Basic Notions on MongoDB Administration

Dimitris Diochnos

Edinburgh, December 12, 2014

Abstract

The aim of this document is to give a brief overview of MongoDB as well as try to cover basic CRUD (Create, Retrieve, Update, Delete) operations using the Mongo shell.

1 Introduction

MongoDB is a NoSQL document-oriented database. BSON (*binary JSON*), which is a binary serialization format, is used to store documents and make remote procedure calls in MongoDB. The BSON specification is located at `http://bsonspec.org`.

1.1 Data Types

The traditional JSON data types are available in MongoDB as well. These are the following.

- strings
- numbers
- booleans
- null
- arrays
- objects/documents

Apart from the above, MongoDB allows the following data types as well.

ObjectId. The ObjectId data type. Every MongoDB document has an `_id` property which is used as the primary key for indexing. Unless specified otherwise by programmers, this is of type ObjectId.

Date. A data type for storing dates. Note that such a data type is not supported on JSON.

BinData. A data type for binary data; e.g. pictures, videos, etc.

1.2 Storage

Consider the following JSON document

```
{
  "day": 12,
  "month": "December"
}
```

Its serialization as a BSON document looks like the one below.

doc size	dt_1 : int32	day\0	12	dt_2 : string	month\0	9	December\0	■
32 bits	8 bits	4 bytes	32 bits	8 bits	6 bytes	32 bits	9 bytes	EOD

1.2.1 Size of Documents

Default. By default MongoDB supports documents of size up to 16 MBytes.

GridFS. GridFS can be used for storing documents of larger size; e.g. 100 TBytes - i.e. documents that can not be stored even on a single server. No limit on document size.

1.3 Communication

The basic communication is shown in Figure 1. Client applications are using a *Mongo-driver* that is responsible

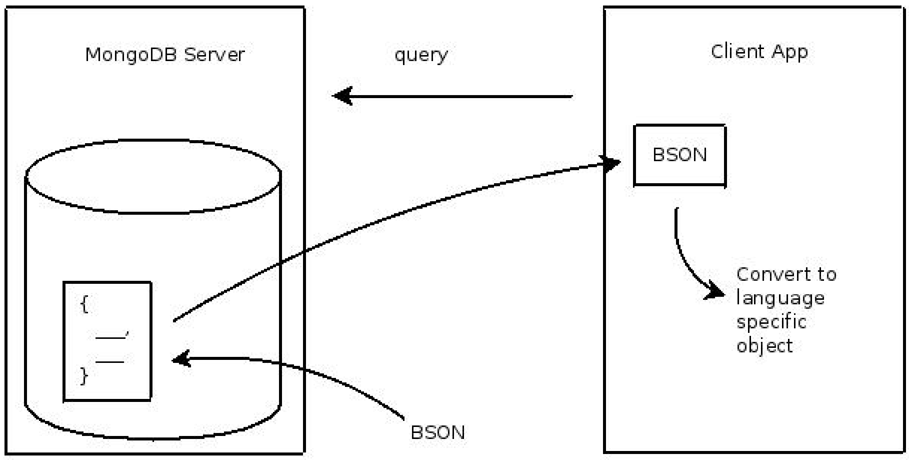

Figure 1: Communication between client applications and the MongoDB server.

for the communication and the translation of the requests between the client and the server.

1.4 Hierarchy

We can observe the following hierarchy.

mongo cluster
↪ databases
↪ collections
↪ documents (BSON/JSON)

2 Obtaining and Installing MongoDB

We can obtain the latest version of MongoDB from the MongoDB website in the address

`http://www.mongodb.org/downloads` .

Once we download and store MongoDB in an appropriate directory (e.g. under `/opt/mongo/`), we just need to make sure that the `bin` subdirectory is added to our path for convenience.

3 Learning MongoDB and MongoDB Documentation

MongoDB has an extensive documentation online: `http://docs.mongodb.org/manual/` .

3.1 MongoDB University

Moreover, there are online courses offered by the MongoDB university (http://university.mongodb.com). Some of the above courses are free and some are not. They typically require about 6-8 hours of work per week and in the end, upon successful completion, one can also obtain a certificate issued by the MongoDB university, which can be a nice addition to one's resume.

