**Óbuda University**

John von Neumann Faculty of Informatics

Institute of Biomatics

**MongoDB project documentation**

on “Database- and Big Data technologies” course

Course unit code: NIXAB1EMNE

Students:

Elizaveta Polushina

NEPTUN code: TFN92J

Hakan Kurtulus

NEPTUN code: HXMKGP

Academic advisors:

Dr. Rita Fleiner,

Péter Piros

Budapest, 2017

Contents

Introduction 3

Theoretical background 3

Scaling 3

1. Simple structure without sharding 3

2. Sharding structure 3

3. Sharding structure with replica sets 5

Practical part 6

Cluster compilation 6

Data handling 15

Problems faced during the project 20

Conclusion 21

# Introduction

**The main task within Database- and Big Data technologies** is obtaining the fundamentals of advanced database management concepts and procedure.

**The homework problem formulation**: to get acquainted with MongoDB database system. **The main task of homework** is to demonstrate the ability to use the acquired knowledge in practice: create a cluster storing the data multiple items in a distributed way.

# Theoretical background

## Scaling

In dynamically growing systems, based on the database, data volumes tend to increase rapidly and it is likely to face the following problem: current resources of the machine would not be enough for system operation.

Scaling is used to solve such problems. There are 2 types of scaling: horizontal and vertical. The concept of vertical scaling is in increasing the machine capacity (adding CPU, RAM, HDD). The horizontal scaling is adding of new machines to the existing ones and distributing the data between them. The first method seems to be easier in terms of implementing, as it does not require additional application settings and any additional configuration of the database itself. The disadvantage is that the power of one machine can theoretically fall into a deadlock. The second method is more complex within configuration but at the same time has a number of advantages:

1. Theoretically infinite scaling (as there can be supplied as many machines as it is necessary)
2. Better data security (only within replication) as machines can be located in different data centers (if one of the machines falls there are other ones remain)

Let’s describe some mongoDB deployment structures.

## 1. Simple structure without sharding

There is an application which communicates with mongod through the driver. **Mongod** is the primary mongoDB process with the following tasks: receiving and processing requests, changes implementation. The data is stored in mongod's chunks. Each chunk has its chunksize, which is 64 MB by default. Chunks are physically stored in the file dbName.n, where n is the sequence number starting with 0. After reaching the size of 64 MB (or other chunksize) chunk is divided in half, it turns 2 chunks with 32 MB chunksize. This separation occurs again after each chunk is filled. Thereby, the size of the file dbName.0 equals to 64 MB, dbName.1 – 128MB, dbName.2 – 256 MB and so on up to 2 GB. After the creation of the dbName.5 file (with the size of 2GB) the size growth stops and mongoDB simply creates files of the same size. MongoDB creates these files in advance, so there is no waste of time on creating a file when the data is needed to be written. Therefore, with a relatively small amount of real data it can be found that the hard disk space is occupied decently.

## 2. Sharding structure

The main difference of this method from the previous one is that the data is written into the chunks according to a certain range of a given field – **Shard key**. It determines the distribution of the collection’s data among the cluster’s chunks. The shard key is an indexed field or an indexed compound field that exists in every record in the data collection. Firstly, when there is an only chunk the range of the shard key value is ().

When the size of the chunk reaches its chunksize mangos evaluates the value of all shard keys within the chunk and divides the chunk so that the data is roughly equally divided.

Let’s assume that we have the following documents with the fields: name, age, id (shard key):

{“name”: “Max”, “age”: 23, “id”: 23}

{“name”: “John”, “age”: 28, “id”: 15}

{“name”: “Nick”, “age”: 19, “id”:56}

{“name”: “Carl”, “age”: 19, “id”: 78}

If the chunksize has already been achieved (let’s assume). The Mongos will split the range according to the Shard key (id) like the following: (-, 45]; (45, +). We got the result with the following operation:

That is the way we would get two chunks.

As the new data appears it will be written to the chunk that corresponds to the shard key range. Every time the chunk is divided the range of the chard key becomes narrower. All the chunks are stored on the **shards** (or servers). Each shard is an independent database (a single instance or replica set), and collectively, the shards make up a single logical database.

Let’s consider the **Sharding in MongoDB** scheme:

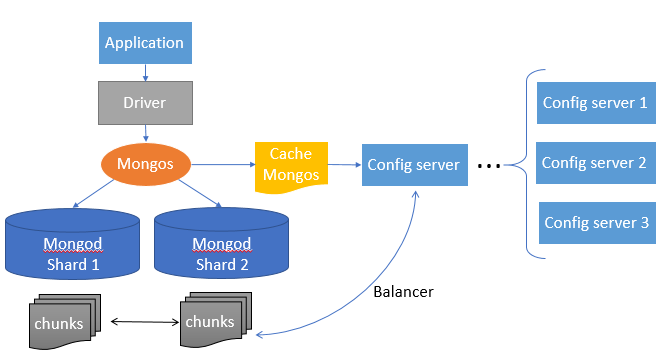


Figure 1 "Sharding structure in MongoDB"

MongoDB supports sharding through the configuration of sharded clusters. A **sharded cluster** is a set of nodes comprising a sharded MongoDB deployment. It consists of config servers, shards (were described above), mongos routing processes.

