Cassandra CQLSH workshop

Install Cassandra on your laptop:

There are many ways to install Cassandra, however, I recommend using the docker image. To install docker follow the instructions of the following link:
https://docs.docker.com/desktop/install/linux-install/

Once docker is installed, follow the instructions of the following link to retrieve the image of Cassandra and to run it:
https://cassandra.apache.org/doc/latest/cassandra/getting_started/installing.html

Executing the following command in a terminal will retrieve the Cassandra's image:
`docker pull cassandra:latest`

To run the Cassandra database: `docker run --name cass_cluster cassandra:latest`

To run the Cassandra Query Language (CQL) shell in a second terminal tab:
`docker exec -it cass_cluster cqlsh`

For those who do not want to install Docker and Cassandra on their laptops, a cloud version provided by DATASTAX exists, see the following link:
https://awesome-astra.github.io/docs/pages/astra/create-instance/

In the rest of this workshop, the presented commands have been tested on the docker version, but they should work on the cloud version as well.

Create a keyspace:

In this workshop, you will create a Cassandra database to store data received form sensor networks.

First let's create the Keyspace of the database. The keyspace allows the user to configure how its data is replicated.

```
CREATE KEYSPACE sensor_data WITH replication = {'class':
'SimpleStrategy', 'replication_factor' : 3};
```

To verify that the keyspace has been created you can execute the describe command to list the available keyspaces:

```
cqlsh:sensor_data> Describe KEYSPACES;

sensor_data  system_auth         system_schema  system_views
system       system_distributed  system_traces  system_virtual_schema
```

You might have noticed that when you created the keyspace sensor_data, the prompt returned a warning saying the number of available nodes is lower than the replication factor. Indeed, in

your cluster you have just one node, your laptop. Usually, the replicas are saved on different nodes, different racks and even different datacenters to maximize their availability.

To use the newly created keyspace, execute: **USE sensor_data;**

You can notice how the prompt display informs you that you are using the sensor_data keyspace.

You can also notice that all the reserved words such as DESC, USE, SELECT, etc. are written in capital letters and each instruction ends with a semicolon.

Create tables:

Let's create the first table of our database that stores the networks which have a name, a description and a location. Copy/paste the following command into your CQL console at the prompt. Try to identify the primary key, the partition key and the clustering columns (if any) for this table in the command:

```
CREATE TABLE IF NOT EXISTS networks (
  name TEXT,
  description TEXT,
  region TEXT,
  PRIMARY KEY ((name))
);
```

There are three columns in the networks table and each of them is of type text.

Then ***DESCRIBE*** your keyspace tables to ensure that the networks table has been created:
DESC TABLES;

A network will contain several sensors. Sensors are uniquely identified by their name, such as s1001. The application interacting with this database should be able to retrieve all the sensors in a given network, sorted by the sensor name. Therefore, you must create a table that contains the sensors grouped by network. In Cassandra's data model, this translates to storing the sensors of each network on a different partition. Only one partition will be accessed to retrieve the sensors of a given network. For performance purposes, you must minimize the number of partitions accessed by a query.

Give the create instruction that creates a table to store the sensors grouped by network. The table should contain the following columns: network, sensor, latitude, longitude and a MAP of characteristics (accuracy and sensitivity). The partition key should be the network name and the clustering key the sensor name. Since in Cassandra there is no referential integrity, is up to you to programmatically check if a network name exists in the networks table before using it in the sensors_by_network table.

Now we can query the networks and their description (networks table), then retrieve the sensors per network (sensors_by_network table).

To store the temperatures measured by the sensors, you will need to create another table. Give the create instruction to create the temperatures_by_sensor table. In this table, each

measurement has a value, a timestamp (when it was taken) and the name of the sensor that took it. The partition key is the sensor's name which means that all the temperatures sensed by a given sensor are stored in the same partition. The data must be ordered by timestamp in a descending order.

The main issue in this table is that the size of the partitions will explode as the sensors send more and more measurements. There is a good practice rule stating that the upper limit for a partition is 100MB or 100k records. If a partition exceeds one of these limits, you need to split values across multiple partitions. This technique is called *bucketing*. To split data across multiple partitions a column must be added to the partition key. For example, the measurements of a given sensor are grouped per day and stored on a different partition.

Drop the temperatures_by_sensor table and recreate another one while applying the bucketing technique. The date column must be added to the schema of the table and to its partition key to split data across multiple partitions.

Insert operations:

1. Insert the following two rows to the networks table:
   ('forest-net', 'forest fire detection network', 'south')
   ('volcano-net', 'volcano monitoring network', 'north')

   The table networks contains three text columns. Whereas the table sensors_by_network contains a column of type MAP<> which lets you define your own key/value mapping.

