gt:30}}}, {$project:{ "name":1, "dept":1}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin" }
```

List of comparison operators: \$cmp, \$eq, \$gt, \$gte, \$lt, \$lte, and \$ne

17. Boolean aggregation operator in projection:

```
db.employees.aggregate([{$match:{dept:"Admin"}}, {$project:{"name":1, "dept":1, age: { $and: [ { $gt: [ "$age", 30 ] }, { $lt: [ "$age", 36 ] } ] }}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e553dedf0228d4862ac"), "name" : "Adma", "dept" : "Admin", "age" : false }
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin", "age" : true }
```

18. Boolean aggregation operator in match:

```
db.employees.aggregate([{$match:{dept:"Admin", $and: [{age: { $gt: 30 }}, {age: { $lt: 36 }} ] }}, {$project:{"name":1, "dept":1, age: { $and: [ { $gt: [ "$age", 30 ] }, { $lt: [ "$age", 36 ] } ] }}}]);
```

Output:

```
{ "_id" : ObjectId("57ff3e5e3dedf0228d4862ad"), "name" : "Anna", "dept" : "Admin", "age" : true }
```

List of boolean aggregation operators: \$and, \$or, and \$not.

Complete reference: <https://docs.mongodb.com/v3.2/reference/operator/aggregation/>

Section 8.6: Match

How to write a query to get all departments where average age of employees making less than or \$70000 is greater than or equal to 35?

In order to that we need to write a query to match employees that have a salary that is less than or equal to \$70000. Then add the aggregate stage to group the employees by the department. Then add an accumulator with a field named e.g. average_age to find the average age per department using the \$avg accumulator and below the existing \$match and \$group aggregates add another \$match aggregate so that we're only retrieving results with an average_age that is greater than or equal to 35.

```
db.employees.aggregate([
  {"$match": {"salary": {"$lte": 70000}}},
  {"$group": {"_id": "$dept",
    "average_age": {"$avg": "$age"}
  }},
  {"$match": {"average_age": {"$gte": 35}}}
])
```

The result is:

```
{
  "_id": "IT",
  "average_age": 31
}
{
  "id": "Customer Service",
```

```

"average_age": 34.5
}
{
  "_id": "Finance",
  "average age": 32.5
}

```

Section 8.7: Get sample data

To get random data from certain collection refer to `$sample` aggregation.

```
db.employees.aggregate({ $sample: { size:1 } })
```

where `size` stands for number of items to select.

Section 8.8: Remove docs that have a duplicate field in a collection (dedupe)

Note that the `allowDiskUse: true` option is optional but will help mitigate out of memory issues as this aggregation can be a memory intensive operation if your collection size is large - so i recommend to always use it.

```

var duplicates = [];

db.transactions.aggregate([
  { $group: {
    _id: { cr_dr: "$cr_dr"},
    dups: { "$addToSet": "$_id" },
    count: { "$sum": 1 }
  }
},
  { $match: {
    count: { "$gt": 1 }
  }
}],allowDiskUse: true)
)
.result
.forEach(function(doc) {
  doc.dups.shift();doc.dups.forEach(
    function(dupId){
      duplicates.push(dupId);
    }
  )
})
// printjson(duplicates);

// Remove all duplicates in one go
db.transactions.remove({_id:{$in:duplicates}})

```

Section 8.9: Left Outer Join with aggregation (\$Lookup)

```

let col_1 = db.collection('col_1');
let col_2 = db.collection('col_2');
col_1 .aggregate([
  { $match: { "_id": 1 } },
  {
    $lookup: {
      from: "col_2",
      localField: "id",

```



```

        foreignField: "id",
        as: "new_document"
    }
  }
}, function (err, result) {
  res.send(result);
});

```

This feature was newly released in the MongoDB **version 3.2**, which gives the user a stage to join one collection with the matching attributes from another collection

[MongoDB \\$Lookup documentation](#)

Section 8.10: Server Aggregation

```

Meteor.publish("someAggregation", function (args) {
  var sub = this;
  // This works for Meteor 0.6.5
  var db = MongoInternals.defaultRemoteCollectionDriver().mongo.db;