4 The MongoDB Daemon

We can start the MongoDB server with the following command.

```
$ mongod
```

The default port where the server is running is 27017. The default directory where the database stores content is under `/data/db/`. These options can be parameterised; for example the command

```
$ mongod --dbpath /home/user/db --port 9001 --rest --httpinterface
```

starts the MongoDB server using the directory `/home/user/db`, listening to port 9001, and moreover enables a simple REST api that can be used in conjunction with a monitoring http interface. Note that the http interface is available on the port where the server is running plus 1000; thus in this particular case, one should point the browser at the address `http://localhost:10001`. For more information on the parameters of the MongoDB daemon we can give the following command.

```
$ mongod --help
```

5 The Mongo Shell

The Mongo shell is used for interacting with a MongoDB server and perform administrative tasks. The Mongo shell is interpreting Javascript code. The following code opens a Mongo shell without connecting to any database, sets a variable `a` to be equal to the current date, and then prints the ISO date string, as well as the number of milliseconds that have elapsed since Jan 1, 1970.

```
$ mongo --nodb
MongoDB shell version: 2.6.5
> var a = new Date()
> a
ISODate("2014-12-11T22:35:37.216Z")
> a.getTime()
1418337337216
> ^d
bye
$
```

5.1 Connecting to a MongoDB Server

Assuming that a MongoDB server is running on host with IP `x.y.z.w` one can connect through the Mongo shell on that server using the command

```
$ mongo x.y.z.w
```

In particular, we can also specify the database to which we want to connect to and perhaps load some Javascript files where we have defined some functions that can be useful for performing various administrative operations. The code below gives an example where we connect to the database `agents` on `localhost` at port 9001 and also load the code from the files `file1.js` and `file2.js`.

```
$ cat file1.js
function f () {
   // Note that we use print instead of console.log
   print("Hello from function f defined in file file1.js!");
}
$ cat file2.js
function successor(a) {
   return a+1;
}
$ mongo --shell localhost:9001/agents file1.js file2.js
MongoDB shell version: 2.6.5
connecting to: localhost:9001/agents
type "help" for help
loading file: file1.js
loading file: file2.js
> f()
Hello from function f defined in file file1.js!
> successor(12)
13
> ^d
bye
$
```

For more information on the parameters that we can pass on `mongo` we can use

```
$ mongo --help
```

5.2 Basic Orientation and Getting Help

Let us attempt to acquire a Mongo shell again on localhost.

```
$ mongo --shell --port 9001
MongoDB shell version: 2.6.5
connecting to: 127.0.0.1:9001/test
type "help" for help
>
```

5.2.1 Basic Orientation

The comments on the lines give a brief explanation of the command. Note that the following commands are valid since the comments are valid Javascript comments.

```
> show dbs        // present all the database names; test not created yet
admin  (empty)
local  0.078GB
> db              // show the database that we are using
test
```

```
> use local           // use the database named `local'
switched to db local
> show collections    // show the collections of this database
startup_log
system.indexes
> db.startup_log.count()   // count the number of docs in the collection `startup_log'
1
> db.startup_log.find()    // find all documents in the collection startup_log
{ ... omitting the output ... }
> db.startup_log.find().pretty()   // present the result in a user-friendly way
{
   ... omitting the output again ...
}
> db.startup_log.findOne()   // find just one document; it will be prettified
{
   ... omitting the output again ...
}
> cls   // clear screen
```

5.2.2 Getting Help

We can attempt to get some help with the following command.

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        ... omitting the rest of the output ...
>
```

The first two lines of the output indicate two more *help* commands. One providing help on commands at the level of databases, and one more providing help on the commands at the level of collections. Below we switch back to our `test` database and then we request help on the methods that are available on the database and a collection called `mycollection`.

```
> use test
> db.help()  // help for commands that we can use with databases
        ... omitting the output ...
> db.mycollection.help() // help for commands that we can use with collections
        ... omitting the output ...
> ^d
```

6 Importing Data

We can import JSON, CSV and TSV files into a collection of MongoDB using the command `mongoimport`. We can specify the database where the data is going to be imported with the `--db` parameter, while the collection is selected with the `--collection` parameter. Moreover, with the `--stopOnError` parameter we can request the import operation to stop on the first error. Finally, we can specify the host and the port with the `--host` and `--port` parameters.

