José Mariano Portilla Landa 20/02/2022

Data Sharding Using MongoDB

Introduction

Sharding is a database partitioning technique to be able to store large data collections across multiple database servers (Espinosa, 2017/2021).

MongoDB is a document oriented database management system, it uses JSON to represent its documents and they are organized in collections, collections are organized in databases.

Sharded clusters are the way that MongoDB supports sharding, it is composed by the following components:

- Shards: They store data.
- Query Routers: Is in charge of redirecting queries to the appropriate shard.
- Config Servers: Stores metadata of clusters that allows query routers to select the appropriate shard.

In MongoDB sharding is enabled per collection, and it uses the attribute called shard key to partition and distribute the collection documents across the cluster (Espinosa, 2017/2021). Furthermore, it uses two kinds of partitioning:

- Ranged Based: data is partitioned using intervals called chunks.
- Hash Based: a hash function is used to partition data.

When a shard server has too many chunks, MongoDB automatically redistributes the chunks across shards, which is known as cluster balancing (Espinosa, 2017/2021).

We can also tag a range of shard key values to guide partitioning, which provides us with isolation of data and the ability to place shards in related geographical regions.

Methodology and Results

To do this practice we used a project called “play with docker” available on web as well as a github repository containing instructions developed by Javier Espinoza, the links are the following:

- https://labs.play-with-docker.com/
- https://github.com/javieraespinosa/dxlab-sharding#inserting-and-querying-data

To install and configure the environment we created an instance in play with docker and enter the following commands.

```
[node1] (local) root@192.168.0.28 ~
$ git clone https://github.com/javieraespinosa/dxlab-sharding.git
Cloning into 'dxlab-sharding'...
remote: Enumerating objects: 77, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 77 (delta 10), reused 3 (delta 0), pack-reused 55
Receiving objects: 100% (77/77), 747.58 KiB | 17.38 MiB/s, done.
Resolving deltas: 100% (41/41), done.
[node1] (local) root@192.168.0.28 ~
$ cd dxlab-sharding
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker-compose pull
Pulling cli               ... done
Pulling configserver      ... done
Pulling queryrouter.docker ... done
Pulling shard1.docker     ... done
Pulling shard2.docker     ... done
Pulling shard3.docker     ... done
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker-compose pull
Pulling cli               ... done
Pulling configserver      ... done
Pulling queryrouter.docker ... done
Pulling shard1.docker     ... done
Pulling shard2.docker     ... done
Pulling shard3.docker     ... done
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
mongo        3.0       fdab8031e252   3 years ago   232MB
```

Figure 1: Cloning the github repository, downloading images specified in docker-compose.yml and verifying the existence of docker images.

To prepare the sharded clusters we enter the following commands.

```
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker-compose up -d
Creating network "dxlab-sharding_default" with the default driver
Creating shard3.docker        ... done
Creating configserver         ... done
Creating shard1.docker        ... done
Creating shard2.docker        ... done
Creating dxlab-sharding_cli_1 ... done
Creating queryrouter.docker   ... done
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS                  PORTS                     NAMES
41d701e3a2e2   mongo:3.0   "docker-entrypoint.s…"   2 seconds ago   Up Less than a second   0.0.0.0:27017->27017/tcp   queryrouter.docker
edf21b009ab2   mongo:3.0   "docker-entrypoint.s…"   5 seconds ago   Up 2 seconds            0.0.0.0:27118->27017/tcp   shard2.docker
c1019f68ab4a   mongo:3.0   "docker-entrypoint.s…"   5 seconds ago   Up 1 second             0.0.0.0:27117->27017/tcp   shard1.docker
3b27d0a6dc72   mongo:3.0   "docker-entrypoint.s…"   5 seconds ago   Up 1 second             27017/tcp                  configserver
ff68eade7176   mongo:3.0   "docker-entrypoint.s…"   5 seconds ago   Up 2 seconds            0.0.0.0:27119->27017/tcp   shard3.docker
[node1] (local) root@192.168.0.28 ~/dxlab-sharding
$ docker network inspect dxlab-sharding_default
[
    {
        "Name": "dxlab-sharding_default",
        "Id": "97b33de8a99f76405da67e3997cf4f50bbbc7d0c47d447c174401849fd515526",
        "Created": "2022-02-21T02:12:38.889313546Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
```

