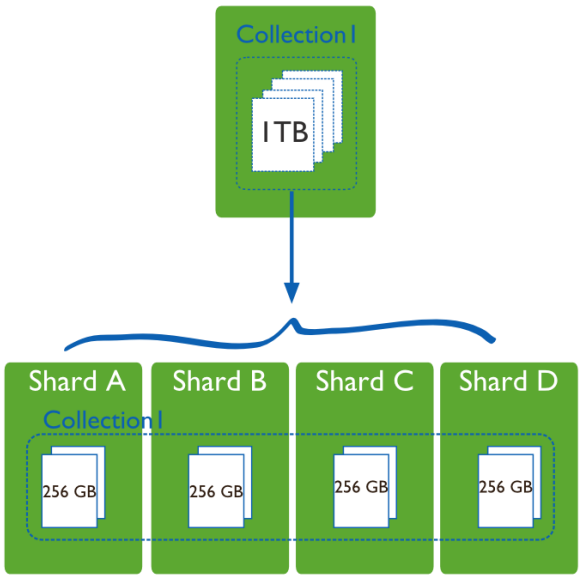
Studying **Data Sharding** using MongoDB

1 Context

NoSQL databases started gaining popularity in the 2000's when companies began investing and researching more into distributed databases. An important aspect of NoSQL databases is that they have no predefined schema. Records can have different fields as necessary. NoSQL databases, apart from using an Application Programming Interface(API) or query language to access and modify data, may also use the MapReduce method which is used for performing a specific function on an entire dataset and retrieving only the result.

Sharding is a *method* for storing a large collection of data across multiple servers called **shards** (cf. image below). This allows increased performance as each server handles different sets of data thus if a single database becomes too large its performance may diminish due to the increased time a query takes.



2 Objectives

- Discover and study the process of sharding a database using **MongoDB**, a *document oriented* NoSQL database management system.
- Shard a database using one of the proposed strategies by **MongoDB** and perform tests on the implemented strategy.

3 TO DO

Execute the following hands on tasks and answer the questions:

- Guided tour of the sharding strategies provided by MongoDB (section 4)
- Application of the studied strategies on a use case (section 5)

4 Guided tour on MongoDB sharding mechanisms

4.1 Preparing a database

MongoDB stores documents using its own binary format called BSON. This format is a binary version of the widely used JSON (JavaScript Object Notation) format and stands for Binary JSON. Although MongoDB uses BSON internally, the manipulation of documents in the MongoDB shell interface and client software is done using JSON due to its readability and open standard. In MongoDB databases are composed of *collections of documents*.

For this exercise, you have first to (i) create a *database* and then (ii) create and populate a database *collection*. Therefore we are going to use a data collection called **`cities`.**

4.1.1 Creating and populating a documents database

4.1.1.1 Starting a MongoDB Instance

- Start a MongoDB instance: [1]

```
mkdir -p ~/db/shard1                    # Folder containing the DB files
mongod --shardsvr --dbpath ~/db/shard1 --port 27021
```

4.1.1.2 Creating a database and database collection

- Using a **new shell**, connect to the MongoDB instance:

```
mongo --host localhost:27021
```

- Create the database **mydb** and the database collection **cities**:

```
use mydb                        # Create the DB if not exists
db.createCollection("cities")
```

- Verify the existence of the database (*mydb*) and the database collection (*cities*):

```
show dbs
show collections
```

[1] Note that the instance will be used later as a *shard server* (option --shardsvr).