  // Your arguments to Mongo's aggregation. Make these however you want.
  var pipeline = [
    { $match: doSomethingWith(args) },
    { $group: {
      _id: whatWeAreGroupingWith(args),
      count: { $sum: 1 }
    } }
  ];

  db.collection("server_collection_name").aggregate(
    pipeline,
    // Need to wrap the callback so it gets called in a Fiber.
    Meteor.bindEnvironment(
      function(err, result) {
        // Add each of the results to the subscription.
        _each(result, function(e) {
          // Generate a random disposable id for aggregated documents
          sub.added("client_collection_name", Random.id(), {
            key: e._id.somethingOfInterest,
            count: e.count
          });
        });
        sub.ready();
      },
      function(error) {
        Meteor._debug( "Error doing aggregation: " + error);
      }
    )
  );
});

```

Section 8.11: Aggregation in a Server Method

Another way of doing aggregations is by using the `Mongo.Collection#rawCollection()`

This can only be run on the Server.

Here is an example you can use in Meteor 1.3 and higher:

```
Meteor.methods({
  'aggregateUsers'(someId) {
    const collection = MyCollection.rawCollection()
    const aggregate = Meteor.wrapAsync(collection.aggregate, collection)

    const match = { age: { $gte: 25 } }
    const group = { _id: '$age', totalUsers: { $sum: 1 } }

    const results = aggregate([
      { $match: match },
      { $group: group }
    ])

    return results
  }
})
```

Section 8.12: Java and Spring example

This is an example code to create and execute the aggregate query in MongoDB using Spring Data.

```
try {
  MongoClient mongo = new MongoClient();
  DB db = mongo.getDB("so");
  DBCollection coll = db.getCollection("employees");

  //Equivalent to $match
 DBObject matchFields = new BasicDBObject();
  matchFields.put("dept", "Admin");
 DBObject match = new BasicDBObject("$match", matchFields);

  //Equivalent to $project
 DBObject projectFields = new BasicDBObject();
  projectFields.put("_id", 1);
  projectFields.put("name", 1);
  projectFields.put("dept", 1);
  projectFields.put("totalExp", 1);
  projectFields.put("age", 1);
  projectFields.put("languages", 1);
 DBObject project = new BasicDBObject("$project", projectFields);

  //Equivalent to $group
 DBObject groupFields = new BasicDBObject("_id", "$dept");
  groupFields.put("ageSet", new BasicDBObject("$addToSet", "$age"));
  DBCollection employeeDocProjection = new BasicDBObject("$addToSet", new
BasicDBObject("totalExp", "$totalExp").append("age", "$age").append("languages",
"$languages").append("dept", "$dept").append("name", "$name"));
  groupFields.put("docs", employeeDocProjection);
 DBObject group = new BasicDBObject("$group", groupFields);

  //Sort results by age
 DBObject sort = new BasicDBObject("$sort", new BasicDBObject("age", 1));

  List<DBObject> aggregationList = new ArrayList<>();
  aggregationList.add(match);
  aggregationList.add(project);
  aggregationList.add(group);
  aggregationList.add(sort);
}
```

```

AggregationOutput output = coll.aggregate(aggregationList);

for (DBObject result : output.results()) {
    BasicDBList employeeList = (BasicDBList) result.get("docs");
    BasicDBObject employeeDoc = (BasicDBObject) employeeList.get(0);
    String name = employeeDoc.get("name").toString();
    System.out.println(name);
}
} catch (Exception ex) {
    ex.printStackTrace();
}
}

```

See the "resultSet" value in JSON format to understand the output format:

```

[ {
  "_id": "Admin",
  "ageSet": [35.0, 30.0],
  "docs": [ {
    "totalExp": 11.0,
    "age": 35.0,
    "languages": ["english", "hindi"],
    "dept": "Admin",
    "name": "Anna"
  }, {
    "totalExp": 10.0,
    "age": 30.0,
    "languages": ["german", "french", "english", "hindi"],
    "dept": "Admin",
    "name": "Adma"
  } ]
} ]

```

The "resultSet" contains one entry for each group, "ageSet" contains the list of age of each employee of that group, "_id" contains the value of the field that is being used for grouping and "docs" contains data of each employee of that group that can be used in our own code and UI.

Session 9: Indexes

Section 9.1: Index Creation Basics

See the below transactions collection.

