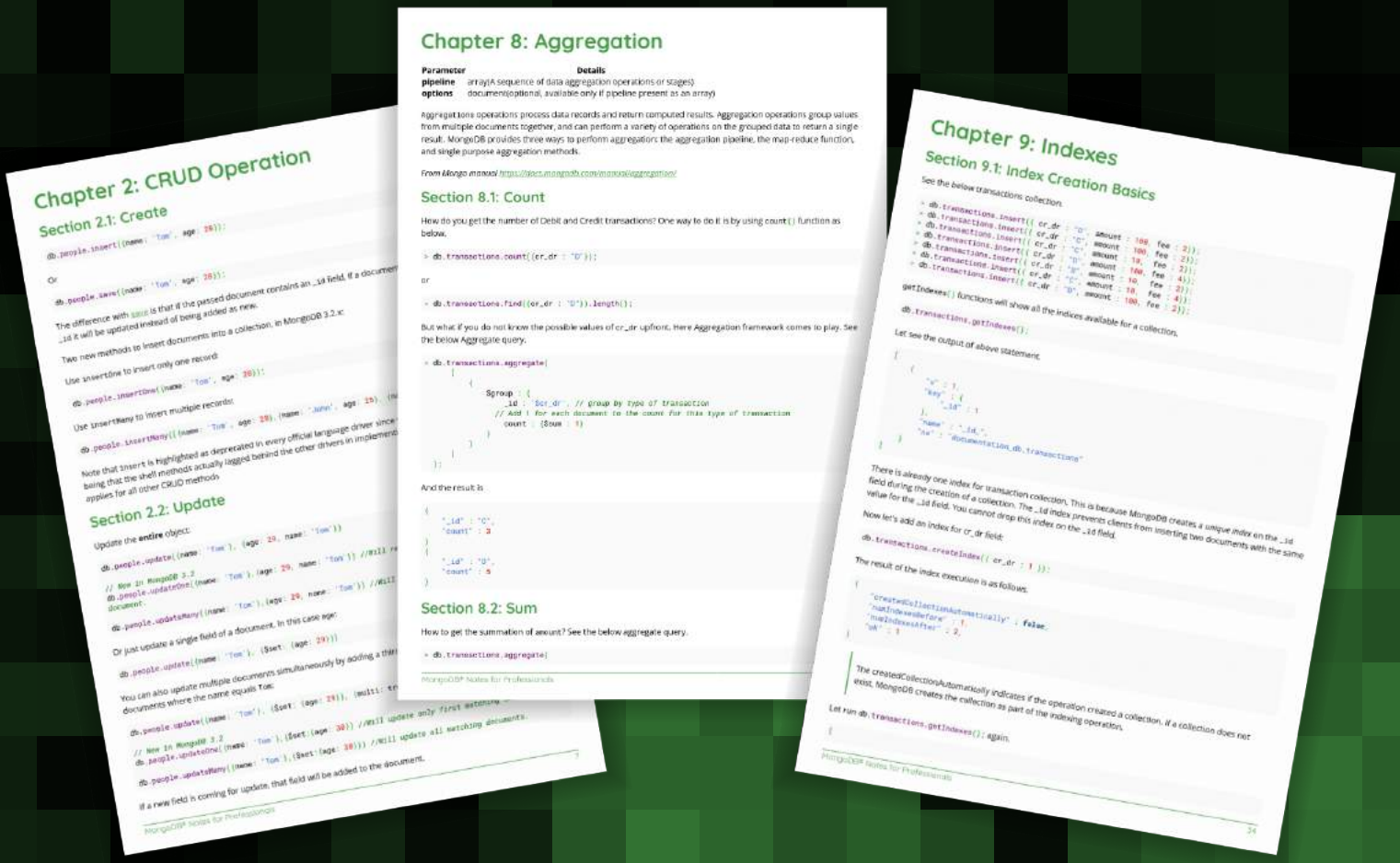
MongoDB®

Notes for Professionals



60+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/MongoDBBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with MongoDB

Version	Release Date
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" : "Val2
1", "key3" : "val31" }
{ "_id" : ObjectId("5790c5e9cae25b3d38c3c7b0"), "key" : "value3", "key2" : "Val2
2", "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	MongoDB 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)

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, their 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 more confident.

- **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 licence 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 **OK**s 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 their. 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.

Chapter 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})

Chapter 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()
```

Chapter 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 and 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.

Chapter 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 overrided** 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
);
```

Chapter 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.

Chapter 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	*Optional.* Configuration options for creating a capped collection or for preallocating space in a new collection.

The fllowing 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 droped 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.

Chapter 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 doucument 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. Minumum 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 form each document in group to an array used to project data in array format, addToSet is simlar 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 opertor 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 opertor 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 opertors: $and, $or, and $not.