4.1.1.3 Populating and querying a database

- Using a **new shell**, import the content of the file ***cities.txt*** into ***mydb.cities*** collection. After that close the shell:

```
mongoimport --host localhost:27021 --db mydb --collection cities --file ~/cities.txt
```

- Using the **shell connected to MongoDB**, verify the existence of data in the database by issuing some queries. After that close the shell:

```
db.cities.find()                    # Equivalent to SELECT * FROM mydb.cities
db.cities.count()
```

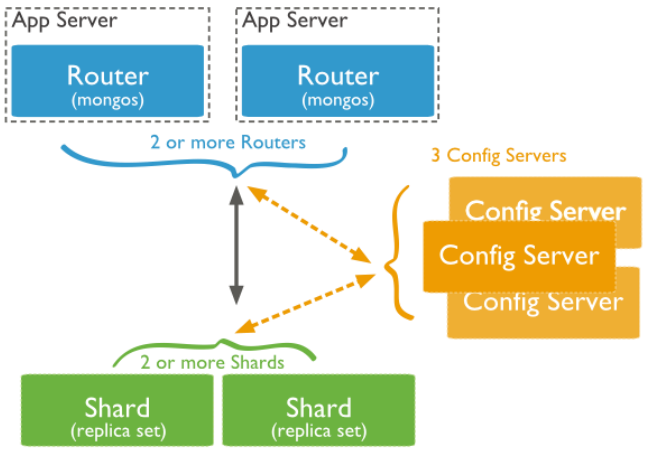
Q1. Describe in natural language the content of the database collection. Test the command `db.cities.find().pretty()` Include a sample of the result in your description.

How many cities are there in the database?

4.2 Configuring a sharded cluster

As discussed in the course, MongoDB supports sharding through a **sharded cluster**. A sharded cluster is composed of the following components:

- **Shards**: store the data.
- **Query routers**: direct operations from *clients* to the appropriate shard(s) and return results to clients.
- **Config servers:** store cluster's metadata. The query router uses this metadata to target operations to specific shards.



For the sake of simplicity you will configure with a *simple sharded cluster* (cf. image below) composed of:

- One config server.
- One query router (*mongos* instance).
- One shard (*mongod* instance).

4.2.1 Starting a config server instance

- Using a **new shell**, start a config server (*mongod instance*):

```
mkdir ~/db/configdb
mongod --configsvr --dbpath ~/db/configdb --port 27020
```

4.2.2 Starting a query router instance

- Using a **new shell**, start a query router (*mongos instance*) connected to the *config server* instance in port 27020:

```
mongos --configdb localhost:27020 --port 27019
```

4.2.3 Adding a shard instance to the cluster

- Using a **new shell**, connect to the query router (*mongos instance*):

```
mongo  --host localhost:27019
```

- Add to the cluster the *mongo instance* containing the **mydb** database:

```
use admin
db.runCommand( { addShard: "localhost:27021", name: "shard1" } )
```

- Verify the state of the cluster:

```
sh.status()
```

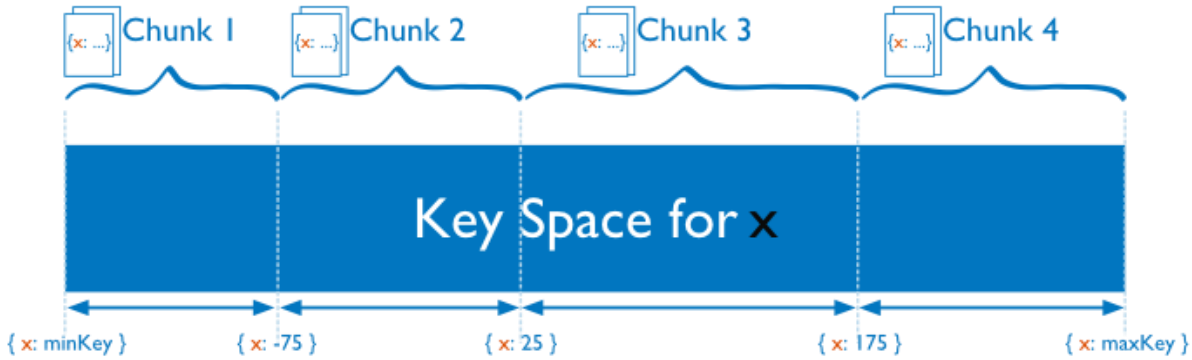
Q2 Which is the important information reported by this command? Refer to the explanation of the previous lecture.

