Apache Cassandra – CQL Practical Assignment 3

Starting Cassandra

After getting the latest Cassandra image, you can run it by calling

docker run --name mycassandra -p 3000:9042 -d cassandra:latest

You can now define databases (called in Cassandra keyspaces), explore their schema and then query your databases with Cassandra's own CQL language.

For that, you will need a client connecting to the Cassandra server. The simplest one to use is the command line CQL shell (cqlsh), which can be launched in a terminal window by

docker exec -it mycassandra bash

then in this bash command terminal run the following command (launching CQL):

cqlsh --encoding=utf8

```
root@9e43fd73a188:/# cqlsh --encoding=utf8
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.7 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Your answers (CQL queries and explanations in some cases) are expected from Section 2 onwards.

Section 1: Metro lines (for testing and warmup purposes only)

We will first model a Cassandra database for storing information on the Parisian metro. You are provided two JSON files containing the raw data, called metropolitan-lines.json and metropolitan-stops.json. The former file details all the metro lines in Paris. The latter file details all the metro stops in Paris. They are only provided as an indication for the kind of data we want to model in this database.

Devise a Cassandra schema corresponding to the information in the two json files and to the queries you anticipate on this dataset. Create the necessary table(s), try to insert some data (only a few manual insertions, one at a time, as in the example below; attention with copy-paste: make sure straight quotes are obtained in the process).

Correction

```
CREATE KEYSPACE IF NOT EXISTS Metros
 WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor': 3 };

USE Metros;
create table lines(color text, name text, number text,
                   route_name text, primary key(color));

create type line(line text, position int);

create table stops(
         description text,
         latitude float,
         lines set<frozen<line>>,
         longitude float,
         name text,
```

```
        primary key(description)
    );
insert into lines JSON '
      {
    "color": "#F58F53",
    "name": "Tramway 3A",
    "number": "3A",
    "route_name": "PONT GARIGLIANO - HOP G.POMPIDOU <-> PORTE DE VINCENNES"
      }';

  insert into stops json '
    {
    "description": "Bd Resistence - 93380",
    "latitude": 48.84218,
    "lines": [{"line": "line10", "position": 2}],
               "longitude": 2.238863,
                "name": "Boulogne-Resistence"
      }
';
```

Section 2. Restaurants and inspections

In this practical assignment, we will set up a database representing restaurants and inspections of these restaurants. A sample of data is provided. Before you start the modelling work, you are strongly advised to take the time to open the restaurants archive and inspect the various files (at least the headers / a few lines thereof, to get a first idea of the initial data layout, using commands such as *more* or *tail*).

Section 2.1 Relational approach

Let us start by creating the database (called keyspace), then we will consider how it can be queried. In this section, we will create the database as if it were relational, and we will formulate some basic queries. Once the limitations of this approach are well understood, we will use the NoSQL features of Cassandra to go further.

Creating the database

For starters, we can use the following command:

```
CREATE KEYSPACE IF NOT EXISTS restaurantsNY WITH REPLICATION = {'class':
'SimpleStrategy', 'replication_factor': 1};
```

Note that the replication factor is set to 1, which is sufficient in a centralized setting.

We will now select for use this database in the cqlsh client.

```
USE restaurantsNY;
```

Tables

We can now create the tables (*Column Family* in Cassandra) *Restaurants* et *Inspections,* with the following schemas:

```
CREATE TABLE Restaurants (
   id INT, Name VARCHAR, borough VARCHAR, BuildingNum VARCHAR, Street VARCHAR,
   ZipCode INT, Phone text, CuisineType VARCHAR, PRIMARY KEY (id));

CREATE INDEX ix_Restaurant_cuisine ON Restaurants (CuisineType);
```

```
CREATE TABLE Inspections (
   idRestaurant INT, InspectionDate date, ViolationCode VARCHAR,
   ViolationDescription VARCHAR, CriticalFlag VARCHAR, Score INT, GRADE
VARCHAR, PRIMARY KEY (idRestaurant, InspectionDate));

CREATE INDEX ix_Inspection_Restaurant ON Inspections (Grade);
```

Note that inspections are linked to restaurants.

We can verify that the two tables have been created in cqlsh:

```
DESC Restaurants;
DESC Inspections;
```

We can see the schema of the two tables but also information about the data layout and storage in the Cassandra database.

Importing the data

Now we can use the csv files to fill the Colum Families.
Note that you must first copy them in the Docker Cassandra container, by running copy commands like below (replace "path-to-file/" with the path where you saved the data). This is to be run in a "normal" terminal of the machine, not in the terminal of Cassandra docker container.