Complete refrence: 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 greather 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 greather 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.emplyees.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

Andrew Mao's solution. Average Aggregation Queries in Meteor

```
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"));
        DBObject 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.

Chapter 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_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: Single field

```
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

Chapter 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-intialise 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); }
```

Chapter 11: 2dsphere Index

Section 11.1: Create a 2dsphere Index

`db.collection.createIndex()` method is used to create a `2dsphere` index. The blueprint of a `2dsphere` index :

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

Here, the `location field` is the key and `2dsphere` is the type of the index. In the following example we are going to create a `2dsphre` index in the `places` collection.

```
db.places.insert(
{
  loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
  name: "Central Park",
  category : "Parks"
})
```

The following operation will create `2dsphere` index on the `loc` field of `places` collection.

```
db.places.createIndex( { loc : "2dsphere" } )
```

Chapter 12: Pluggable Storage Engines

Section 12.1: WiredTiger

WiredTiger supports **LSM trees to store indexes**. LSM trees are faster for write operations when you need to write huge workloads of random inserts.

In WiredTiger, there is **no in-place updates**. If you need to update an element of a document, a new document will be inserted while the old document will be deleted.

WiredTiger also offers **document-level concurrency**. It assumes that two write operations will not affect the same document, but if it does, one operation will be rewind and executed later. That's a great performance boost if rewinds are rare.

WiredTiger supports **Snappy and zLib algorithms for compression** of data and indexes in the file system. Snappy is the default. It is less CPU-intensive but have a lower compression rate than zLib.

How to use WiredTiger Engine

```
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>
```

Note:

1. After mongodb 3.2, the default engine is WiredTiger.
2. `newWiredTigerDBPath` should not contain data of another storage engine. To migrate your data, you have to dump them, and re-import them in the new storage engine.

```
mongodump --out <exportDataDestination>
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>
mongorestore <exportDataDestination>
```

Section 12.2: MMAP

MMAP is a pluggable storage engine that was named after the `mmap()` Linux command. It maps files to the virtual memory and optimizes read calls. If you have a large file but needs to read just a small part of it, `mmap()` is much faster then a `read()` call that would bring the entire file to the memory.

One disadvantage is that you can't have two write calls being processed in parallel for the same collection. So, MMAP has collection-level locking (and not document-level locking as WiredTiger offers). This collection-locking is necessary because one MMAP index can reference multiples documents and if those docs could be updated simultaneously, the index would be inconsistent.

Section 12.3: In-memory

All data is stored in-memory (RAM) for faster read/access.

Section 12.4: mongo-rocks

A key-value engine created to integrate with Facebook's RocksDB.

Section 12.5: Fusion-io

A storage engine created by SanDisk that makes it possible to bypass the OS file system layer and write directly to

the storage device.

Section 12.6: TokuMX

A storage engine created by Percona that uses fractal tree indexes.

Chapter 13: Java Driver

Section 13.1: Fetch Collection data with condition

To get data from `testcollection` collection in `testdb` database where `name=dev`

```
import org.bson.Document;
import com.mongodb.BasicDBObject;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;

MongoClient mongoClient = new MongoClient(new ServerAddress("localhost", 27017));
MongoDatabase db = mongoClient.getDatabase("testdb");
MongoCollection<Document> collection = db.getCollection("testcollection");

BasicDBObject searchQuery = new BasicDBObject();
searchQuery.put("name","dev");

MongoCursor<Document> cursor = collection.find(searchQuery).iterator();
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next().toJson());
    }
} finally {
    cursor.close();
}
```

Section 13.2: Create a database user

To create a user **dev** with password **password123**

```
MongoClient mongo = new MongoClient("localhost", 27017);
MongoDatabase db =  mongo.getDatabase("testDB");
Map<String, Object> commandArguments = new BasicDBObject();
commandArguments.put("createUser", "dev");
commandArguments.put("pwd", "password123");
String[] roles = { "readWrite" };
commandArguments.put("roles", roles);
BasicDBObject command = new BasicDBObject(commandArguments);
db.runCommand(command);
```

Section 13.3: Create a tailable cursor

```
find(query).projection(fields).cursorType(CursorType.TailableAwait).iterator();
```

That code applies to the `MongoCollection` class.

CursorType is an enum and it has the following values:

```
Tailable
TailableAwait
```

Corresponding to the old (<3.0) DBCursor addOption Bytes types:

```
Bytes.QUERYOPTION_TAILABLE
Bytes.QUERYOPTION_AWAITDATA
```

Chapter 14: Python Driver

Parameter	Detail
hostX	Optional. You can specify as many hosts as necessary. You would specify multiple hosts, for example, for connections to replica sets.
:portX	Optional. The default value is :27017 if not specified.
database	Optional. The name of the database to authenticate if the connection string includes authentication credentialsIf /database is not specified and the connection string includes credentials, the driver will authenticate to the admin database.
?options	Connection specific options

Section 14.1: Connect to MongoDB using pymongo