**Config server** has the main difference from mongod process: it does not process client requests as it is a repository of metadata. It knows the physical addresses of every chunk, what shard is needed to look for, what is the range of a particular shard. All the data is stored in a special place – **config database**.

There is one more component in the scheme – **mongos**, which is a router of the queries. It has the following responsibilities:

1. Caching of the data stored on the config server
2. Read and write requests routing: routing of the requests from applications to the necessary shards. Mongos knows where is the location of this or that chunk
3. Balancer background process running

**Balancer** is a background process (thread) that manages chunk migration. Its main function is to migrate chunks from one shard to another. The process proceeds as follows: Balancer sends the moveChunk command to the shard, from which the chunk will be migrated to another one. Shard receives the command and starts the process of copying the chunk to another shard. After the data have been copied, it is synchronized between these 2 chunks (because while the migration took place, new data could be added to the original chunk). After synchronization the shard, which adopted a new chunk, sends its address to the config server. The last in turn updates the address in the mangos cache. At the end of the process the original chunk is deleted.

Thus, the method can be suitable for a test environment of large applications or for small application production (with minimum 3 config servers’ usage). Three config servers ensure data redundancy and if one of them falls mongos will still receive up-to-date chunk addresses from other config servers.

## 3. Sharding structure with replica sets

Apart from sharding, the mentioned scheme contains replication of shards. MongoDB applies write operations on the primary and then records them to the oplog collection as logs, from which these logs are written to replica set 1 and replica set 2 (secondaries) asynchronously (there may be more than two secondaries). Replica sets apply the written operations to their data sets thereafter.

**Replica set** is a group of mongod instances that maintain the same data set. Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes. Every replica set (server) has priority characteristic: priority 0 or priority 1. A priority 0 component cannot be or become a primary. A priority 0 member might be a **secondary, arbiter** or **hidden secondary.** A priority 1 component can become **primary**.

**Primary** node is the only member in the replica set that receives write operations. A replica set can have only one primary capable of confirming write operations. **Secondaries** replicate the primary’s oplog and apply the operations to their data sets such that the secondaries’ data set reflects the primary’s data set (priority 1).

Every replica set could also have one more component – **Arbiter**, which does not have a copy of data and cannot become a primary (as it has priority 0). At the same time, it comes into a play when the primary becomes unavailable and election is held to pick a new primary from those servers still available. The key point of the election process is that a majority is required to elect a new primary (not just a majority from available servers, but a majority from all of the servers in the set). Thus, arbiter is needed when there is an even number of votes (i.e. servers). A 3-server replica set can tolerate a single failure. A 5-server replica set (which will be developed during the project) can tolerate 2 failed servers.

It is also very helpful to implement a **hidden secondary** into the replica set structure (priority 0). It maintains a copy of the primary’s data set (backup) but is invisible to client applications. At the same time hidden secondary has the right to vote in a new primary election. This component might be very useful to restore the data if a user manipulated with data incorrectly and want to restore it.

As it is seen from the schema, there is a duplication of the data. There are several reasons for the method implementation:

1. Redundancy ensures data security: when the primary crashes there is a vote occurring between replica sets and one of them becomes the primary at the end
2. The primary and secondaries can be located in different data centers. That feature can be useful if the server is physically damaged
3. Replica sets can be used to read data more efficiently. That is extremely useful when the database stores information about clients in different regions. (as it is possible to place replica sets in different countries and configure them in the following way: customers from one country subtract the data from the replica set placed in their country)

Described method is most commonly used in serious production applications where the data preservation is very important or there is a large number of data readings (application logic allows reading from different replica sets in this way).

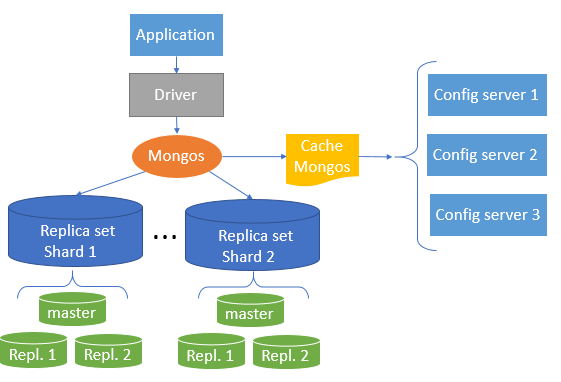


Figure 2 "Sharding structure with replication in MongoDB"

# Practical part

## Cluster compilation

First of all, let’s describe the cluster which will be implemented during the course of the project work. The hostname for all components is localhost.

