# TP sur MongoDB

Inspired from labs by P.Rigaux et al.

Useful documents:

- docker tutorial: https://docs.docker.com/get-started/
- docker instructions: https://docs.docker.com/engine/reference/commandline/cli/
- Everything you need to know about MongoDB is in the doc, and even often in the starting guide, but I provide you with indications to save time.

Color code:
`docker help` : instruction for the shell
`help` : instruction for the mongo shell.

## Docker help

`docker pull <image>` download a docker image (but docker run will automatically download if needed).

`docker images` list the images that have been dowloaded.

`docker ps -a` list containers, whether active or not.

`docker rm -f <conteneur>` removes a container(`-f` (force) allows to remove even active containers).

`docker run <image> [programme sur l'image]` to launch a container. Options for docker run:

- `--help`
- `-d` launches container in detached mode (i.e., background).
- `--name <conteneur>` allows to name a container
- `--net host` opens the port on host (puts container in host network)

Exemples:

```
docker run -d --name shell-mongo-tp1 mongo mongo
docker run --name <conteneur> --net host mongo mongod --replSet <nomset> --port <numport>
```

`docker exec -it <conteneur> <application>` lance l'application dans un conteneur en cours d'exécution.

`docker cp <fichier> <conteneur>:<fichier>` copie le fichier dans le conteneur, depuis l'hôte.

## Parameters for mongo servers (mongod/mongos)

`mongo` the mongo shell, which can be executed on servers `mongod` or `mongos`

`mongod --replSet <nomset> --port <numport>` launches server (daemon) `mongod`:

- `--replSet` allows to connect containers into the replica set.
- `--port` specifies on which port the server is listening

In case of sharding:

- `--configserv` for a configuration server
- `--shardsvr` for a server storing data

`mongos --port 27017 --configdb <nomsetcfg>/<hote>:<numportcfg>` launches a router

## Instructions for the mongo shell

```
// Initialize a replica set: from one member, add the other servers
rs.initiate()
rs.add ("<hostname1>:<port1>")
rs.add ("<hostname2>:<port2>")

// to enable reads on a slave server (must be typed on the slave):
rs.slaveOk()

// to add chunks (must be typed on router)):
sh.addShard("<nomset>/<hostname>:<port>")
```

# 1 Preparing the database

1. Check that docker is correctly installed by launching the `hello-world` image (no need to name the container).
2. Pull the mongodb image : `mongo:latest`

   If you want to know more, you may check `https://hub.docker.com/_/mongo/`, but you don't have to.
3. Launch the server `mongod` in some container that you will name `mongoserv`.
4. Launch the Mongo shell (🇬🇧 shell) `mongo`, in the very same docker container.

   > The mongo shell is a javascript interpreter. So we can send both javascript and MongoDB instructions. It admits many Unix shortcuts (Tab, Up, Ctrl+a, Ctrl+r, Ctrl+k, Ctrl+c...).
5. Ask for database `bdtest`. What happens? is the database created?
6. Insert some arbitrary doc in the `movies` collection. What happens?
7. Inserting a lot of data with individual insertion instructions is not an option. So we will use the dedicated program `mongoimport` which can import json and csv files.

   Download the file with enron mails, and copy that file into the `mongoserv` container. Then, load the file into the `emails` collection of the `bdenron` databases.

   > `mongoimport` is one of the programs available on the `mongo` docker imag. It assumes that the input file contains one JSON document per line, without any comma at the end of a line. Our document satisfies that syntax. We can therefore use the syntax:
   >
   > `mongoimport --db <base> --collection <collection> --file <fichier.json>`
   >
   > Had the file been a JSON array, we would have used option `--jsonArray`

# 2 MongoDB Queries

1. Connect to the database. Cout the number of documents in `emails` collection.
2. What does `db.emails.find().pretty()` return? Perform a few cursor iterations.
3. List mails sent by `no.address@enron.com`.
4. Display the $10^{\text{th}}$ mail, by alphabetical order on sender, with the following 2 approaches:
   (a) store the cursor returned by `find()` in a javascript variable, then iterate over this cursor (or transform the cursor into an array)
   (b) using the `skip()` and `limit()` instructions from MongoDB.
5. List `folder` from which mails have been extracted. Check the type of the result..
6. For each mail, display its author and recipient, sorting the result by author.
7. Display messages received in 2000 from April onwards.

   *Hint: date is stored as a string.*
8. Display messages for which "replyto" is not null.

   *Hint: we may use query operators* `$not` *and* `$type`*, or directly compare to* `null`
9. Number of mails sent by each operator in the above time range. Sort the result by decreasing number of mails.