```
from pymongo import MongoClient

uri = "mongodb://localhost:27017/"

client = MongoClient(uri)

db = client['test_db']
# or
# db = client.test_db

# collection = db['test_collection']
# or
collection = db.test_collection


collection.save({"hello":"world"})

print collection.find_one()
```

Section 14.2: PyMongo queries

Once you got a `collection` object, queries use the same syntax as in the mongo shell. Some slight differences are:

- every key must be enclosed in brackets. For example:

  ```
  db.find({frequencies: {$exists: true}})
  ```

  becomes in `pymongo` (note the `True` in uppercase):

  ```
  db.find({"frequencies": { "$exists": True }})
  ```

- objects such as object ids or `ISODate` are manipulated using python classes. PyMongo uses its own `ObjectId` class to deal with object ids, while dates use the standard `datetime` package. For example, if you want to query all events between 2010 and 2011, you can do:

  ```
  from datetime import datetime

  date_from = datetime(2010, 1, 1)
  date_to = datetime(2011, 1, 1)
  db.find({ "date": { "$gte": date_from, "$lt": date_to } }):
  ```

Section 14.3: Update all documents in a collection using PyMongo

Let's say you need to add a field to every document in a collection.

```
import pymongo

client = pymongo.MongoClient('localhost', 27017)
db = client.mydb.mycollection

for doc in db.find():
   db.update(
       {'_id': doc['_id']},
       {'$set': {'newField': 10} }, upsert=False, multi=False)
```

The `find` method returns a `Cursor`, on which you can easily iterate over using the `for in` syntax. Then, we call the `update` method, specifying the `_id` and that we add a field (`$set`). The parameters `upsert` and `multi` come from mongodb (see here for more info).

Chapter 15: Mongo as Shards

Section 15.1: Sharding Environment Setup

Sharding Group Members :

For sharding there are three players.

1. Config Server
2. Replica Sets
3. Mongos

   For a mongo shard we need to setup the above three servers.

Config Server Setup : add the following to mongod conf file

```
sharding:
  clusterRole: configsvr
replication:
  replSetName: <setname>
```

run : mongod --config

we can choose config server as replica set or may be a standalone server. Based on our requirement we can choose the best. If config need to run in replica set we need to follow the replica set setup

Replica Setup : Create replica set // Please refer the replica setup

MongoS Setup : Mongos is main setup in shard. Its is query router to access all replica sets

Add the following in mongos conf file

```
    sharding:
      configDB: <configReplSetName>/cfg1.example.net:27017;
```

Configure Shared :

Connect the mongos via shell (mongo --host --port)

1. sh.addShard("/s1-mongo1.example.net:27017")
2. sh.enableSharding("")
3. sh.shardCollection("< database >.< collection >", { < key > : < direction > })
4. sh.status() // To ensure the sharding

Chapter 16: Replication

Section 16.1: Basic configuration with three nodes

The replica set is a group of mongod instances that maintain the same data set.

This example shows how to configure a replica set with three instances on the same server.

Creating data folders

```
mkdir /srv/mongodb/data/rs0-0
mkdir /srv/mongodb/data/rs0-1
mkdir /srv/mongodb/data/rs0-2
```

Starting mongod instances

```
mongod --port 27017 --dbpath /srv/mongodb/data/rs0-0 --replSet rs0
mongod --port 27018 --dbpath /srv/mongodb/data/rs0-1 --replSet rs0
mongod --port 27019 --dbpath /srv/mongodb/data/rs0-2 --replSet rs0
```

Configuring replica set

```
mongo --port 27017    // connection to the instance 27017

rs.initiate();                // initilization of replica set on the 1st node
rs.add("<hostname>:27018")    // adding a 2nd node
rs.add("<hostname>:27019")    // adding a 3rd node
```

Testing your setup

For checking the configuration type `rs.status()`, the result should be like:

```
{
        "set" : "rs0",
        "date" : ISODate("2016-09-01T12:34:24.968Z"),
        "myState" : 1,
        "term" : NumberLong(4),
        "heartbeatIntervalMillis" : NumberLong(2000),
        "members" : [
                {
                        "_id" : 0,
                        "name" : "<hostname>:27017",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        ........................
                },
                {
                        "_id" : 1,
                        "name" : "<hostname>:27018",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        ........................
                },
                {
                        "_id" : 2,
                        "name" : "<hostname>:27019",
```