|  |  |  |  |
| --- | --- | --- | --- |
| Machine Type | Components Installed | Description | Port |
| Mongos Router | Application | This server will server | 27131 |
| Mongo Config 1 (primary) | Mongo Config Server | Used as mongodb config server | 27121 |
| Mongo Config 2 (secondary) | Mongo Config Server | Used as mongodb config server | 27122 |
| Mongo Config 3 (secondary) | Mongo Config Server | Used as mongodb config server | 27123 |
| Shard 1 Primary | Mongo DB | Used as primary DB server in shard 1 | 27102 |
| Shard 1 Secondary | Mongo DB | Used as secondary DB server in shard 1 | 27101 |
| Shard 1 Secondary | Mongo DB | Used as secondary DB server in shard 1 | 27103 |
| Shard 1 Arbiter | Mongo DB | Used for election process in shard 1 | 27105 |
| Shard 1 Hidden Secondary | Mongo DB | Used as DB backup in shard 1 | 27104 |
| Shard 2 Primary | Mongo DB | Used as primary DB server in shard 2 | 27106 |
| Shard 2 Secondary | Mongo DB | Used as secondary DB server in shard 2 | 27107 |
| Shard 2 Secondary | Mongo DB | Used as secondary DB server in shard 2 | 27108 |
| Shard 2 Arbiter | Mongo DB | Used for election process in shard 2 | 27110 |
| Shard 2 Hidden Secondary | Mongo DB | Used as DB backup in shard 2 | 27109 |
| Shard 3 Primary | Mongo DB | Used as primary DB server in shard 3 | 27111 |
| Shard 3 Secondary | Mongo DB | Used as secondary DB server in shard 3 | 27112 |
| Shard 3 Secondary | Mongo DB | Used as secondary DB server in shard 3 | 27113 |
| Shard 3 Arbiter | Mongo DB | Used for election process in shard 3 | 27114 |
| Shard 3 Hidden Secondary | Mongo DB | Used as DB backup in shard 3 | 27115 |

Deployment will take place on the local Windows operating system. For cluster implementation we need installed mongoDB, version 3.2.

First, let’s create out first instances for the first Replica set, so called “Shard 1”, using the following command:

C:\mongo\server\3.2\bin\mongod.exe

--dbpath C:\mongo\data\shard1\primary --port 27101 --replSet "rs0" –shardsvr

(example above is for primary server of Shard 1)

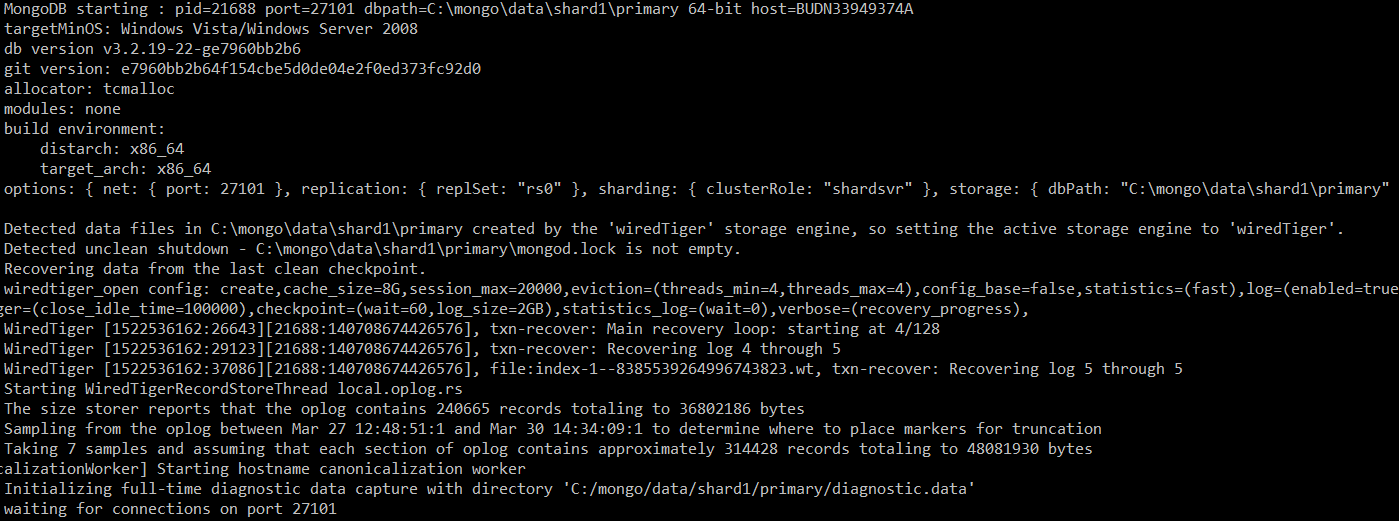


Figure 3 “Instance creation (Shard 1, primary)”

While creating new instance we should change the number of port. Thus, we have 3 servers in config structure, 5 servers in every shard (we have 3 shards overall), 1 router.

**--dbpath** parameter indicates the path, where the files will be stored. Those files store the data in binary format about the instance itself. File of **.ns** format is a namespace, needed for Database navigation.

**--port** is a port on which the Database object will be available.

**--shardsvr** parameter indicates that the instance is going to be a shard server.

**--replSet** parameter indicates that the instance is going to be a Replica set.

The same command is applicable for creating other Replica set members, the only difference is the port number and “dbpath”.