4.3 Sharding a database collection

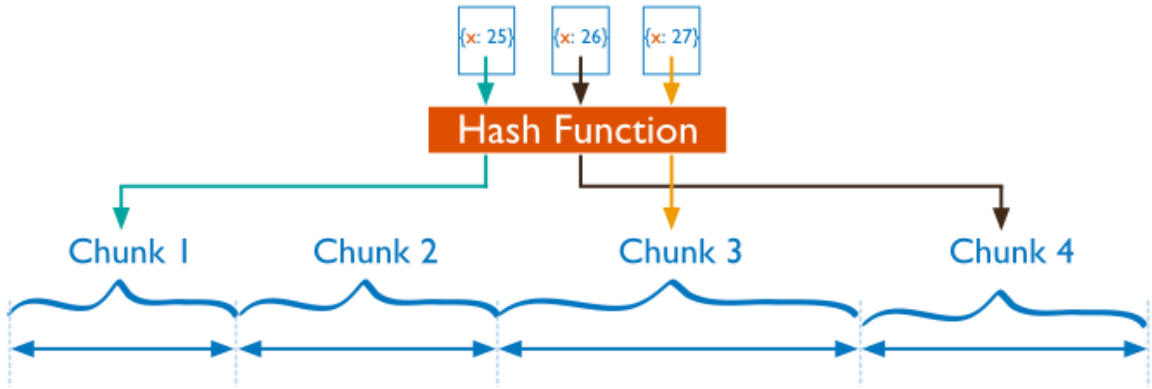
Recall that sharding is enabled in MongoDB on a *per-basis collection*. When sharding is *enabled* on a collection, MongoDB partitions the data into the shards of a cluster using a **shard key**, an *indexed field* that exists in every document stored in the collection.

MongoDB divides the *shard key values* into **chunks** (of documents) and distributes the chunks evenly across shards. To divide the shard key values into chunks, MongoDB uses two kinds of partitioning strategies:

- ***Range based partitioning***: data is partitioned into *ranges* [min, max] determined by the shard key. Each range represents a chunk.



- **Hash based partitioning**: data is partitioned into chunks using a hash function.



In what follows you will shard copies of the collection **mydb.cities** using *range based* and *hash based* partitioning.

4.3.1 Sharding a collection using ranges

- Using the **shell connected to the query router** (*mongos instance*), create the collection **cities1** in database **mydb**:

```
use mydb
db.createCollection("cities1")
show collections   # Verify collection existence
```

- Enable sharding on the collection **mydb.cities1** using as *shard key* the attribute **`state`**:

```
sh.enableSharding("mydb")
sh.shardCollection("mydb.cities1", { "state": 1} )
```

- Verify the **number of chunks**:

```
sh.status()
```

Q3 How many chunks did you create? Which are their associated ranges? Include a screen copy of the results of the command in your answer to support your answer.

- Populate collection **cities1** using the content of the collection **mydb.cities**:

```
db.cities.find().forEach(
  function(d) {
    db.cities1.insert(d);
  }
)
```

- Verify the **number of chunks** after population:

```
sh.status()
```

Q4 How many chunks are there now? Which are their associated ranges? Which changes can you identify in particular? Include a screen copy of the results of the command in your answer to support your answer.

4.3.2 Sharding a collection using hash-based partitioning

Now let us study the sharding strategy using a hash function.

- Using the **shell connected to the query router** (*mongos instance*), create the collection **cities2**:

```
db.createCollection("cities2")
show collections   # Verify collection existence
```

- Enable sharding on the collection **mydb.cities2**. The principle that we will adopt is to use the attribute **state** as *shard key*.

```
sh.shardCollection("mydb.cities2", { "state": "hashed" } )
```

- Verify the **number of chunks** before collection population:

```
sh.status()
```

Q5 How many chunks did you create? What differences do you see with respect to the same task in the range sharding strategy? Include a screen copy of the results of the command in your answer to support your answer.