```
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        .........................
                }
        ],
        "ok" : 1
}
```

Chapter 17: Mongo as a Replica Set

Section 17.1: Mongodb as a Replica Set

We would be creating mongodb as a replica set having 3 instances. One instance would be primary and the other 2 instances would be secondary.

For simplicity, I am going to have a replica set with 3 instances of mongodb running on the same server and thus to achieve this, all three mongodb instances would be running on different port numbers.

In production environment where in you have a dedicated mongodb instance running on a single server you can reuse the same port numbers.

1. Create data directories (path where mongodb data would be stored in a file)

```
- mkdir c:\data\server1 (datafile path for instance 1)
- mkdir c:\data\server2 (datafile path for instance 2)
- mkdir c:\data\server3 (datafile path for instance 3)
```

2. a. Start the first mongod instance

- Open command prompt and type the following press enter.

```
mongod --replSet s0 --dbpath c:\data\server1 --port 37017  --smallfiles --oplogSize 100
```

The above command associates the instance of mongodb to a replicaSet name "s0" and the starts the first instance of mongodb on port 37017 with oplogSize 100MB

2. b. Similarly start the second instance of Mongodb

```
mongod --replSet s0 --dbpath c:\data\server2 --port 37018  --smallfiles --oplogSize 100
```

The above command associates the instance of mongodb to a replicaSet name "s0" and the starts the first instance of mongodb on port 37018 with oplogSize 100MB

2. c. Now start the third instance of Mongodb

```
mongod --replSet s0 --dbpath c:\data\server3 --port 37019  --smallfiles --oplogSize 100
```

The above command associates the instance of mongodb to a replicaSet name "s0" and the starts the first instance of mongodb on port 37019 with oplogSize 100MB

With all the 3 instances started, these 3 instances are independent of each other currently. We would now need to group these instances as a replica set. We do this with the help of a config object.

3.a Connect to any of the mongod servers via the mongo shell. To do that open the command prompt and type.

```
mongo --port 37017
```

Once connected to the mongo shell, create a config object

```
   var config = {"_id":"s0", members[]};
```

this config object has 2 attributes

1. _id: the name of the replica Set ("s0")
2. members: [] (members is an array of mongod instances. lets keep this blank for now, we will add members via the push command.

3.b To Push(add) mongod instances to the members array in the config object. On the mongo shell type

```
config.members.push({"_id":0,"host":"localhost:37017"});
config.members.push({"_id":1,"host":"localhost:37018"});
config.members.push({"_id":2,"host":"localhost:37019"});
```

We assign each mongod instance an _id and an host. _id can be any unique number and the host should be the hostname of the server on which its running followed by the port number.

4. Initiate the config object by the following command in the mongo shell.

```
rs.initiate(config)
```

5. Give it a few seconds and we have a replica set of 3 mongod instances running on the server. type the following command to check the status of the replica set and to identify which one is primary and which one is secondary.

```
rs.status();
```

Section 17.2: Check MongoDB Replica Set states

Use the below command to check the replica set status.

Command : rs.status()

Connect any one of replica member and fire this command it will give the full state of the replica set

Example :

```
{
 "set" : "ReplicaName",
 "date" : ISODate("2016-09-26T07:36:04.935Z"),
 "myState" : 1,
 "term" : NumberLong(-1),
 "heartbeatIntervalMillis" : NumberLong(2000),
 "members" : [
        {
                "_id" : 0,
                "name" : "<IP>:<PORT>,
                "health" : 1,
                "state" : 1,
                "stateStr" : "PRIMARY",
                "uptime" : 5953744,
                "optime" : Timestamp(1474875364, 36),
                "optimeDate" : ISODate("2016-09-26T07:36:04Z"),
                "electionTime" : Timestamp(1468921646, 1),
                "electionDate" : ISODate("2016-07-19T09:47:26Z"),
                "configVersion" : 6,
                "self" : true
        },
        {
                "_id" : 1,
                "name" : "<IP>:<PORT>",
                "health" : 1,
```