Figure 2: Starting the cluster, listing containers, and showing IP's (part 1).

```
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "3b27d0a6dc7214121fb8db9e8c47b13528737ca8b7ef57c1b4d09620daf3970e": {
                "Name": "configserver",
                "EndpointID": "34c87d58500f574ad9e0c8325defd5b83e80fa7e1cd48ef046e003f6c33dc1ef",
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "41d701e3a2e25058f04872e1dcaa9819ba767ec1c674f24d613b8bb5c81e9ca6": {
                "Name": "queryrouter.docker",
                "EndpointID": "989f7b154fe2091eaa870d75dbae406f99754c56508f78cf23222485af2f1d96",
                "MacAddress": "02:42:ac:13:00:05",
                "IPv4Address": "172.19.0.5/16",
                "IPv6Address": ""
            },
            "c1019f68ab4a20cff61d43480446317cb69190f62d9fb5a6a71817f23c8448cf": {
                "Name": "shard1.docker",
                "EndpointID": "20c75fe5caeaac06bb32326fdbd9fd0a707e9d1cd8310e6ad9c37ca5b9ebab9c",
                "MacAddress": "02:42:ac:13:00:06",
                "IPv4Address": "172.19.0.6/16",
                "IPv6Address": ""
            },
            "edf21b009ab2c2676972f94117c41749a465eec9be69fee201b6127edacf0f00": {
                "Name": "shard2.docker",
                "EndpointID": "17fbf63d90659ce82ac6b015843676061d14926168fc6d4d562d465c17cc7574",
                "MacAddress": "02:42:ac:13:00:04",
                "IPv4Address": "172.19.0.4/16",
                "IPv6Address": ""
```


Figure 3: Listing containers IP's (part 2).


```
            },
            "ff68eade71763b58d4d7e21ee4c42698e781fb95a558e1cd53b8e3afdde487d0": {
                "Name": "shard3.docker",
                "EndpointID": "51ec3ad82123a25e58dbf784136848861dc45c0867d93a1034a7f23dcb8ead77",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "default",
            "com.docker.compose.project": "dxlab-sharding",
            "com.docker.compose.version": "1.26.0"
        }
    }
]
```


Figure 4: Listing containers IP's (part 3).

To add a shard server we need to perform the following commands.

```
$ docker-compose run --rm cli
root@d2734f7317fa:~# mongo --host queryrouter.docker
MongoDB shell version: 3.0.15
connecting to: queryrouter.docker:27017/test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
mongos> use admin
switched to db admin
mongos> db.runCommand({
...   addShard: "shard1.docker",
...   name: "shard1"
... })
{ "shardAdded" : "shard1", "ok" : 1 }
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f51bfb8033f2a46712a5")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
```

Figure 5: Entering the cluster environment, connecting to the query router and adding a shard to the cluster.

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
```

Figure 6: Verifying the status of the cluster.

Now we need to populate the database.

```
root@bd9cb128a63a:~# mongoimport \
>   --host queryrouter.docker \
>   --db mydb \
>   --collection cities \
>   --file ./cities.txt
2022-02-21T02:20:59.323+0000    connected to: queryrouter.docker
2022-02-21T02:20:59.933+0000    imported 29353 documents
root@bd9cb128a63a:~# mongo --host queryrouter.docker
MongoDB shell version: 3.0.15
connecting to: queryrouter.docker:27017/test
mongos> show dbs
admin   (empty)
config  0.016GB
mydb    0.078GB
mongos> use mydb
switched to db mydb
mongos> show collections
cities
system.indexes
mongos>
```

Figure 7: Importing the contents of cities.txt into the database.

```
mongos> db.cities.find().pretty()
{
        "_id" : "01001",
        "city" : "AGAWAM",
        "loc" : [
                -72.622739,
                42.070206
        ],
        "pop" : 15338,
        "state" : "MA"
}
{
        "_id" : "01002",
        "city" : "CUSHMAN",
        "loc" : [
                -72.51565,
                42.377017
        ],
        "pop" : 36963,
        "state" : "MA"
}
```

Figure 8: Running a query on the new imported data.

Figure 9: Counting the documents of the collection cities.

To create a ranged based partitioning the following commands need to be executed. We can see in the sharding status (in Fig. 10) that it displays the sharding version, the shards, the balancer, and the databases, and as expected, we get the shard key (state) with a minKey and a maxKey as well as the chunks. In Fig. 11 we can see that we get three chunks that depend on the shard key.

```
mongos> db.createCollection("cities1")
{ "ok" : 1 }
mongos> show collections
cities
cities1
system.indexes
mongos> sh.enableSharding("mydb")
{ "ok" : 1 }
mongos> sh.shardCollection("mydb.cities1", { "state": 1} )
{ "collectionsharded" : "mydb.cities1", "ok" : 1 }
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "mydb",  "partitioned" : true,  "primary" : "shard1" }
                mydb.cities1
                        shard key: { "state" : 1 }
                        chunks:
                                shard1  1
                        { "state" : { "$minKey" : 1 } } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 0)
```

Figure 10: Creating a new collection and enabling sharding using the attribute called state as shard key.