```
$ cat fall2014.json
{"date": "10/31/2014", "speaker": "Paolo", "kind": "talk", "title": ... }
```

```
{"date": "11/07/2014", "speaker": "Pavlos", "kind": "talk", "title": ... }
{"date": "11/14/2014", "speaker": "Nico", "kind": "talk", "title": ... }
{"date": "11/21/2014", "speaker": "Orestis", "kind": "talk", "title": ... }
{"date": "11/28/2014", "speaker": "Steven", "kind": "talk", "title": ... }
{"date": "12/12/2014", "speaker": "Dimitris", "kind": "tutorial", "title": ... }
$ mongoimport --host localhost --port 9001 --stopOnError --db agents \
> --collection fall2014 < fall2014_withIDs.json
connected to: localhost:9001
2014-12-12T08:58:17.979+0000 imported 6 objects
$
```

7 Dropping Collections and Databases

Once we connect to a MongoDB server using the Mongo shell, we can drop collections or even entire databases using the following commands.

```
> use agents
switched to db agents
> db.fall2014.drop()     // drops collection `fall2014'
true
> db.dropDatabase()      // drops the entire `agents' database
{ "dropped" : "agents", "ok" : 1 }
>
```

8 Queries

We can specify the properties of our queries using JSON documents.

8.1 Limit, Skip and Sort

Apart from specifying the actual queries, there are times that we want to *limit* our results, *skip* some results, or even *sort* them in a specific way. We will use the 'fall2014' collection that we created earlier. Recall that we can see all the documents in the collection using the command

```
> db.fall2014.find()   // db.fall2014.find().pretty() for prettified output
```

We can now use `limit`, `skip`, and *sort* all at the same time. For example, suppose we want to sort the results in inverse chronological order, skip the first one, and limit the output to 2 documents. Then the following command suffices.

```
> db.fall2014.find().limit(2).skip(1).sort({"date": -1})
{ "_id" : 5, "date" : "11/28/2014", "speaker" : "Steven", "kind" : "talk", ... }
{ "_id" : 3, "date" : "11/21/2014", "speaker" : "Orestis", "kind" : "talk", ... }
>
```

Notice that for sorting we passed a JSON object specifying the field (`date`) that we want to use for sorting the data, and moreover with the -1 we indicated that we want the results to be sorted in *descending* order. We could use 1 to sort the documents in *ascending* order.

Finally, note that the above example is for demonstration purposes only. We have actually sorted the documents in descending order while we were using *lexicographical ordering*. We will verify this soon below.

8.2 Simple Queries

Assume we want to find all the documents where the speaker is Pavlos. We could perform the following query.

```
> db.fall2014.find({"speaker": "Pavlos"})
{ "_id" : 1, "date" : "11/07/2014", "speaker" : "Pavlos", "kind" : "talk", ... }
> db.fall2014.find({"speaker": "Pavlos"}).count()      // use count to enumerate
1
>
```

Note that above we used double quotes for the word speaker. We could have omitted the quotes and we would still get the same result. Actually, we can use count directly.

```
> db.fall2014.count({speaker: "Pavlos"})
1
>
```

However, it is a good practice to include the quotation marks. For example, assume we can find in our collection a document that looks like the one below.

```
{
   "_id" : 8,
   "date" : "12/12/2014",
   "a" : {
      "b" : "blurb",
      "c" : {
         "k" : 1
      }
   }
}
```

Then, in order to achieve an actual match using *properties of subdocuments* we need to list those properties inside double quotes. Thus, we can identify the document in the collection with a find command that has the following format.

```
> db.fall2014.find({"a.c.k": 1})
{ "_id" : 8, "date" : "12/12/2014", "a" : { "b" : "blurb", "c" : { "k" : 1 } } }
```

8.3 Operators for Queries

In order to perform more complex queries we can use the operators that are shown in Table 1.

Below are some example queries.

```
> db.fall2014.find({"date": {$exists: true}}).count() // count docs where date exists
6
> db.fall2014.find({"date": {$exists: false}}).count() // ... where date does not exist
0
> db.fall2014.find({"date": {$lt : "11"}}) // find docs where date is less than "11"
{ "_id" : 2, "date" : "10/31/2014", "speaker" : "Paolo", "kind" : "talk", ... }
>
> // Find docs where the date is less than "11" or greater than or equal to "12"
> db.fall2014.find({$or: [{"date": {$lt: "11"}}, {"date": {$gte: "12"}}] })
```

operator	meaning
$gte	greater than or equal
$lte	less than or equal
$gt	greater than
$lt	less than
$or	logical or
$and	logical and (implicit by commas in JSON)
$in	∈ operator
$nin	∉ operator
$exists	match docs where a property exists

Table 1: Operators used for queries.