```
docker cp path-to-file/RestaurantsNY.csv    mycassandra:/
docker cp path-to-file/RestaurantsInspectionsNY.csv    mycassandra:/
```

Then you can import the two files **RestaurantsNY.csv** and **RestaurantsInspectionsNY.csv**

```
use restaurantsNY;
COPY Restaurants (id, name, borough, buildingnum, street,
                  zipcode, phone, cuisinetype)
FROM '/RestaurantsNY.csv' WITH DELIMITER=',';
COPY Inspections (idrestaurant, inspectiondate, violationcode,
                  violationdescription, criticalflag, score, grade)
FROM '/RestaurantsInspectionsNY.csv' WITH DELIMITER=',';
```

(Note: the number of imported records may be different then the number of records in the file, this may be due to overwriting records sharing the same primary key.)

To verify the newly imported data you can run the following queries:

```
SELECT count(*) FROM Restaurants; -- 25624 results
SELECT count(*) FROM Inspections; -- 149818 results
```

Querying

The queries will be formulated in CQL (for *Cassandra Query Language*), which is to a large extent inspired by SQL. You can find the detailed syntax description here: <https://cassandra.apache.org/doc/latest/cql/dml.html#select>).

Formulate in CQL the following queries (provide CQL expression and eventually screenshot of result; for count queries you must provide the exact result):

1. List all the restaurants / 10 restaurants.

   (Note :In `cqlsh`, Ctrl+C to exit the printing of result lines)

2. List all the restaurant names.
3. Name and quarter (*borough*) of restaurant number 41569764.
4. Number of restaurants offering French cuisine.
5. Number of inspections having grade C.
6. Number of inspections having grade above C.
7. Dates et grades of the inspections of restaurant number 41569764.
8. Names of the restaurants offering French cuisine.
9. Names of the restaurants from the Brooklyn quarter.
10. Grades et scores of inspections for restaurant number 41569764 with a score of at least 10.
11. Grades (non null) of inspections with a score greater than 30.
12. Number of results for the previous query.
13. Name of restaurants having a Cassandra identifier above 0.

    Hint: Use the *token()* function.

    Note: The Cassandra identifier is not the value of the primary key, but the hash (function token ( )) of the *partition key*.

    For our purposes, the partition key is either the primary key itself, if the primary key is a single attribute, or the first attribute of the primary key, if the primary key is composed of several attributes. Therefore, the order in which we list the primary key attributes matters.

    The token function is the hash function that Cassandra uses to take rows and to map them onto the token ring, i.e., decide in which partition to put them.

    The Cassandra identifier / hash of the partition-key is obtained by token(partition key) .

14. Number of restaurants having a Cassandra identifier

    (a) above 0
    (b) below 0

    What can you observe with these results ?

Additional queries

1. For the query below, take the necessary steps in order to be able to run it without *ALLOW FILTERING*.

```
SELECT Name FROM Restaurants WHERE borough='BROOKLYN' ;
SELECT count(*) FROM Restaurants WHERE borough='BROOKLYN' ;
```

2. Using two indexes on *Restaurants* (*borough* and *cuisineType*) find the names of all French cuisine restaurants in Brooklyn.
3. Use the command *TRACING ON* before executing the previous query, in order to see what index(es) has (have) been used.
4. We want the names of the restaurants having at least one A grade in some inspection. Is this possible in CQL?

Section 2.2 NoSQL approach

Joins are not supported in CQL, but this limitation is partly compensated by the possibly to nest data, in this way creating documents that can represent the pre-computed result of a join.

This of course depends on knowing beforehand the queries that may be formulated, since the nested data model cannot by symmetrical; this means that certain accesses may be supported while others may not be.
We have a few alternatives for nesting data, such as

- Nested types
- Maps

The following exercises start by indicating the access patterns (queries) we anticipate. You must propose a data model / schema accordingly, using these two alternatives, and then verify that they meet our needs in terms of querying.

First access pattern

We need to be able to select restaurants depending on their grades. For instance, we may want to answer the question left unanswered in the previous section (at least one A…).

Here are the suggested steps:

Define a schema associating restaurants and their inspections, using nested types.

1. Create the table InspectionsRestaurantsNY.
2. Insert a test document in this table.
3. Do the import using the import app for large JSON documents as follows:

Note

To import a large json file, you will rely on the Java app provided, as follows:

```
java -jar JSonFile2Cassandra [-host <host>] [-port <port>]
            [-keyspace <keyspace>] [-columnFamily <columnFamily>] [file]
```

Example :