```
mongos> db.cities.find().forEach(
...    function(doc) {
...       db.cities1.insert(doc);
...    }
... )
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "mydb",  "partitioned" : true,  "primary" : "shard1" }
                mydb.cities1
                        shard key: { "state" : 1 }
                        chunks:
                                shard1  3
                        { "state" : { "$minKey" : 1 } } -->> { "state" : "MA" } on : shard1 Timestamp(1, 1)
                        { "state" : "MA" } -->> { "state" : "RI" } on : shard1 Timestamp(1, 2)
                        { "state" : "RI" } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 3)
```

Figure 11: Populating the new collection and verifying the state.

To define a hash-based partitioning we follow a similar approach. We create a new collection and enable sharding with the same attribute called state, however the difference is that now it has a value of "hashed", we get two shards (Fig. 12). After populating the collection, we can see in Fig. 13 that now we have 4 chunks that depend on the shard key.

```
mongos> sh.shardCollection(
...    "mydb.cities2", { "state": "hashed" }
... )
{ "collectionsharded" : "mydb.cities2", "ok" : 1 }
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "mydb",  "partitioned" : true,  "primary" : "shard1" }
                mydb.cities1
                        shard key: { "state" : 1 }
                        chunks:
                                shard1  3
                        { "state" : { "$minKey" : 1 } } -->> { "state" : "MA" } on : shard1 Timestamp(1, 1)
                        { "state" : "MA" } -->> { "state" : "RI" } on : shard1 Timestamp(1, 2)
                        { "state" : "RI" } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 3)
                mydb.cities2
                        shard key: { "state" : "hashed" }
                        chunks:
                                shard1  2
                        { "state" : { "$minKey" : 1 } } -->> { "state" : NumberLong(0) } on : shard1 Timestamp(1, 1)
                        { "state" : NumberLong(0) } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 2)
```

Figure 12: Enabling sharding and verifying status.

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "mydb",  "partitioned" : true,  "primary" : "shard1" }
                mydb.cities1
                        shard key: { "state" : 1 }
                        chunks:
                                shard1  3
                        { "state" : { "$minKey" : 1 } } -->> { "state" : "MA" } on : shard1 Timestamp(1, 1)
                        { "state" : "MA" } -->> { "state" : "RI" } on : shard1 Timestamp(1, 2)
                        { "state" : "RI" } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 3)
                mydb.cities2
                        shard key: { "state" : "hashed" }
                        chunks:
                                shard1  4
                        { "state" : { "$minKey" : 1 } } -->> { "state" : NumberLong(0) } on : shard1 Timestamp(1, 1)
                        { "state" : NumberLong(0) } -->> { "state" : NumberLong("3630192931154748514") } on : shard1 Timestamp(1, 3)
                        { "state" : NumberLong("3630192931154748514") } -->> { "state" : NumberLong("8134625179139298768") } on : shard1 Timestamp(1, 4)
                        { "state" : NumberLong("8134625179139298768") } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 5)
```

Figure 13: Populating collection and verifying status.

To activate the cluster balancing first we add more shards in Fig. 14, then in Fig. 15 we can see that shard1 gets two chunks, shard2 gets 1, and shard3 also gets 1.

```
mongos> use admin
switched to db admin
mongos> db.runCommand( { addShard: "shard2.docker", name: "shard2" } )
{ "shardAdded" : "shard2", "ok" : 1 }
mongos> db.runCommand( { addShard: "shard3.docker", name: "shard3" } )
{ "shardAdded" : "shard3", "ok" : 1 }
```

Figure 14: Adding more shards to the cluster.

```
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("6212f6a722b8cafd8f9988ce")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1.docker:27017" }
        {  "_id" : "shard2",  "host" : "shard2.docker:27017" }
        {  "_id" : "shard3",  "host" : "shard3.docker:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                4 : Success
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "mydb",  "partitioned" : true,  "primary" : "shard1" }
                mydb.cities1
                        shard key: { "state" : 1 }
                        chunks:
                                shard1  1
                                shard2  1
                                shard3  1
                        { "state" : { "$minKey" : 1 } } -->> { "state" : "MA" } on : shard2 Timestamp(2, 0)
                        { "state" : "MA" } -->> { "state" : "RI" } on : shard3 Timestamp(3, 0)
                        { "state" : "RI" } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(3, 1)
                mydb.cities2
                        shard key: { "state" : "hashed" }
                        chunks:
                                shard1  2
                                shard2  1
                                shard3  1
                        { "state" : { "$minKey" : 1 } } -->> { "state" : NumberLong(0) } on : shard2 Timestamp(2, 0)
                        { "state" : NumberLong(0) } -->> { "state" : NumberLong("3630192931154748514") } on : shard3 Timestamp(3, 0)
                        { "state" : NumberLong("3630192931154748514") } -->> { "state" : NumberLong("8134625179139298768") } on : shard1 Timestamp(3, 1)
                        { "state" : NumberLong("8134625179139298768") } -->> { "state" : { "$maxKey" : 1 } } on : shard1 Timestamp(1, 5)
```

Figure 15: Verifying status.