After each Replica set component initialization it is crucial to engage them.

With the help of the following command we will initiate each component of a Replica set (given example shows the creation of Shard 1, command is run from the server of Replica set itself):

rs.initiate( {

\_id : "rs0",

members: [

{ \_id: 0, host: "localhost:27101" },

{ \_id: 1, host: "localhost:27102" },

{ \_id: 2, host: "localhost:27103" },

{ \_id: 3, host: "localhost:27104" }

]

})

Let’s add an arbiter to the Shard 1 using the following command:

rs.addArb("localhost:27105")

The following code represents the algorithm: showing configuration of the Replica set, define the priority = 0 of the Hidden secondary and Arbiter, saving updated Replica set configuration.

use config

cfg = rs.conf()

cfg.members[3].priority = 0

cfg.members[4].priority = 0

rs.reconfig(cfg)

We can check the status of the created Replica set using the method:

rs.status()

As a result, we will get the following:

{

"set" : "rs0",

"date" : ISODate("2018-03-30T12:34:32.642Z"),

"myState" : 1,

"term" : NumberLong(3),

"heartbeatIntervalMillis" : NumberLong(2000),

"members" : [

{

"\_id" : 0,

"name" : "localhost:27101",

"health" : 1,

"state" : 2,

"stateStr" : "SECONDARY",

"uptime" : 278,

"optime" : {

"ts" : Timestamp(1522413249, 1),

"t" : NumberLong(3)

},

"optimeDate" : ISODate("2018-03-30T12:34:09Z"),

"lastHeartbeat" : ISODate("2018-03-30T12:34:31.842Z"),

"lastHeartbeatRecv" : ISODate("2018-03-30T12:34:31.080Z"),

"pingMs" : NumberLong(0),

"syncingTo" : "localhost:27102",

"configVersion" : 3

},

{

"\_id" : 1,

"name" : "localhost:27102",

"health" : 1,

"state" : 1,

"stateStr" : "PRIMARY",

"uptime" : 283,

"optime" : {

"ts" : Timestamp(1522413249, 1),

"t" : NumberLong(3)

},

"optimeDate" : ISODate("2018-03-30T12:34:09Z"),

"electionTime" : Timestamp(1522413003, 1),

"electionDate" : ISODate("2018-03-30T12:30:03Z"),

"configVersion" : 3,

"self" : true

},

{

"\_id" : 2,

"name" : "localhost:27103",

"health" : 1,

"state" : 2,

"stateStr" : "SECONDARY",

"uptime" : 278,

"optime" : {

"ts" : Timestamp(1522413249, 1),

"t" : NumberLong(3)

},

"optimeDate" : ISODate("2018-03-30T12:34:09Z"),

"lastHeartbeat" : ISODate("2018-03-30T12:34:30.869Z"),

"lastHeartbeatRecv" : ISODate("2018-03-30T12:34:31.131Z"),

"pingMs" : NumberLong(0),

"syncingTo" : "localhost:27102",

"configVersion" : 3

},

{

"\_id" : 3,

"name" : "localhost:27104",

"health" : 1,

"state" : 2,

"stateStr" : "SECONDARY",

"uptime" : 278,

"optime" : {

"ts" : Timestamp(1522413249, 1),

"t" : NumberLong(3)

},

"optimeDate" : ISODate("2018-03-30T12:34:09Z"),

"lastHeartbeat" : ISODate("2018-03-30T12:34:31.842Z"),

"lastHeartbeatRecv" : ISODate("2018-03-30T12:34:30.643Z"),

"pingMs" : NumberLong(0),

"syncingTo" : "localhost:27102",

"configVersion" : 3

},

{

"\_id" : 4,

"name" : "localhost:27105",

"health" : 1,

"state" : 7,

"stateStr" : "ARBITER",

"uptime" : 278,

"lastHeartbeat" : ISODate("2018-03-30T12:34:30.869Z"),

"lastHeartbeatRecv" : ISODate("2018-03-30T12:34:29.686Z"),

"pingMs" : NumberLong(0),

"configVersion" : 3

}

],

"ok" : 1

}

We can also see the current configuration of each Replica set, using:

rs.config()

The result is the following:

{

"\_id" : "rs0",

"version" : 3,

"protocolVersion" : NumberLong(1),

"members" : [

{

"\_id" : 0,

"host" : "localhost:27101",

"arbiterOnly" : false,

"buildIndexes" : true,

"hidden" : false,

"priority" : 1,

"tags" : {

},

"slaveDelay" : NumberLong(0),

"votes" : 1

},

{

"\_id" : 1,

"host" : "localhost:27102",

"arbiterOnly" : false,

"buildIndexes" : true,

"hidden" : false,

"priority" : 1,

"tags" : {

},

"slaveDelay" : NumberLong(0),

"votes" : 1

},

{

"\_id" : 2,

"host" : "localhost:27103",

"arbiterOnly" : false,

"buildIndexes" : true,

"hidden" : false,

"priority" : 1,

"tags" : {

},

"slaveDelay" : NumberLong(0),

"votes" : 1

},

{

"\_id" : 3,

"host" : "localhost:27104",

"arbiterOnly" : false,

"buildIndexes" : true,

"hidden" : true,

"priority" : 0,

"tags" : {

},

"slaveDelay" : NumberLong(0),

"votes" : 1

},

{

"\_id" : 4,

"host" : "localhost:27105",

"arbiterOnly" : true,

"buildIndexes" : true,

"hidden" : false,

"priority" : 0,

"tags" : {

},

"slaveDelay" : NumberLong(0),

"votes" : 1

}

],

"settings" : {

"chainingAllowed" : true,

"heartbeatIntervalMillis" : 2000,

"heartbeatTimeoutSecs" : 10,

"electionTimeoutMillis" : 10000,

"getLastErrorModes" : {

},

"getLastErrorDefaults" : {

"w" : 1,

"wtimeout" : 0

},

"replicaSetId" : ObjectId("5aba21925dd7601035438de5")

}

}

The same commands are applied for Shard 2 and Shard 3, the only difference is in localhost port.

The procedure of creating Config replica set is the same as for Shards (the difference is in port and folder):

rs.initiate( {

\_id: "configReplSet",

configsvr: true,

members: [

{ \_id: 0, host: "localhost:27121" },

{ \_id: 1, host: "localhost:27122" },

{ \_id: 2, host: "localhost:27123" }

]

} )

The last step is the router server creation:

C:\mongo\server\3.2\bin\mongod.exe

--congigdb configReplSet/localhost:27121, localhost:27122, localhost: 27123

--port 27131 –pidfilepath C:\mongo\data\router\mongos.pid

We defined config Replica set for router and “pidfilepath” to write configuration files inside.

We don’t need to define the priority of each server as it equals 1 by default.

For mongos connection we will use the following command (from the config window) for router connection:

C:\mongo\server\3.2\bin>mongo --host localhost:27131

Let’s add shards to the cluster using the following command (from the config window):

sh.addShard( "rs0/localhost:27101" )

sh.addShard( "rs1/localhost:27106" )

sh.addShard( "rs2/localhost:27111" )

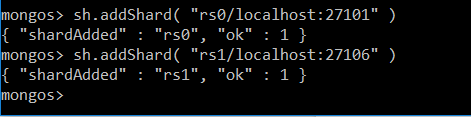


Figure 4 “Adding shard”

After running all the described commands, we obtain the following cluster:



Figure 5 "Project cluster to be set up"

Overall data structure looks like the following:

+---configs

| +---config

| | primary.bat

| | secondary1.bat

| | secondary2.bat

| |

| +---router

| | primary.bat

| |

| +---shard1

| | arbiter.bat

| | hidden\_secondary.bat

| | primary.bat

| | secondary1.bat

| | secondary2.bat

| |

| +---shard2

| | arbiter.bat

| | hidden\_secondary.bat

| | primary.bat

| | secondary1.bat

| | secondary2.bat

| |

| \---shard3

| arbiter.bat

| hidden\_secondary.bat

| primary.bat

| secondary1.bat

| secondary2.bat

|

+---data

| +---config

| | +---primary

| | +---secondary1

| | \---secondary2

| |

| +---router

| | mongos.pid

| |

| +---shard1

| | +---arbiter

| | +---hidden\_secondary

| | +---primary

| | +---secondary1

| | \---secondary2

| |

| +---shard2

| | +---arbiter

| | +---hidden\_secondary

| | +---primary

| | +---secondary1

| | \---secondary2

| |

| \---shard3

| +---arbiter

| +---hidden\_secondary

| +---primary

| +---secondary1

| \---secondary2

|

\---server

\---3.2

| GNU-AGPL-3.0

| MPL-2

| README

| THIRD-PARTY-NOTICES

|

\---bin

bsondump.exe

mongo.exe

mongod.exe

mongod.pdb

mongodump.exe

mongoexport.exe

mongofiles.exe

mongoimport.exe

mongooplog.exe

mongoperf.exe

mongorestore.exe

mongos.exe

mongos.pdb

mongostat.exe

mongotop.exe

## Data handling

Let’s connect once again to the Router server of the system to data creation. We will use the following command:

C:\mongo\server\3.2\bin\> mongo –host localhost: 27131

First of all, let’s build data collection, which will be named as “Users”. From the Mongos window we will create “cluster” database. The data will be generated with the help of db.collection.initializeUnorderBulkOp() method, which initializes and returns a new Bulk() operations builder for a collection. The builder constructs an unordered list of write operations that MongoDB executes in bulk.

use cluster

var bulk = db.users.initializeUnorderedBulkOp();

people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina", "Katie", "Jeff", "Balint", "Hakan", "Lisa"];

sex\_list = ["Male", "Female", "Other"];

for(var i=0; i<5000; i++){

user\_id = i;

name = people[Math.floor(Math.random()\*people.length)];

sex = sex\_list[Math.floor(Math.random()\*sex\_list.length)];

age = Math.floor(Math.random() \* (100 - 1 + 1)) + 1;

bulk.insert( { "user\_id":user\_id, "name":name, "sex":sex, "age":age });