```
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 5953720,
                "optime" : Timestamp(1474875364, 13),
                "optimeDate" : ISODate("2016-09-26T07:36:04Z"),
                "lastHeartbeat" : ISODate("2016-09-26T07:36:04.244Z"),
                "lastHeartbeatRecv" : ISODate("2016-09-26T07:36:03.871Z"),
                "pingMs" : NumberLong(0),
                "syncingTo" : "10.9.52.55:10050",
                "configVersion" : 6
        },
        {
                "_id" : 2,
                "name" : "<IP>:<PORT>",
                "health" : 1,
                "state" : 7,
                "stateStr" : "ARBITER",
                "uptime" : 5953696,
                "lastHeartbeat" : ISODate("2016-09-26T07:36:03.183Z"),
                "lastHeartbeatRecv" : ISODate("2016-09-26T07:36:03.715Z"),
                "pingMs" : NumberLong(0),
                "configVersion" : 6
        },
        {
                "_id" : 3,
                "name" : "<IP>:<PORT>",
                "health" : 1,
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 1984305,
                "optime" : Timestamp(1474875361, 16),
                "optimeDate" : ISODate("2016-09-26T07:36:01Z"),
                "lastHeartbeat" : ISODate("2016-09-26T07:36:02.921Z"),
                "lastHeartbeatRecv" : ISODate("2016-09-26T07:36:03.793Z"),
                "pingMs" : NumberLong(22),
                "lastHeartbeatMessage" : "syncing from: 10.9.52.56:10050",
                "syncingTo" : "10.9.52.56:10050",
                "configVersion" : 6
        }
    ],
    "ok" : 1
    }
```

From the above we can know the entire replica set status

Chapter 18: MongoDB - Configure a ReplicaSet to support TLS/SSL

How to configure a ReplicaSet to support TLS/SSL?

We will deploy a 3 Nodes ReplicaSet in your local environment and we will use a self-signed certificate. Do not use a self-signed certificate in PRODUCTION.

How to connect your Client to this ReplicaSet?

We will connect a Mongo Shell.

A description of TLS/SSL, PKI (Public Key Infrastructure) certificates, and Certificate Authority is beyond the scope of this documentation.

Section 18.1: How to configure a ReplicaSet to support TLS/SSL?

Create the Root Certificate

The Root Certificate (aka CA File) will be used to sign and identify your certificate. To generate it, run the command below.

```
openssl req -nodes -out ca.pem -new -x509 -keyout ca.key
```

Keep the root certificate and its key carefully, both will be used to sign your certificates. The root certificate might be used by your client as well.

Generate the Certificate Requests and the Private Keys

When generating the Certificate Signing Request (aka CSR), **input the exact hostname (or IP) of your node in the Common Name (aka CN) field. The others fields must have exactly the same value.** You might need to modify your */etc/hosts* file.

The commands below will generate the CSR files and the RSA Private Keys (4096 bits).

```
openssl req -nodes -newkey rsa:4096 -sha256 -keyout mongodb_node_1.key -out mongodb_node_1.csr
openssl req -nodes -newkey rsa:4096 -sha256 -keyout mongodb_node_2.key -out mongodb_node_2.csr
openssl req -nodes -newkey rsa:4096 -sha256 -keyout mongodb_node_3.key -out mongodb_node_3.csr
```

You must generate one CSR for each node of your ReplicaSet. Remember that the Common Name is not the same from one node to another. Don't base multiple CSRs on the same Private Key.

You must now have 3 CSRs and 3 Private Keys.

```
mongodb_node_1.key - mongodb_node_2.key - mongodb_node_3.key
mongodb_node_1.csr - mongodb_node_2.csr - mongodb_node_3.csr
```

Sign your Certificate Requests

Use the CA File (ca.pem) and its Private Key (ca.key) generated previously to sign each Certificate Request by running the commands below.

```
openssl x509 -req -in mongodb_node_1.csr -CA ca.pem -CAkey ca.key -set_serial 00 -out
```

```
mongodb_node_1.crt
openssl x509 -req -in mongodb_node_2.csr -CA ca.pem -CAkey ca.key -set_serial 00 -out
mongodb_node_2.crt
openssl x509 -req -in mongodb_node_3.csr -CA ca.pem -CAkey ca.key -set_serial 00 -out
mongodb_node_3.crt
```

You must sign each CSR.

Your must now have 3 CSRs, 3 Private Keys and 3 self-signed Certificates. Only the Private Keys and the Certificates will be used by MongoDB.

```
mongodb_node_1.key - mongodb_node_2.key - mongodb_node_3.key
mongodb_node_1.csr - mongodb_node_2.csr - mongodb_node_3.csr
mongodb_node_1.crt - mongodb_node_2.crt - mongodb_node_3.crt
```

Each certificate corresponds to one node. Remember carefully which CN / hostname your gave to each CSR.

Concat each Node Certificate with its key

Run the commands below to concat each Node Certificate with its key in one file (MongoDB requirement).

```
cat mongodb_node_1.key mongodb_node_1.crt > mongodb_node_1.pem
cat mongodb_node_2.key mongodb_node_2.crt > mongodb_node_2.pem
cat mongodb_node_3.key mongodb_node_3.crt > mongodb_node_3.pem
```

Your must now have 3 PEM files.

```
mongodb_node_1.pem - mongodb_node_2.pem - mongodb_node_3.pem
```

Deploy your ReplicaSet

We will assume that your pem files are located in your current folder as well as data/data1, data/data2 and data/data3.

Run the commands below to deploy your 3 Nodes ReplicaSet listening on port 27017, 27018 and 27019.