   *Hint: sorting will be one step in the aggregation pipeline. One can indeed not perform a* `sort()` *on the somewhat specific cursor returned by* `aggregate()`*. If you want to know more, check* the doc about aggregate.
10. Recompute the above query, using mapReduce (without sort).

    *Hint: first compute query without period condition, then filter using query* query check doc on mapReduce
11. Using the doc, and through experimentation, check what the following query returns:

```
db.emails.find({   text: {$regex: '^(?si).*P(?-si)resident Bush.*$'}   })
```

# 3 MongoDB on multiple nodes: replication

1. Launch 3 `mongod` servers listening on 27018, 27019, 27020 respectively. Each in its own container, of course. Do not detach the servers. If you did, launch `docker attach <containername>` to observe in the terminal what happens..

2. Launch the `mongo` interpreter in containers, specifying the corresponding port. `docker exec -it mongoserv1 mongo -port 27018`

3. Initialize the replica set by adding two new servers. The ip address is the same for all 3, and you can, for instance, see the address in the result of `rs.initiate()`.

4. Check the nodes taking part in the replica set with `rs.conf()`. For more information about synchronization (check the primary node and secondary nodes), use `rs.status()` or equivalently `db.adminCommand( replSetGetStatus : 1 )`.

   - Observe the interval between 2 heartbeats.
   - (skip that question if you do not know about javascript)
     display only information about the members, from the status, and using some javascript notation (dot or bracket) to access the property of a javascript object.
     The display only for each member: number status and name in the following format:
     `"0: PRIMARY - a-140395.ups.u-psud.fr:27018"`

5. Launch a mongo interpreter on each slave node. Insert enron data. Check that data are propagated from the primary server. (try to read directly on other servers, then try again after `rs.slaveOk()`). Check what happens if you try to write on a slave.

6. Stop the primary server with `db.shutdownServer()`) , then observe the messages in the shell to understand what happens to the replica set.

# 4 MongoDB on multiple nodes: sharding

1. Launch a configuration server (`mongod --configserv ...`). Your server will conssist in a single replica set having a single primary node[1]. Then initialize the replica set.

   I suggest using the following values for parameters[2]: listen on port 27018, name of replica set: serveurconfig, server name: mconfig.

2. Launch a router (`mongos -configdb ...`).

   Suggested parameters: listen on port 27017, router name: mrouteur.

3. Launch 2 servers to contain partitioned data (`mongod --shardsvr`). Each data server will be a replica set made of a single node. Initialize the replica sets.

   Suggested parameters: listen on port 27030 (respectively 27031), replica set names: rset1, rset2, server names: mshard1, mshard2.

4. Add two chunks to the cluster managed by the router, with:

   `sh.addShard(<nomset> --port <numport>)`

   If you followed the suggestions above, apart from the machine name which may not be a-140395:
   `docker exec -it mrouteur mongo -port 27017`
   then
   `sh.addShard("rset1/a-140395.ups.u-psud.fr:27030")`
   `sh.addShard("rset2/a-140395.ups.u-psud.fr:27031")`

5. Check the status of your database with

   `sh.status()`

   `db.printShardingStatus()`

   `db.stats()`

[1] it is not advisable to have a single node in the replica set, but we will not bother about fault tolerance here to keep thing simpler.
[2] thisis not mandatory; you may alter the values. But those are the ones I will use in my solution

6. create a database `bdpart`, with a collection `dblp`. Authorize partitioning of collections in database `bdpart`:

   `db.adminCommand( { enableSharding:  "<database name>" } )`

   Partition through hashing the collection `dblp` on the field "ident"

   `sh.shardCollection("<base>.<collection>", {"clé":  "hashed"})`

7. Retrieve and unzip the data:

   `wget http://b3d.bdpedia.fr/files/dblp.json.zip; unzip dblp.json.zip`

   You will see that the file already contains a field `_id` whose values are unfit for a mongodb identifier. Consequently, you should rename this field into `ident`, for instance using `sed`.

8. Copy the file in the router container, then load the data with `mongoimport`: you need to use `--jsonArray` given the format of that document. Observer.

9. Write a query that returns in alphabetical order (on authors) all documents in collection `dblp` for which `"type" = "Article"`. No need to run the query, but analyze its execution plan with `explain()`:

   `<your query>.explain()`

## Before you leave, you must run the following script:

```
#!/bin/bash

# remove  containers:
docker ps -aq | xargs -r docker rm -f

# remove unused images (all, here, since all containers were killed):
docker images --no-trunc | awk '{ print $3 }' | xargs -r docker rmi

# remove unused volumes:
docker volume ls -qf dangling=true | xargs -r docker volume rm
```

And delete the JSON files you have downloaded. (Otherwise you are likely to face quota troubles).