}

bulk.execute();

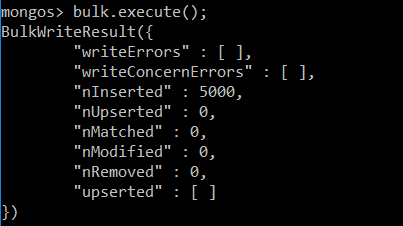


Figure 6 “Data generation”

Our future data collection will consist from the ‘user’ documents. Users have the following attributes: name, age and gender. Field values will be generated randomly by choosing from the defined lists: people, sex\_list. Age value will be generated from the following range: [1; 100].

Let’s enable sharding of the created database from the mongos window:

sh.enableSharding( "cluster" )

After running the command above, it is possible to begin sharding collection process. Firstly, we will try to run sharding process with disabled Balancer, moving chunks manually. Let’s define age field as a shard key. We can also define the size of chunk or it will change its size by default.

db.settings.save( { \_id:"chunksize", value: 5 } )

After running the command above, we specified the chunk size manually (5MB). This command is not necessary since we will chunk the data manually. Firstly, we should disable balancer for manual chunking. The following command disables Balancer:

use config

sh.setBalancerState( false )

The following code represents the algorithm: creating index for ‘age’ field, adding “users” collection to the shards, showing the current status of sharding process.

db.users.createIndex( { age : 1 } )

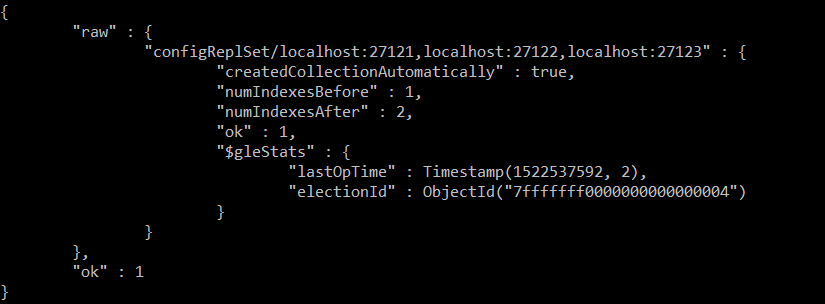


Figure 7 “Index creation”

As we will use sharing features of MongoDB we must define a shard key for the collection. The shard key determines the distribution of the collection’s documents among the cluster’s shards. The shard key is either an indexed field or indexed compound fields that exists in every document in the collection in this case the shard key is an indexed field.

In order to define a shard key before that we shall have an index for desired field, in this case we have created an index for “age” field and we will use this field as a shard key. Created index data and shard key information stored in config server.

Let’s define the index for the “users” collection as shard key:

sh.shardCollection( "cluster.users", { "age" : 1 } )

Let’s check the status of the system and see inserted data. As a default, shard mongoDB chooses “rs1” and put the inserted data into it:

db.printShardingStatus()

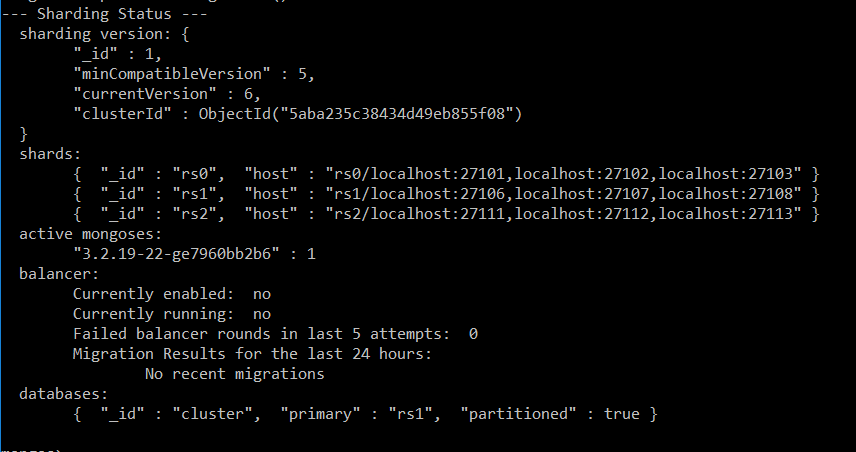
****

Figure 8 “Sharding status”

Let’s split the data into three chunks from the mongos window. The split command identifies the chunk in the ”users” collection of the cluster database. The first chunk will contain all of the documents with ‘age’ field value: 0≤25, the second one – 25<‘age’≤50, the third one – 50< ‘age’ ≤100 (as the maximum value of ‘age’ field is 100). Let’s run the command (against the admin database):

use admin

db.runCommand({split: "cluster.users", middle: {age: 25}})

db.runCommand({split: "cluster.users", middle: {age: 50}})