```
> db.transactions.insert({ cr_dr : "D", amount : 100, fee : 2 });
> db.transactions.insert({ cr_dr : "C", amount : 100, fee : 2 });
> db.transactions.insert({ cr_dr : "C", amount : 10,  fee : 2 });
> db.transactions.insert({ cr_dr : "D", amount : 100, fee : 4 });
> db.transactions.insert({ cr_dr : "D", amount : 10,  fee : 2 });
> db.transactions.insert({ cr_dr : "C", amount : 10,  fee : 4 });
> db.transactions.insert({ cr_dr : "D", amount : 100, fee : 2 });
```

`getIndexes()` functions will show all the indices available for a collection.

```
db.transactions.getIndexes();
```

Let see the output of above statement.

```
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "documentation_db.transactions"
  }
]
```

There is already one index for transaction collection. This is because MongoDB creates a *unique index* on the `_id` field during the creation of a collection. The `_id` index prevents clients from inserting two documents with the same value for the `_id` field. You cannot drop this index on the `_id` field.

Now let's add an index for `cr_dr` field;

```
db.transactions.createIndex({ cr_dr : 1 });
```

The result of the index execution is as follows.

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

The `createdCollectionAutomatically` indicates if the operation created a collection. If a collection does not exist, MongoDB creates the collection as part of the indexing operation.

Let run `db.transactions.getIndexes()`; again.

```
[
```

```

{
  "v" : 1,
  "key" : {
    "_id" : 1
  },
  "name" : "_id_",
  "ns" : "documentation_db.transactions"
},
{
  "v" : 1,
  "key" : {
    "cr_dr" : 1
  },
  "name" : "cr_dr_1",
  "ns" : "documentation_db.transactions"
}
]

```

Now you see transactions collection have two indices. Default `_id` index and `cr_dr_1` which we created. The name is assigned by MongoDB. You can set your own name like below.

```
db.transactions.createIndex({ cr_dr : -1 }, {name : "index on cr_dr desc"})
```

Now `db.transactions.getIndexes()` ; will give you three indices.

```

[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "documentation_db.transactions"
  },
  {
    "v" : 1,
    "key" : {
      "cr_dr" : 1
    },
    "name" : "cr_dr_1",
    "ns" : "documentation_db.transactions"
  },
  {
    "v" : 1,
    "key" : {
      "cr_dr" : -1
    },
    "name" : "index on cr_dr desc",
    "ns" : "documentation_db.transactions"
  }
]

```

While creating index `{ cr_dr : -1 }` 1 means index will be in ascending order and -1 for descending order.

Version ≥ 2.4

Hashed indexes

Indexes can be defined also as *hashed*. This is more performant on *equality queries*, but is not efficient for *range queries*; however you can define both hashed and ascending/descending indexes on the same field.

```
> db.transactions.createIndex({ cr_dr : "hashed" });

> db.transactions.getIndexes (
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "documentation_db.transactions"
  },
  {
    "v" : 1,
    "key" : {
      "cr_dr" : "hashed"
    },
    "name" : "cr_dr_hashed",
    "ns" : "documentation_db.transactions"
  }
]
```

Section 9.2: Dropping/Deleting an Index

If index name is known,

```
db.collection.dropIndex('name_of_index');
```

If index name is not known,

```
db.collection.dropIndex( { 'name_of_field' : -1 } );
```

Section 9.3: Sparse indexes and Partial indexes

Sparse indexes:

These can be particularly useful for fields that are optional but which should also be unique.