```
java -jar JSonFile2Cassandra.jar -host localhost -port 3000 -keyspace
restaurantsNY       -columnFamily       InspectionsRestaurantsNY       -file
InspectionsRestaurantsNY.json
```

(This is to be run in a "normal" terminal of the machine, not in the terminal of the Cassandra docker container)

(Note: the number of imported records may be different then the number of records in the file, this may be due to overwriting records sharing the same primary key.)

(Note: ignore SLF4J warnings at execution of this java -jar command)

To verify the newly imported data you can run the following query :

```
SELECT count(*) FROM InspectionsRestaurantsNY; -- 150882 results
```

4. Create an index on *Grade* in the table InspectionsRestaurantsNY, then find the restaurants with at least one A grade.

5. List for each restaurant (described by idRestaurant and name) the number of inspections.

Second access pattern

Now we want to be able to search inspections by their quarter (*borough*).

1. Is this possible with the previous schema?
2. Suggest a different schema and create the table. Use this time the map alternative, with the inspection date as key.
3. Hint: Inspections map<text , frozen< Inspection>>

4. Perform only 1-2 manual insertions to test this schema (no large import from a JSON file required here).
   Hint (sample on the same idea, but on artists…):

```
CREATE TABLE ... (
...
Artists map<text, frozen<Artist>>
);

INSERT INTO ...  JSON '{...
"artists":{"Travolta":{"firstname":"John", "lastname":
"Travolta", "age":45, "country":"US", "grade":""}, ... }
}';
```

5. Find all the restaurants (along with their inspections) in the Bronx.

TURN PAGE.

Replication & consistency
(optional – not evaluated)

Our cluster

Let's create a Cassandra cluster, with 5 nodes. For that, we create a first node serving as the access point (seed in Cassandra terminology), by which we will add more nodes.

```
docker run -d -e "CASSANDRA_TOKEN=1"  --name mycassandra1 -p 3001:9042 cassandra:latest
```

Note that we explicitly indicate the placement of the server on the ring. In production, it is better to use virtual nodes, as previously explained. This requires a bit more configuration, and we are going to be satisfied with a simple exploration here.

We need the IP address of this first server. The following command retrieves the NetworkSettings information.
IPAddress of the JSON document returned by the inspect statement.

```
docker inspect -f '{{.NetworkSettings.IPAddress}}' mycassandra1
```

You will obtain an IP address. Let us assume it is 172.17.0.X (replace with yours!)
Let us now create the other servers, indicating the first one as the seed.

```
docker run -d -e "CASSANDRA_TOKEN=10" -e "CASSANDRA_SEEDS=172.17.0.X"  --name
mycassandra2 cassandra:latest
```

```
docker run -d -e "CASSANDRA_TOKEN=100" -e "CASSANDRA_SEEDS=172.17.0.X"  --name
mycassandra3 cassandra:latest
```

```
docker run -d -e "CASSANDRA_TOKEN=1000" -e "CASSANDRA_SEEDS=172.17.0.X" --name
mycassandra4 cassandra:latest
```

```
docker run -d -e "CASSANDRA_TOKEN=10000" -e "CASSANDRA_SEEDS=172.17.0.X" --name
mycassandra5 cassandra:latest
```

We have just created 5 Cassandra nodes, all running in the background, thanks to the Docker engine.

Run the same command for each of them …
```
docker inspect -f '{{.NetworkSettings.IPAddress}}' mycassandra2
```

Keyspace and data

Let's insert some data now. In practice, it may be faster to directly launch the command interpreter on one of the nodes with the following command:

```
docker exec -it mycassandra1 /bin/bash
```

```
[docker]$ cqlsh 172.17.0.X
```

Create a keyspace:

```
CREATE KEYSPACE ReplicaEx WITH REPLICATION = {'class':'SimpleStrategy',
'replication_factor':3};
```

```
USE ReplicaEx;
```

Let’s insert one document:

```
CREATE TABLE data (id int, value text, PRIMARY KEY (id));
INSERT INTO data (id, value) VALUES (10, 'My first document');
```

We have just created a keyspace, which will replicate the data on 3 nodes. The data table will use the primary key id and the hash function of the partitioner to store the document in one of the 5 nodes, then replicate in the 2 next nodes on the ring. It is possible to obtain with the function token () the hash value for the key of the documents.

```
select token(id), id from data;
```

Let's check with the *nodetool* utilitary that the cluster has indeed 5 nodes, and let's look at how each node has been distributed on the ring. Nodes are expected to be placed in ascending order of their identifier.

```
docker exec -it mycassandra1 /bin/bash
[docker]$ /usr/bin/nodetool ring
```

Let's test that the previously inserted document has been replicated on 2 nodes.