- Populate collection **cities2**:

```
db.cities.find().forEach(
  function(d) { db.cities2.insert(d); }
)
```

- Verify the **number of chunks** after population:

```
sh.status()
```

Q6. How many chunks are there now? Include a screen copy of the results of the command in your answer to support your answer. Compare the result with respect to the range sharding. Include a screen copy of the results of the command in your answer to support your answer.

4.4 Balancing data across sharded cluster

Balancing is the process MongoDB uses to distribute data of a sharded collection evenly across a sharded cluster. When a shard has too many of a sharded collection's chunks

compared to other shards, MongoDB *automatically balances* the chunks across the shards.

MongoDB balancer supports **tagging** a range of shard key values. Using *tags* you can:

- Isolate specific subset of data on a specific set of shards.
- Ensure that relevant data reside on shards that are geographically close to the user.

For the final part of this exercise you will analyze the behavior of the *MongoDB balancing process* by adding *tagged shards* to your cluster.

4.4.1 Adding shards to a cluster

- Using a **new shell**, start another MongoDB instances:

```
mkdir -p ~/db/shard2                    # Folders containing DB files
mongod --shardsvr --dbpath ~/db/shard2 --port 27022
```

- Using the **shell connected to the query router** (*mongos instance*), add the new *mongo instance* to the cluster:

```
use admin
db.runCommand( { addShard: "localhost:27022", name: "shard2" } )
```

- Wait a few seconds and check the status of the cluster:

```
sh.status()
```

Q6. Draw the new configuration of the cluster and label each element (`router`, `config server` and `shards`) with its corresponding port as you defined in the previous tasks.

4.4.2 Sharding using tagged shards

- Using a **new shell**, start another MongoDB instance:

```
mkdir -p ~/db/shard3                    # Folders containing DB files
mongod --shardsvr --dbpath ~/db/shard3 --port 27023
```

- Using the **shell connected to the query router** (*mongos instance*), add the new *mongo instance* to the cluster:

```
use admin
db.runCommand( { addShard: "localhost:27023", name: "shard3" } )
sh.status()
```

- Associate tags to *shard instances*:

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

- Create, shard and populate a new collection named **cities3**:

```
db.createCollection("cities3")

sh.shardCollection("mydb.cities3", { "state": 1} )

use mydb
db.cities.find().forEach(
   function(d) {  db.cities3.insert(d);  }
)
```

- Associate **shard key ranges** to *tagged shards*:

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

- Display *cluster* information:

```
sh.status()
```

Q**7**. Analyze the results and explain the logic behind this tagging strategy. Connect to the shard that contains the data about California, and count the documents. Do the same operation with the other shards. Is the sharded data collection complete with respect to initial one?

5 Designing and testing a sharded databases

One of the challenges of designing a sharded collection is to identify the attribute that will be used as *sharding key*, independently of the strategy that will be used. In the previous task we chose the attribute **state** as *sharding key*.

Next we ask you to analyze again the collection **cities** and choose another attribute that seems appropriate for scaling queries.

Q8

1. Explain your reasons for choosing your *sharding key.* Which are possible implications when querying data and when adding/deleting data?
2. Implement a cluster with 2/3 shards and distribute the data across them according to your chosen key using **both** the *range* and *hash based* strategies.
3. Populate your sharded database using the data collection **cities.txt**.
4. Compare the behavior of your sharding solution with the range and hash based strategies.
    a. Once you populate the sharded database, is the result balanced?
    b. Give an example of query or a manipulation operation that can potentially benefit from your sharding strategy. **Test your hypothesis and report the operations and the results of this operation by copying them from the screen**.
    c. In which cases is your sharding useless for scaling the management of the data collection? Give examples to support your arguments.
    d. Define a criterion for defining critical documents and use the tagging strategy for isolating these data. Show evidence of the operation and results.