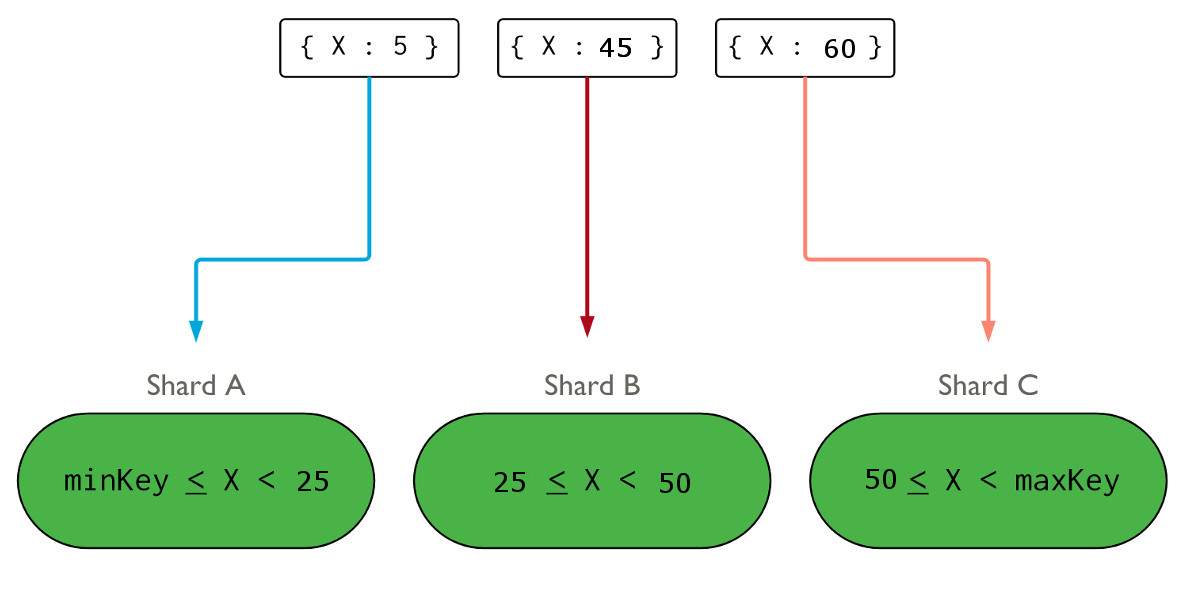


Figure 9 “Data split schema”

The following command will move chunks between the Shards (first chunk to the Shard 1 etc.):

db.runCommand({movechunk: "cluster.users", find: {age: 1}, to: "rs0"})

db.runCommand({movechunk: "cluster.users", find: {age: 25}, to: "rs1"})

db.runCommand({movechunk: "cluster.users", find: {age: 50}, to: "rs2"})

To ensure that the data collections was distributed properly we will use the following command:

db.printShardingStatus()

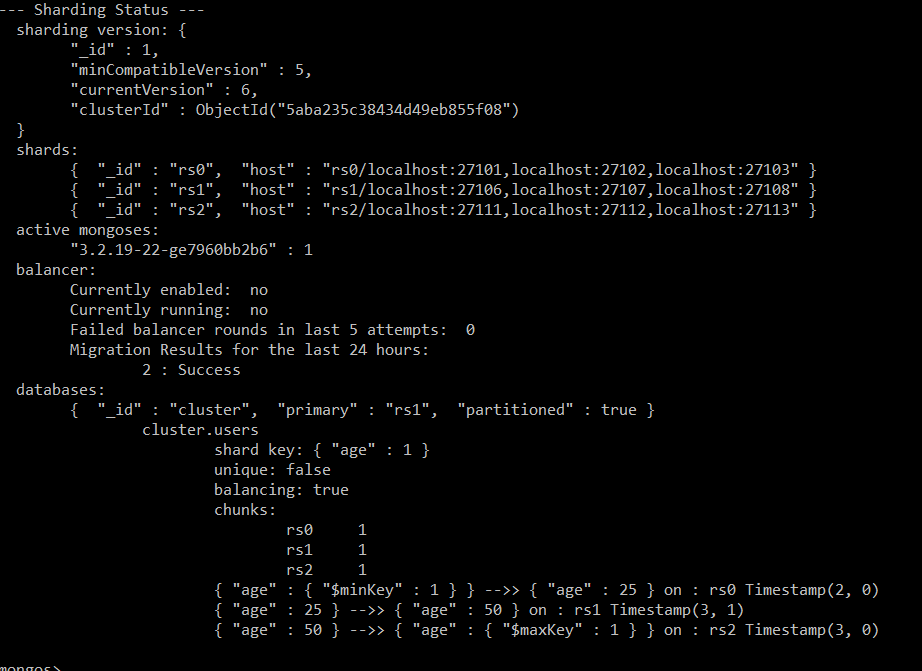


Figure 10 “Sharding status”

Let’s try to enable Balancer thread and see that the collection will be split up into 3 chunks. From the mongos window we will use command:

sh.enableBalancing("cluster.users")

We successfully distributed the dataset over three shards. We had three separated chunks and each of them is located in different Replica set.