```
docker exec -it mycassandra1 /bin/bash
[docker]$ /usr/bin/nodetool cfstats replicaex
```

Look for each node at the value of Write Count. It should be 1 for 3 consecutive nodes on the ring, and 0 for the others. Let's check now that by connecting to a node that does not contain the document, we can still access it. For example, let's consider that the node mycassandra2 does not contain the document.

```
docker exec -it mycassandra2 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > USE ReplicaEx;
cqlsh:ReplicaEx > SELECT * FROM data;
```

Consistency of readings

To study the consistency of the data when reading, we will stop 2 of the 3 Cassandra nodes having the data. To do this, we will use Docker. Consider that the data is stored on the nodes mycassandra1, mycassandra2, and mycassandra3.

```
docker pause mycassandra2
docker pause mycassandra3
docker exec -it mycassandra1 /bin/bash
[docker]$ /usr/bin/nodetool ring
```

Check that the nodes are in a "down" state. We can now set the level of data consistency. Let's make a read request. The system is set to ensure the best consistency of the data. The request is expected to crash because in ALL mode, as Cassandra is waiting for the answer from all the nodes.

```
docker exec -it mycassandra1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use ReplicaEx;
cqlsh:ReplicaEx > consistency ALL;
# should display "Consistency level set to ALL."
cqlsh:ReplicaEx> select * from data;
# should display "Unable to complete request: one or more nodes were unavailable." Or "NoHostAvailable: … "
```

Now let's test the ONE mode, which should normally return the resource (the data) from the fastest node.

```
docker exec -it mycassandra1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use ReplicaEx;
cqlsh:ReplicaEx > consistency ONE; # should display "Consistency level set to ONE."
```

```
cqlsh:replicaEx > select * from data;
```

In this scheme, the system is very available, but does not check the consistency of the data. As proof, it returns the resource (the data) to the client even though the other nodes that contain the resource are unavailable (they might contain a more recent version!).

Finally, let's test the QUORUM strategy. With 2 nodes out of the 3 lost (so more than half), the query should normally return an error.

```
docker exec -it mycassandra1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use ReplicaEx;
cqlsh:ReplicaEx > consistency QUORUM;
# should display "Consistency level set to QUORUM"
cqlsh:ReplicaEx > select * from data;
# should display "Unable to complete request: one or more nodes were unavailable"
```

Let's now reactivate one of the nodes and test again:

```
docker unpause mycassandra2
docker exec -it mycassandra1 /bin/bash
[docker]$ nodetool ring
[docker]$ cqlsh 172.17.0.X
cqlsh > use ReplicaEx;
cqlsh:repli > consistency QUORUM;
# should display "Consistency level set to QUORUM"
cqlsh:ReplicaEx > select * from data;
```

When the node is reactivated (via Docker), it still takes a few tens of seconds before it is actually reintegrated to the cluster. The important thing is that the quorum rule is validated, with 2 nodes out of 3 available, and Cassandra agrees to return the resource to the client.

To go further with learning about consistency on Cassandra, you can have a look at one of Datastax tutorials: https://docs.datastax.com/en/cql/3.3/cql/cql_using/useTracing.html.

Cassandra and Big Data – Conclusions

Cassandra is considered today as one of the most powerful NoSQL databases in a Big Data environment. When a real-world project requires working on very large volumes of data, the challenge is to write the data quickly. And on this point, Cassandra has demonstrated its superiority. As we saw before, the scalability of Cassandra makes it particularly suitable for an environment where data are distributed on multiple servers. Thanks to Cassandra's architecture, distribution requires lightweight management, balanced over all nodes.

One would think that putting a Cassandra cluster in production is done with a few simple steps/clicks. In reality, this may be much more delicate. Indeed, Cassandra offers a very open data modeling, which gives access to a lot of possibilities, and allows above all to do anything. Unlike the relational data, with Cassandra, we cannot just store documents. It is indeed necessary to have a detailed knowledge of the data that will be stored, the way in which it will be queried, its distribution on the different nodes. The design of the Cassandra data model therefore requires special attention, as poorly performing modeling in production with PBs of data will give catastrophic results.

Cassandra also makes it possible not to constrain the number of key / value pairs in the documents. When a document has a lot of values, we are talking about a "wide row". Wide rows are an advantage in terms of modeling, but the more values a document has, the heavier it is. We must therefore consider this tradeoff. Remember also that in Cassandra, being a NoSQL database, the concept of joins does not exist.

The similarities with the relational model and particularly SQL make the initial learning curve easy, particularly to those who have a lot of experience with SQL. On the other hand, they can lead users to underestimate this extremely powerful database. Cassandra offers high performance, provided that the model of the data is adequate for the workload.