```
mongod --dbpath data/data_1 --replSet rs0 --port 27017 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_1.pem
mongod --dbpath data/data_2 --replSet rs0 --port 27018 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_2.pem
mongod --dbpath data/data_3 --replSet rs0 --port 27019 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_3.pem
```

You now have a 3 Nodes ReplicaSet deployed on your local environment and all their transactions are encrypted. You cannot connect to this ReplicaSet without using TLS.

Deploy your ReplicaSet for Mutual SSL / Mutual Trust

To force your client to provide a Client Certificate (Mutual SSL), you must add the CA File when running your instances.

```
mongod --dbpath data/data_1 --replSet rs0 --port 27017 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_1.pem --sslCAFile ca.pem
mongod --dbpath data/data_2 --replSet rs0 --port 27018 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_2.pem --sslCAFile ca.pem
mongod --dbpath data/data_3 --replSet rs0 --port 27019 --sslMode requireSSL --sslPEMKeyFile
mongodb_node_3.pem --sslCAFile ca.pem
```

You now have a 3 Nodes ReplicaSet deployed on your local environment and all their transactions are encrypted. You cannot connect to this ReplicaSet without using TLS or without providing a Client Certificate trusted by your CA.

Section 18.2: How to connect your Client (Mongo Shell) to a ReplicaSet?

No Mutual SSL

In this example, we might use the CA File (ca.pem) that you generated during the "*How to configure a ReplicaSet to support TLS/SSL?*" section. We will assume that the CA file is located in your current folder.

We will assume that your 3 nodes are running on mongo1:27017, mongo2:27018 and mongo3:27019. (You might need to modify your */etc/hosts* file.)

From MongoDB 3.2.6, if your CA File is registered in your Operating System Trust Store, you can connect to your ReplicaSet without providing the CA File.

```
mongo --ssl --host rs0/mongo1:27017,mongo2:27018,mongo3:27019
```

Otherwise you must provide the CA File.

```
mongo --ssl --sslCAFile ca.pem --host rs0/mongo1:27017,mongo2:27018,mongo3:27019
```

You are now connected to your ReplicaSet and all the transactions between your Mongo Shell and your ReplicaSet are encrypted.

With Mutual SSL

If your ReplicaSet asks for a Client Certificate, you must provide one signed by the CA used by the ReplicaSet Deployment. The steps to generate the Client Certificate are almost the same as the ones to generate the Server Certificate.

Indeed, you just need to modify the Common Name Field during the CSR creation. Instead of providing 1 Node Hostname in the Common Name Field, **you need to provide all the ReplicaSet Hostnames separated by a comma**.

```
openssl req -nodes -newkey rsa:4096 -sha256 -keyout mongodb_client.key -out mongodb_client.csr
...
Common Name (e.g. server FQDN or YOUR name) []: mongo1,mongo2,mongo3
```

You might face the Common Name size limitation if the Common Name field is too long (more than 64 bytes long). To bypass this limitation, you must use the SubjectAltName when generating the CSR.

```
openssl req -nodes -newkey rsa:4096 -sha256 -keyout mongodb_client.key -out mongodb_client.csr -
config <(
cat <<-EOF
[req]
default_bits = 4096
prompt = no
default_md = sha256
req_extensions = req_ext
distinguished_name = dn

[ dn ]
CN = .
```

```
[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = mongo1
DNS.2 = mongo2
DNS.3 = mongo3
EOF
)
```

Then you sign the CSR using the CA certificate and key.

```
openssl x509 -req -in mongodb_client.csr  -CA ca.pem -CAkey ca.key -set_serial 00 -out
mongodb_client.crt
```

Finally, you concat the key and the signed certificate.

```
cat mongodb_client.key mongodb_client.crt > mongodb_client.pem
```

To connect to your ReplicaSet, you can now provide the newly generated Client Certificate.