2. Insert the following rows to the sensors_by_network table:

('forest-net','s1001',30.526503,-95.582815,{'accuracy':'medium','sensitivity':'high'});
('forest-net','s1002',30.518650,-95.583585,{'accuracy':'medium','sensitivity':'high'});
('forest-net','s1003',30.515056,-95.556225, {'accuracy':'medium','sensitivity':'high'});
('volcano-net','s2001',44.460321,-110.828151,{'accuracy':'high','sensitivity':'medium'});
('volcano-net','s2002',44.463195,-110.830124,{'accuracy':'high','sensitivity':'medium'});

3. Finally, some data to add to the temperatures_by_sensor table:

```
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-04','2020-07-04 00:00:01',80);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-04','2020-07-04 00:59:59',79);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-04','2020-07-04 12:00:01',97);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-04','2020-07-04 12:59:59',98);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-04','2020-07-04 00:00:01',82);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-04','2020-07-04 00:59:59',80);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-04','2020-07-04 12:00:01',100);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-04','2020-07-04 12:59:59',100);
```

```
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-04','2020-07-04 00:00:01',81);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-04','2020-07-04 00:59:59',80);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-04','2020-07-04 12:00:01',99);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-04','2020-07-04 12:59:59',98);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-05','2020-07-05 00:00:01',81);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-05','2020-07-05 00:59:59',80);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-05','2020-07-05 12:00:01',98);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-05','2020-07-05 12:59:59',99);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-05','2020-07-05 00:00:01',82);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-05','2020-07-05 00:59:59',82);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-05','2020-07-05 12:00:01',100);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-05','2020-07-05 12:59:59',99);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-05','2020-07-05 00:00:01',83);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-05','2020-07-05 00:59:59',82);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-05','2020-07-05 12:00:01',101);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-05','2020-07-05 12:59:59',102);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-06','2020-07-06 00:00:01',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-06','2020-07-06 00:59:59',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-06','2020-07-06 12:00:01',106);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1001','2020-07-06','2020-07-06 12:59:59',107);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-06','2020-07-06 00:00:01',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-06','2020-07-06 00:59:59',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-06','2020-07-06 12:00:01',108);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1002','2020-07-06','2020-07-06 12:59:59',110);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-06','2020-07-06 00:00:01',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-06','2020-07-06 00:59:59',90);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-06','2020-07-06 12:00:01',1315);
INSERT INTO temperatures_by_sensor (sensor,date,timestamp,value)
  VALUES ('s1003','2020-07-06','2020-07-06 12:59:59',1429);
```

Read operations:

READ Operations are done with **SELECT** statements, for example: **SELECT * FROM sensors_by_network;**
However, you should avoid these queries because no partition key is selected in the where clause. Therefore, all the partitions must be read, and the query is clearly not performant.

A more appropriate query is to select the sensors in a given network.

4. Give the instruction to select the name, characteristics, latitude and longitude of the sensors belonging to the forest-net network.

5. Give the query to select the temperatures measured by sensor s1002 on the 2020-07-05 date.

   What happens if we omit the date constraint?

   You should always provide, in the where clause, constraints on all the columns of the partition key. If you want to force the execution of this query you have to use the ALLOW FILTERING option.

6. Try to get the maximum temperature measured by each sensor on each date using the "group by" option and the "MAX()" function.

7. Try to get all the networks in the north region from table networks.
   To be able to add constraints on a column that is not part of the primary key, you should create a secondary index for that column:
   CREATE INDEX regionIndex ON networks (region);

   After the creation of the secondary column, try to execute the select query again.

Update operations:

8. Give the update instruction to change the value of the temperature measured by sensor s1002 on the 05/07/2020 12:59:59 to 92°F.

   you could also achieve the same result with another INSERT statement, which will simply overwrite the previous values if the partition key is the same. This is because Cassandra does not read before writing, i.e. updates are inserts!

   Verify with a Select that the temperature was updated.

Delete operations:

The delete operation removes data from the database. In Apache Cassandra, using the same delete command a single column or even a whole partition can be removed.

Generally speaking, it's best to perform as few delete operations as possible on the largest amount of data. Think of it this way, if you want to delete ALL data in a table, don't delete each individual cell, just **TRUNCATE** the table. If you need to delete all the rows in a partition, don't delete each row, **DELETE** the partition, and so on.

When deleting a row on a given table, we have to specify the values of the primary key for that table. Don't forget that, if your data model has the same information stored twice in different tables, it will be up to you to issue two different **DELETE** operations.

9. Remove all the temperatures measured by the sensor s1002 on the 05/07/2020 date. Verify that this partition has been deleted.

10. Remove the temperature measured by the sensor s1002 on the 04/07/2020 date and has the following timestamp: 2020-07-04 00:00:01.

What must be done to be able to answer these queries:

1. Find hourly average temperatures.
2. Find hourly average temperatures for every sensor.
3. Find hourly average temperatures in a specified network.
4. Find hourly average temperatures for a given date range.