Let’s test the dataset: is it distributed over shards or not? To check the dataset in Replica set 1 we need primary server connection:

C:\mongo\server\3.2\bin\> mongo –host localhost: 27101

With the following command we searched the documents have the value of ‘age’ field = 10:

use cluster

db.users.find({age:10})

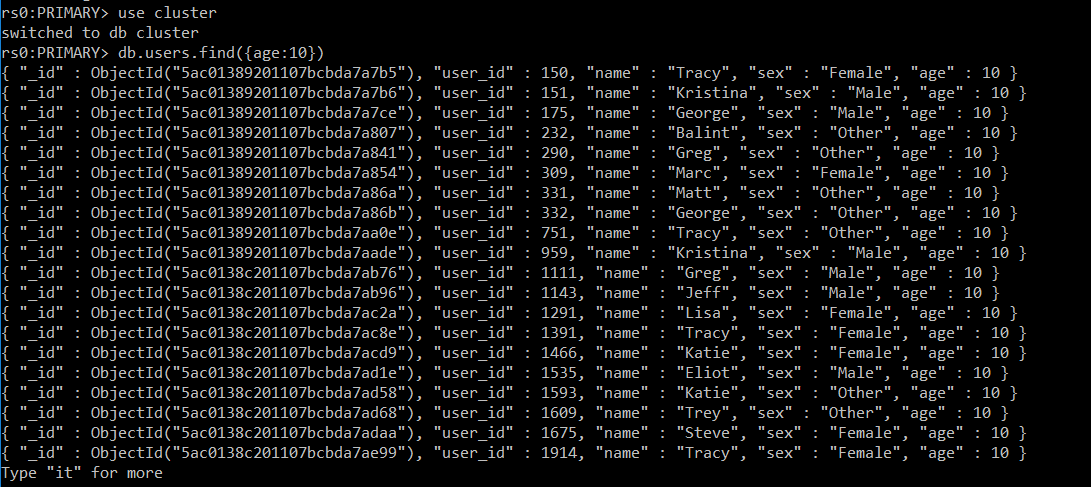


Figure 11 “Data search”

We ensured that we have first chunk on this Replica set. Let’s try to reach other chunks:

use cluster

db.users.find({age:38})

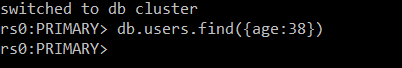


Figure 12 “Data search”

We could not get any result. Thus, this Replica set does not have other chunks except the first one. The first chunk stores only documents with ‘age’ field value: 1 < value < 25.

We can check Balancer status, running the following command (against the admin database):

use admin

db.adminCommand( { balancerStatus: 1 } )

The mode indicates whether the Balancer is running or stopped (‘full’ – running, ‘off’ – stopped). inBalancerRound specifies if the Balancer is currently distributing chunks. numBalancerRounds shows the number of rounds that have occurred since the config servers were started.

# Problems faced during the project

During the project work we faced the only problem concerning Balancer handling. After several attempts to enable Balancer thread we realized that we tried to run it against the cluster database (where we store the data). We switched to admin database thereafter and Balancer was enabled and worked properly.

The main purpose of this admin database is to store system collections and user authentication and authorization data, which includes the administrator and user's usernames, passwords, and roles. Access is limited to only to administrators, who can create, update, and delete users and assign roles. In a development environment it is convenient not to worry about users and passwords, however, when users interact with the database remotely on an application, it is essential to activate user authentication. In our case we did not have authenticated user to modify some configurations, in order to override this access problem, we connected to “admin” database to configure our settings as “admin”.

We were interested in the result of the following scenario. Let’s imagine that the Primary goes down. As a result of the election process, one of the secondaries becomes a new Primary. After some time, a new Primary goes down as well and remaining Secondary becomes a new Primary. What will happen if the original Primary will be recovered? Will it become a Primary and the data from the current Primary will be transferred to the original one? Or the original Primary will become a new Secondary? That is a question for our further investigation of the topic as we did not find the answer.

# Conclusion

Using NoSQL technology like MongoDB involves some trade-offs and different project realization approaches versus RDBMS (Relational Database Management System). We don’t see this structure as a replacement of a relational database, but it is another tool that helps to achieve the goal of the production application development.

During the course of the project work we defined several advantages and disadvantages of MongoDB.

Disadvantages:

* Overall data storage capacity is much higher as there are components that store the same data
* Less flexibility with querying as there are no JOINs
* It could be difficult to process the whole data as some operations are supported at a single document/shard level

Advantages:

* Flexible schema, that can be improved without large labor costs
* Easy scale-out and auto balancing
* Cost as the structure is ideal for running on cheaper commodity kit comparing with RDBMS

In conclusion, it is crucial to mention that NoSQL technologies like MongoDB provide an alternative, that complements RDBMS technologies. Today’s fast-paced world forces us to be able to offer as many project realization options as possible. This ability allows to be competitive on the market and acquire the customer.