```
mongo --ssl --sslCAFile ca.pem --host rs0/mongo1:27017,mongo2:27018,mongo3:27019 --sslPEMKeyFile
mongodb_client.pem
```

You are now connected to your ReplicaSet and all the transactions between your Mongo Shell and your ReplicaSet are encrypted.

Chapter 19: Authentication Mechanisms in MongoDB

Authentication is the process of verifying the identity of a client. When access control, i.e. authorization, is enabled, MongoDB requires all clients to authenticate themselves in order to determine their access.

MongoDB supports a number of authentication mechanisms that clients can use to verify their identity. These mechanisms allow MongoDB to integrate into your existing authentication system.

Section 19.1: Authentication Mechanisms

MongoDB supports multiple authentication mechanisms.

Client and User Authentication Mechanisms

- SCRAM-SHA-1
- X.509 Certificate Authentication
- MongoDB Challenge and Response (MONGODB-CR)
- LDAP proxy authentication, and
- Kerberos authentication

Internal Authentication Mechanisms

- Keyfile
- X.509

Chapter 20: MongoDB Authorization Model

Authorization is the basically verifies user privileges. MongoDB support different kind of authorization models. 1. **Role base access control**
 Role are group of privileges, actions over resources. That are gain to users over a given namespace (Database). Actions are performs on resources. Resources are any object that hold state in database.

Section 20.1: Build-in Roles

Built-in database user roles and database administration roles roles exist in each database.

Database User Roles

1. `read`
2. `readwrite`

Chapter 21: Configuration

Parameter	Default
systemLog.verbosity	0
systemLog.quiet	false
systemLog.traceAllExceptions	false
systemLog.syslogFacility	user
systemLog.path	-
systemLog.logAppend	false
systemLog.logRotate	rename
systemLog.destination	stdout
systemLog.timeStampFormat	iso8601-local
systemLog.component.accessControl.verbosity	0
systemLog.component.command.verbosity	0
systemLog.component.control.verbosity	0
systemLog.component.ftdc.verbosity	0
systemLog.component.geo.verbosity	0
systemLog.component.index.verbosity	0
systemLog.component.network.verbo	0
systemLog.component.query.verbosity	0
systemLog.component.replication.verbosity	0
systemLog.component.sharding.verbosity	0
systemLog.component.storage.verbosity	0
systemLog.component.storage.journal.verbosity	0
systemLog.component.write.verbosity	0
processManagement.fork	false
processManagement.pidFilePath	none
net.port	27017
net.bindIp	0.0.0.0
net.maxIncomingConnections	65536
net.wireObjectCheck	true
net.ipv6	false
net.unixDomainSocket.enabled	true
net.unixDomainSocket.pathPrefix	/tmp
net.unixDomainSocket.filePermissions	0700
net.http.enabled	false
net.http.JSONPEnabled	false
net.http.RESTInterfaceEnabled	false
net.ssl.sslOnNormalPorts	false
net.ssl.mode	disabled
net.ssl.PEMKeyFile	none
net.ssl.PEMKeyPassword	none
net.ssl.clusterFile	none
net.ssl.clusterPassword	none
net.ssl.CAFile	none
net.ssl.CRLFile	none
net.ssl.allowConnectionsWithoutCertificates	false
net.ssl.allowInvalidCertificates	false
net.ssl.allowInvalidHostnames	false
net.ssl.disabledProtocols	none
net.ssl.FIPSMode	false

Section 21.1: Starting mongo with a specific config file

Using the `--config` flag.

```
$ /bin/mongod --config /etc/mongod.conf
$ /bin/mongos --config /etc/mongos.conf
```

Note that `-f` is the shorter synonym for `--config`.

Chapter 22: Backing up and Restoring Data

Section 22.1: Basic mongodump of local default mongod instance

```
mongodump --db mydb --gzip --out "mydb.dump.$(date +%F_%R)"
```

This command will dump a bson gzipped archive of your local mongod 'mydb' database to the 'mydb.dump.{timestamp}' directory

Section 22.2: Basic mongorestore of local default mongod dump

```
mongorestore --db mydb mydb.dump.2016-08-27_12:44/mydb --drop --gzip
```

This command will first drop your current 'mydb' database and then restore your gzipped bson dump from the 'mydb mydb.dump.2016-08-27_12:44/mydb' archive dump file.

Section 22.3: mongoimport with JSON

Sample zipcode dataset in zipcodes.json stored in c:\Users\yc03ak1\Desktop\zips.json

```
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" :
"MA" }
{ "_id" : "01002", "city" : "CUSHMAN", "loc" : [ -72.51564999999999, 42.377017 ], "pop" : 36963,
"state" : "MA" }
{ "_id" : "01005", "city" : "BARRE", "loc" : [ -72.10835400000001, 42.409698 ], "pop" : 4546,
"state" : "MA" }
{ "_id" : "01007", "city" : "BELCHERTOWN", "loc" : [ -72.41095300000001, 42.275103 ], "pop" :
10579, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" :
"MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [ -72.188455, 42.116543 ], "pop" : 3706, "state" :
"MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" :
"MA" }
```

to import this data-set to the database named "test" and collection named "zips"

```
C:\Users\yc03ak1>mongoimport --db test --collection "zips" --drop --type json --host
"localhost:47019"  --file "c:\Users\yc03ak1\Desktop\zips.json"
```

- --db : name of the database where data is to be imported to
- --collection: name of the collection in the database where data is to be improted
- --drop : drops the collection first before importing
- --type : document type which needs to be imported. default JSON
- --host : mongodb host and port on which data is to be imported.
- --file : path where the json file is

output :