In order to use tags as a guide to partitioning, first we need to associate tags to shards in Fig. 16. Then we create, populate and enable sharding in Fig. 17. In Fig. 18 we define and associate key ranges to shards. Finally we check the status in Fig. 19, we can see that the tagging was successful because shard3 gets 5 chunks while shard1 and shard2 only get 1.

```
mongos> sh.addShardTag("shard1", "CA")
mongos> sh.addShardTag("shard2", "NY")
mongos> sh.addShardTag("shard3", "Others")
```

Figure 16: Associating tags to shards.

```
mongos> use mydb
switched to db mydb
mongos> db.createCollection("cities3")
{ "ok" : 1 }
mongos> sh.shardCollection("mydb.cities3", { "state": 1} )
{ "collectionsharded" : "mydb.cities3", "ok" : 1 }
mongos> db.cities.find().forEach(
...     function(doc) {
...        db.cities3.insert(doc);
...     }
... )
```

Figure 17: Create, populate and enable sharding.

```
mongos> sh.addTagRange("mydb.cities3", { state: MinKey }, { state: "CA" }, "Others")
mongos> sh.addTagRange("mydb.cities3", { state: "CA" }, { state: "CA_" }, "CA")
mongos> sh.addTagRange("mydb.cities3", { state: "CA_" }, { state: "NY" }, "Others")
mongos> sh.addTagRange("mydb.cities3", { state: "NY" }, { state: "NY_" }, "NY")
mongos> sh.addTagRange("mydb.cities3", { state: "NY_" }, { state: MaxKey }, "Others")
```

Figure 18: Defining and associating key ranges to shards.

```
mydb.cities3
        shard key: { "state" : 1 }
        chunks:
                shard1  1
                shard2  1
                shard3  5
        { "state" : { "$minKey" : 1 } } -->> { "state" : "CA" } on : shard3 Timestamp(7, 1)
        { "state" : "CA" } -->> { "state" : "CA_" } on : shard1 Timestamp(6, 0)
        { "state" : "CA_" } -->> { "state" : "MA" } on : shard3 Timestamp(4, 4)
        { "state" : "MA" } -->> { "state" : "NY" } on : shard3 Timestamp(4, 5)
        { "state" : "NY" } -->> { "state" : "NY_" } on : shard2 Timestamp(7, 0)
        { "state" : "NY_" } -->> { "state" : "RI" } on : shard3 Timestamp(4, 8)
        { "state" : "RI" } -->> { "state" : { "$maxKey" : 1 } } on : shard3 Timestamp(5, 0)
         tag: Others  { "state" : { "$minKey" : 1 } } -->> { "state" : "CA" }
         tag: CA  { "state" : "CA" } -->> { "state" : "CA_" }
         tag: Others  { "state" : "CA_" } -->> { "state" : "NY" }
         tag: NY  { "state" : "NY" } -->> { "state" : "NY_" }
         tag: Others  { "state" : "NY_" } -->> { "state" : { "$maxKey" : 1 } }
```

Figure 19: Verifying the status.

Discussion

A shard key is an attribute that is used to partition and distribute the collection documents across the cluster, the sharding key may impact the sharding strategy but it is not directly dependent of it, we can have either strategy as we saw in our results. In the case of the interval-oriented strategy MongoDB limit chunks depending on the domain of the sharding key, this is well suited when the collection has documents in which the values of the shard key are well distributed among them. The hash-based strategy on the other hand, is well suited when the values of the sharding key is not well distributed among the collection. To have a more balanced distribution of data across shards, the strategy to be used depends on the data of the collection, however, a hash-based approach should be more balanced in general.

Tag based sharding is an interesting option when we need more control of where our shardings are going, one example would be when we need to divide our customers depending on their location, if we use tags, then we can make sure that the shards are positioned in geographical related regions. When a new shard is added to a cluster containing already other shards with data, cluster balancing takes place, the chunks are automatically distributed between the old and the new shards.

Finally, to test if a sharded collection was an interesting solution in comparison to a centralized one, I would measure the time that it takes to run queries on both cases while using a large enough database so that the return time of queries can be appreciated.

Conclusions

Sharding a database is necessary when data grows too large and the response time is not the same as it once was, it allows us to scale depending on our needs. To do this, MongoDB uses sharded clusters with different strategies that can be applied depending on the dataset itself and on what we believe that would be better. Sharding can be really advantageous when having a really big database, because it can greatly impact the efficiency of our system, however, when not having a large enough database we can say that the advantages are almost insignificant and that time should not be invested in developing this solution.

References

Espinosa, J. (2021). *Sharding Data Collections with MongoDB* [Jupyter Notebook].

https://github.com/javieraespinosa/dxlab-sharding (Original work published 2017)