```
{ "_id" : 2, "date" : "10/31/2014", "speaker" : "Paolo", "kind" : "talk", ... }
{ "_id" : 4, "date" : "12/12/2014", "speaker" : "Dimitris", "kind" : "tutorial", ... }
>
> db.fall2014.find({$and: [{"date": {$lt: "11"}}, {"date": {$gte: "12"}}] }) // and ...
>
> db.fall2014.find({date: {$in : ["11/21/2014", "11/28/2014"]}}) // $in operator
{ "_id" : 3, "date" : "11/21/2014", "speaker" : "Orestis", "kind" : "talk", ... }
{ "_id" : 5, "date" : "11/28/2014", "speaker" : "Steven", "kind" : "talk", "... }
>
> // Common types: double: 1, string: 2, object: 3, array: 4, boolean: 8, int32: 16
> db.fall2014.find({"date": {$type: 2}}).count()
6
> db.fall2014.find({"date": {$type: 1}}).count()
0
>
```

8.4 Projections

In some cases we do not want the full output, but rather certain fields from the search results. We can pass a second argument to the `find` method thus indicating with a 0/1 entry whether we want a specific field to appear among the results. Below we have an example with the dates and the speakers in each case.

```
> db.fall2014.find({}, {date: 1, speaker: 1, _id: 0}).sort({"date": 1})
{ "date" : "10/31/2014", "speaker" : "Paolo" }
{ "date" : "11/07/2014", "speaker" : "Pavlos" }
{ "date" : "11/14/2014", "speaker" : "Nico" }
{ "date" : "11/21/2014", "speaker" : "Orestis" }
{ "date" : "11/28/2014", "speaker" : "Steven" }
{ "date" : "12/12/2014", "speaker" : "Dimitris" }
>
```

8.5 Results in an Array

The method `find` typically returns multiple results that satisfy our query criteria. In order to get the results of a `find` method as an array we can use the `toArray` method together with `find`. Thus we can store the results in a variable and perhaps iterate through them on the shell. Below we have an example.

```
> var res = db.fall2014.find({}, {"_id": 0, "date": 1, "speaker": 1})
> var res = db.fall2014.find({}, {"_id": 0, "date": 1,
... "speaker": 1}).sort({"date": 1}).toArray()
> res.length
6
> res[3]
{ "date" : "11/21/2014", "speaker" : "Orestis" }
>
```

8.6 Explanations and Indices

Suppose we want to find the document that has `_id` equal to 5. Then we can give the following command.

```
> db.fall2014.find({"_id": 5})
{ "_id" : 5, "date" : "11/28/2014", "speaker" : "Steven", "kind" : "talk", ... }
>
```

MongoDB provides an *explanation service* for the queries that we are making, and it can be called with the function `explain`.

```
> db.fall2014.find({"_id": 5}).explain()
{
   "cursor" : "IDCursor",
   "n" : 1,
   "nscannedObjects" : 1,
   "nscanned" : 1,
   // omitting stuff ...
}
> db.fall2014.find({"date": "11/28/2014"}).explain()
{
   "cursor" : "BasicCursor",
   "n" : 1,
   "nscannedObjects" : 6,
   "nscanned" : 6,
   // omitting stuff ...
}
>
```

In the output above perhaps the most important entries are `cursor`, `n`, `nscanned` and `nscannedObjects`. Cursors are the objects that are returned from search queries and essentially they are used so that we can iterate through them and view all the matching results. More information is available on the MongoDB documentation website. Regarding the rest we have: `n` is the number of documents that match the query selection criteria, `nscanned` is the total number of index entries scanned (or documents for a collection scan), and `nscannedObjects` is the total number of documents scanned.

Note that in the second case we have just a basic cursor, and thus all the documents in the collection had to be scanned. We can however create an index on `date` using the command `ensureIndex`. The parameter of `ensureIndex` is an expression which indicates which fields we want to include for this index as well as whether we want to sort the documents in ascending or descending order. Ascending and descending order is indicated with 1 and -1 just like in the case of sorting that we saw earlier. Optionally, we can pass a second argument with additional options. Thus the following command creates an index on the property `date` in descending order and is given an appropriate name. Then we verify that the index has been created using the `getIndices` method.

```
> db.fall2014.ensureIndex({"date": -1}, {"name": "dates in descending order"})
{
   "createdCollectionAutomatically" : false,
   "numIndexesBefore" : 1,
   "numIndexesAfter" : 2,
   "ok" : 1
}
> db.fall2014.getIndices()
[
   {
      "v" : 1,
      "key" : { "_id" : 1 },
      "name" : "_id_",
      "ns" : "agents.fall2014"
   },
   {
      "v" : 1,
      "key" : { "date" : -1 },
      "name" : "dates in descending order",
      "ns" : "agents.fall2014"
   }
]
>
```

If we now attempt to get an explanation for our query we can see that a B-tree index has been generated (look at the `cursor`) and only one document was actually scanned.