```
{ "_id" : "john@example.com", "nickname" : "Johnnie" }
{ "_id" : "jane@example.com" }
{ "_id" : "julia@example.com", "nickname" : "Jules" }
{ "_id" : "jack@example.com" }
```

Since two entries have no "nickname" specified and indexing will treat unspecified fields as null, the index creation would fail with 2 documents having 'null', so:

```
db.scores.createIndex( { nickname: 1 } , { unique: true, sparse: true } )
```

will let you still have 'null' nicknames.

Sparse indexes are more compact since they skip/ignore documents that don't specify that field. So if you have a collection where only less than 10% of documents specify this field, you can create much smaller indexes - making better use of limited memory if you want to do queries like:

```
db.scores.find({ 'nickname': 'Johnnie' })
```

Partial indexes:

Partial indexes represent a superset of the functionality offered by sparse indexes and should be preferred over sparse indexes. *(New in version 3.2)*

Partial indexes determine the index entries based on the specified filter.

```
db.restaurants.createIndex(  
  { cuisine: 1 },  
  { partialFilterExpression: { rating: { $gt: 5 } } }  
)
```

If `rating` is greater than 5, then `cuisine` will be indexed. Yes, we can specify a property to be indexed based on the value of other properties also.

Difference between Sparse and Partial indexes:

Sparse indexes select documents to index solely based on the existence of the indexed field, or for compound indexes, the existence of the indexed fields.

Partial indexes determine the index entries based on the specified filter. The filter can include fields other than the index keys and can specify conditions other than just an existence check.

Still, a partial index can implement the same behavior as a sparse index

Eg:

```
db.contacts.createIndex(  
  { name: 1 },  
  { partialFilterExpression: { name: { $exists: true } } }  
)
```

Note: Both the *partialFilterExpression* option and the *sparse* option cannot be specified at the same time.

Section 9.4: Get Indices of a Collection

```
db.collection.getIndexes();
```

Output

```
[  
  {  
    "v" : 1,  
    "key" : {  
      "_id" : 1  
    },  
    "name" : "_id_",  
    "ns" : "documentation_db.transactions"  
  },  
  {  
    "v" : 1,  
    "key" : {  
      "cr_dr" : 1  
    },  
    "name" : "cr_dr",  
    "ns" : "documentation_db.transactions"  
  }  
]
```

```

    "name" : "cr_dr_1",
    "ns" : "documentation_db.transactions"
  },
  {
    "v" : 1,
    "key" : {
      "cr_dr" : -1
    },
    "name" : "index on cr_dr desc",
    "ns" : "documentation_db.transactions"
  }
]

```

Section 9.5: Compound

```
db.people.createIndex({name: 1, age: -1})
```

This creates an index on multiple fields, in this case on the `name` and `age` fields. It will be ascending in `name` and descending in `age`.

In this type of index, the sort order is relevant, because it will determine whether the index can support a sort operation or not. Reverse sorting is supported on any prefix of a compound index, as long as the sort is in the reverse sort direction for **all** of the keys in the sort. Otherwise, sorting for compound indexes need to match the order of the index.

Field order is also important, in this case the index will be sorted first by `name`, and within each name value, sorted by the values of the `age` field. This allows the index to be used by queries on the `name` field, or on `name` and `age`, but not on `age` alone.

Section 9.6: Unique Index

```
db.collection.createIndex( { "user_id": 1 }, { unique: true } )
```

enforce uniqueness on the defined index (either single or compound). Building the index will fail if the collection already contains duplicate values; the indexing will fail also with multiple entries missing the field (since they will all be indexed with the value `null`) unless `sparse: true` is specified.

Section 9.7: Singlefield

```
db.people.createIndex({name: 1})
```

This creates an ascending single field index on the field `name`.

In this type of indexes the sort order is irrelevant, because mongo can traverse the index in both directions.

Section 9.8: Delete

To drop an index you could use the index name

```
db.people.dropIndex("nameIndex")
```

Or the index specification document

```
db.people.dropIndex({name: 1})
```


Section 9.9: List

```
db.people.getIndexes()
```

This will return an array of documents each describing an index on the *people* collection

Session 10: Bulk Operations

Section 10.1: Converting a field to another type and updating the entire collection in Bulk

Usually the case when one wants to change a field type to another, for instance the original collection may have "numerical" or "date" fields saved as strings:

```
{
  "name": "Alice",
  "salary": "57871",
  "dob": "1986-08-21"
},
{
  "name": "Bob",
  "salary": "48974",
  "dob": "1990-11-04"
}
```

The objective would be to update a humongous collection like the above to

```
{
  "name": "Alice",
  "salary": 57871,
  "dob": ISODate("1986-08-21T00:00:00.000Z")
},
{
  "name": "Bob",
  "salary": 48974,
  "dob": ISODate("1990-11-04T00:00:00.000Z")
}
```

For relatively small data, one can achieve the above by iterating the collection using a [snapshot](#) with the cursor's [forEach\(\)](#) method and updating each document as follows:

```
db.test.find({
  "salary": { "$exists": true, "$type": 2 },
  "dob": { "$exists": true, "$type": 2 }
}).snapshot().forEach(function(doc) {
  var newSalary = parseInt(doc.salary),
      newDob = new ISODate(doc.dob);
  db.test.updateOne(
    { "_id": doc._id },
    { "$set": { "salary": newSalary, "dob": newDob } }
  );
});
```

Whilst this is optimal for small collections, performance with large collections is greatly reduced since looping through a large dataset and sending each update operation per request to the server incurs a computational penalty.

The [Bulk\(\)](#) API comes to the rescue and greatly improves performance since write operations are sent to the server only once in bulk. Efficiency is achieved since the method does not send every write request to the server (as with the current update statement within the [forEach\(\)](#) loop) but just once in every 1000 requests, thus making updates more efficient and quicker than currently is.

Using the same concept above with the `forEach()` loop to create the batches, we can update the collection in bulk as follows. In this demonstration the `Bulk()` API available in MongoDB versions `>= 2.6` and `< 3.2` uses the `initializeUnorderedBulkOp()` method to execute in parallel, as well as in a nondeterministic order, the write operations in the batches.

It updates all the documents in the clients collection by changing the `salary` and `dob` fields to `numerical` and `datetime` values respectively:

```
var bulk = db.test.initializeUnorderedBulkOp(),
    counter = 0; // counter to keep track of the batch update size

db.test.find({
  "salary": { "$exists": true, "$type": 2 },
  "dob": { "$exists": true, "$type": 2 }
}).snapshot().forEach(function(doc) {
  var newSalary = parseInt(doc.salary),
      newDob = new ISODate(doc.dob);
  bulk.find({ "_id": doc._id }).updateOne({
    "$set": { "salary": newSalary, "dob": newDob }
  });

  counter++; // increment counter
  if (counter % 1000 == 0) {
    bulk.execute(); // Execute per 1000 operations and re-initialize every 1000 update
    statements
    bulk = db.test.initializeUnorderedBulkOp();
  }
});
```

The next example applies to the new MongoDB version `3.2` which has since deprecated the `Bulk()` API and provided a newer set of apis using `bulkWrite()`.

It uses the same cursors as above but creates the arrays with the bulk operations using the same `forEach()` cursor method to push each bulk write document to the array. Because write commands can accept no more than 1000 operations, there's need to group operations to have at most 1000 operations and re-initialise the array when the loop hits the 1000 iteration:

```
var cursor = db.test.find({
  "salary": { "$exists": true, "$type": 2 },
  "dob": { "$exists": true, "$type": 2 }
}),
bulkUpdateOps = [];

cursor.snapshot().forEach(function(doc) {
  var newSalary = parseInt(doc.salary),
      newDob = new ISODate(doc.dob);
  bulkUpdateOps.push({
    "updateOne": {
      "filter": { "_id": doc._id },
      "update": { "$set": { "salary": newSalary, "dob": newDob } }
    }
  });

  if (bulkUpdateOps.length === 1000) {
    db.test.bulkWrite(bulkUpdateOps);
    bulkUpdateOps = [];
  }
});
```

```
if (bulkUpdateOps.length > 0) { db.test.bulkWrite(bulkUpdateOps); }
```