```
2016-08-10T20:10:50.159-0700    connected to: localhost:47019
```

```
2016-08-10T20:10:50.163-0700    dropping: test.zips
2016-08-10T20:10:53.155-0700    [################........] test.zips    2.1 MB/3.0 MB (68.5%)
2016-08-10T20:10:56.150-0700    [########################] test.zips    3.0 MB/3.0 MB (100.0%)
2016-08-10T20:10:57.819-0700    [########################] test.zips    3.0 MB/3.0 MB (100.0%)
2016-08-10T20:10:57.821-0700    imported 29353 documents
```

Section 22.4: mongoimport with CSV

Sample test dataset CSV file stored at the location c:\Users\yc03ak1\Desktop\testing.csv

```
_id    city    loc    pop    state
1    A    [10.0, 20.0]    2222    PQE
2    B    [10.1, 20.1]    22122    RW
3    C    [10.2, 20.0]    255222    RWE
4    D    [10.3, 20.3]    226622    SFDS
5    E    [10.4, 20.0]    222122    FDS
```

to import this data-set to the database named "test" and collection named "sample"

```
C:\Users\yc03ak1>mongoimport --db test --collection "sample" --drop --type csv --headerline --host
"localhost:47019"  --file "c:\Users\yc03ak1\Desktop\testing.csv"
```

- --headerline : use the first line of the csv file as the fields for the json document

output :

```
2016-08-10T20:25:48.572-0700    connected to: localhost:47019
2016-08-10T20:25:48.576-0700    dropping: test.sample
2016-08-10T20:25:49.109-0700    imported 5 documents
```

OR

```
C:\Users\yc03ak1>mongoimport --db test --collection "sample" --drop --type csv --fields
_id,city,loc,pop,state --host "localhost:47019"  --file "c:\Users\yc03ak1\Desktop\testing.csv"
```

- --fields : comma seperated list of fields which needs to be imported in the json document. Output:

```
2016-08-10T20:26:48.978-0700    connected to: localhost:47019
2016-08-10T20:26:48.982-0700    dropping: test.sample
2016-08-10T20:26:49.611-0700    imported 6 documents
```

Chapter 23: Upgrading MongoDB version

How to update the version of MongoDB on your machine on different platforms and versions.

Section 23.1: Upgrading to 3.4 on Ubuntu 16.04 using apt

You must have 3.2 to be able to upgrade to 3.4. This example assumes you are using `apt`.

1. `sudo service mongod stop`
2. `sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 0C49F3730359A14518585931BC711F9BA15703C6`
3. `echo "deb [ arch=amd64,arm64 ] http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.4.list`
4. `sudo apt-get update`
5. `sudo apt-get upgrade`
6. `sudo service mongod start`

Ensure the new version is running with `mongo`. The shell will print out the MongoDB server version that should be 3.4 now.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Abdul Rehman Sayed	Chapter 1
ADIMO	Chapter 16
Antti_M	Chapter 23
Ashari	Chapter 1
Avindu Hewa	Chapter 4
bappr	Chapter 18
Batsu	Chapter 9
chridam	Chapter 10
Constantin Guay	Chapters 12 and 9
Derlin	Chapter 14
dev □	Chapter 13
Emil Burzo	Chapter 13
fracz	Chapters 2 and 3
grape	Chapter 8
gypsyCoder	Chapter 11
HoefMeistert	Chapter 8
ipip	Chapter 1
Ishan Soni	Chapter 2
Jain	Chapter 2
jerry	Chapter 2
JohnnyHK	Chapter 2
Juan Carlos Farah	Chapter 9
Kelum Senanayake	Chapter 2
KrisVos130	Chapter 2
Kuhan	Chapter 6
Lakmal Vithanage	Chapters 2 and 8
LoicM	Chapter 8
Luzan Baral	Chapter 19
manetsus	Chapters 1 and 8
Marco	Chapter 2
Matt Clark	Chapter 21
Nic Cottrell	Chapter 9
Niroshan Ranapathi	Chapters 19 and 20
oggo	Chapter 4
Prosen Ghosh	Chapters 1, 2, 4 and 7
RaR	Chapters 8 and 9
Renukaradhya	Chapters 1 and 2
Rotem	Chapter 2
Sean Reilly	Chapter 1
Selva Kumar	Chapter 15
sergiuz	Chapter 14
Shrabanee	Chapter 2
SommerEngineering	Chapter 4
sstyvane	Chapter 2
steveinatorx	Chapter 8
Thomas Bormans	Chapter 2
tim	Chapter 12
titogeo	Chapters 1, 8 and 9
Tomás Cañibano	Chapters 2 and 9

user641887	Chapters 22 and 17
WAF	Chapter 1
yellowB	Chapter 5
Zanon	Chapter 12

You may also like