```
> db.fall2014.find({"date": "11/28/2014"}).explain()
{
   "cursor" : "BtreeCursor date_1",
   "n" : 1,
   "nscannedObjects" : 1,
   "nscanned" : 1,
   // omitting stuff ...
}
>
```

9 Insertions

We can insert documents in our collection using a command of the following form.

```
db.<collection name>.insert(<document>)
```

Below we give an example.

```
> var a = {"_id": 6, "date": "12/19/2014", "speaker": null}
>
> db.fall2014.insert(a);
WriteResult({ "nInserted" : 1 })
> db.getLastError()               // Returns null for no error; string otherwise
null
```

```
> db.fall2014.find({})
{ "_id" : 0, "date" : "11/14/2014", "speaker" : "Nico", "kind" : "talk", ... }
{ "_id" : 1, "date" : "11/07/2014", "speaker" : "Pavlos", "kind" : "talk", ... }
{ "_id" : 2, "date" : "10/31/2014", "speaker" : "Paolo", "kind" : "talk", ... }
{ "_id" : 3, "date" : "11/21/2014", "speaker" : "Orestis", "kind" : "talk", ... }
{ "_id" : 4, "date" : "12/12/2014", "speaker" : "Dimitris", "kind" : "tutorial", ...}
{ "_id" : 5, "date" : "11/28/2014", "speaker" : "Steven", "kind" : "talk", ... }
{ "_id" : 6, "date" : "12/19/2014", "speaker" : null }
>
```

Note that MongoDB did not complain at all for adding the last document into the collection since it is *schema-less.*

10 Updates

We can update entries using commands of the following form.

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

- `query` refers to the selection criteria for the update.
- If the `<update>` document contains only `field:value` expressions then *one* matching document will be replaced entirely. Otherwise, if *only* modifiers are present, then only the relevant fields will be updated.
- If `upsert` is `true` and no document matches the query criteria, `update()` inserts a single document.
- If `multi` is set to `true`, the `update()` method updates all documents that meet the `<query>` criteria.
- The option `writeConcern` is beyond the scope of this tutorial.

For example the following command adds a comment to our recent entry.

```
> db.fall2014.update({_id: 6}, {$set: {"comments": ["Happy holiday!"]}},
... {upsert: false, multi: false})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.fall2014.find({_id: 6})
{ "_id" : 6, "date" : "12/19/2014", "speaker" : null, "comments" : [ "Happy holiday!" ] }
>
```

10.1 Operators for Updates

In order to perform more complex updates we can use the operators that are shown in Table 2.

Below are some example updates.

operator	meaning
$set	set a field to have a specific value
$unset	unset a field
$push	push a value into an array
$addToSet	treat the array as a set and attempt to add the element
$pop	pop a value

Table 2: Operators used for updates.

```
> db.fall2014.update({_id: 6}, {$push: {"comments": "another comment!"}})  // push
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.fall2014.find({_id: 6})
{ "_id" : 6, ..., "comments" : [ "Happy holiday!", "another comment!" ] }
>
> db.fall2014.update({_id: 6}, {$pop: {"comments": 1}})                    // pop
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.fall2014.find({_id: 6})
{ "_id" : 6, ..., "comments" : [ "Happy holiday!" ] }
>
> // addToSet below
> db.fall2014.update({_id: 6}, {$addToSet: {"comments": "another comment!"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.fall2014.find({_id: 6})
{ "_id" : 6, ..., "comments" : [ "Happy holiday!", "another comment!" ] }
> db.fall2014.update({_id: 6}, {$addToSet: {"comments": "another comment!"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
> // `nModified' indicates that no documents were affected after the second call
> db.fall2014.find({_id: 6})
{ "_id" : 6, ..., "comments" : [ "Happy holiday!", "another comment!" ] }
>
> db.fall2014.update({_id: 6}, {$unset: {"comments": true}})             // unset
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.fall2014.find({_id: 6})
{ "_id" : 6, "date" : "12/19/2014", "speaker" : null }
>
```

11 Deletions / Removals

We can delete documents from the collection using a command of the following form.

```
db.<collection name>.remove(<expression>)
```

For example we can remove our last entry with the following command.

```
> db.fall2014.remove({"_id": 6})
WriteResult({ "nRemoved" : 1 })
> db.fall2014.count()
6
>
```