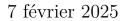


# NoSQL - Cassandra

## ESILV A4 - DIA2

Élèves :

Nour AFFES Hugo BONNELL Lucas BLANCHET Rayan HAMADEH





## Table des matières

1	$\operatorname{Cre}$	ate a database using Cassandra	2			
2	Simple Queries					
	2.1	Listing the restaurants	7			
	2.2	Get inspection details from a specific restaurant	7			
	2.3	Count the total number of inspections				
	2.4	Get the average inspection score	8			
	2.5	Count the restaurants that are in Manhattan	8			
	2.6	Get the maximum score of an inspection	9			
3	Cor	nplex Queries	9			
	3.1	Get the number of inspections per restaurant	9			
	3.2	Get the average score of inspections per restaurant				
4	Har	ed Query	10			
	4.1	Get restaurant details with their inspection date	10			



## 1 Create a database using Cassandra

#### Running Cassandra

After Installing its image onto our computer, we run Cassandra as a container using the command :

```
docker run --rm -d --name Cassandra -p 127.0.0.1:9042:9042 cassandra
```

#### Modeling the table

Either using TablePlus or cqlsh from the exec panel of the container, we create the keyspace and tables that will hold the database.

Creating the keyspace:

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

USE restaurant_inspections;
```

Creating the tables, according to the model of the JSON file. We create a table 'restaurant' holding informations about the restaurants, and a table inspections, with the inspections informations. Here is an example of how we created these tables:

```
CREATE TABLE inspections (
idRestaurant INT PRIMARY KEY,
inspectionDate DATE,
violationCode TEXT,
violationDescription TEXT,
criticalFlag TEXT,
score INT,
grade TEXT,
idInspection UUID,
PRIMARY KEY (idRestaurant, idInspection)
);
```



#### Sidenote: There's no Foreign Keys in Cassandra! Why?

- 1. **Distributed Architecture**: Verifying foreign key constraints in a distributed system would require cross-node communication, which can be slow and inefficient.
- 2. **Denormalized Data**: Cassandra encourages duplicating data (denormalization) to optimize for read performance rather than maintaining relational integrity.

#### Inserting the data

Cassandra's COPY command works with CSV files, so we first need to convert our JSON to CSV. We can use python to do so!

Our InspectionsRestaurant dataset has a ndjson format, not json. Thus, to avoid problems the python conversion function first converts the file to json before converting the json to csv. Direct conversion from NDJSON to CSV caused problems so that's how we overcame the issue!

#### The workflow of our algorithm is the following:

- Turn the NDJSON file to a JSON file (easier to deal with when it comes to making a CSV).
- Turning the JSON to one big CSV with all the data flattened.
- Using pandas dataframes and to csv commands, turn the newly created CSV into various CSV files that will create the tables.

```
import json, csv, os, uuid
      import pandas as pd
      def main():
           # Input NDJSON file and output CSV file
           # By default, set to the dataBatch for testing purposes,
              can be changed to your actual (or full) dataset !
           input_file = '../../Bigdata/InspectionsRestaurant.json'
           output_file = '../../Bigdata/InspectionsRestaurant.csv'
           output_folder = '../../Bigdata/'
9
           # STEP 1
           # Turn the NDJSON file to a JSON file.
           converted_file = ndjson_to_json(input_file)
16
17
           # STEP 2
18
```



```
#
19
           # Open the JSON file and the CSV file
20
           with open(converted_file, 'r') as json_file, open(
              output_file, 'w', newline='', encoding='utf-8') as
              csv_file:
               # Parse the JSON array from the file
               records = json.load(json_file)
24
               # Define the field names for the CSV file
25
               fieldnames = [
26
                    'idRestaurant', 'name', 'borough', 'buildingnum',
                        'street', 'zipcode', 'phone',
                    'cuisineType', 'inspectionDate', 'violationCode',
28
                        'violationDescription',
                    'criticalFlag', 'score', 'grade'
29
               ]
30
               writer = csv.DictWriter(csv_file, fieldnames=
31
                  fieldnames)
               writer.writeheader()
32
33
               # Process each record in the JSON array
34
               total_records = len(records)
35
               for counter, record in enumerate(records, start=1):
                   print(f'Processing... {round((counter /
                       total_records) * 100, 2)}%')
                   writer.writerow({
38
                        'idRestaurant': record['idRestaurant'],
39
                        'name': record['restaurant']['name'],
40
                        'borough': record['restaurant']['borough'],
41
                        'buildingnum': record['restaurant']['
42
                           buildingnum'],
                        'street': record['restaurant']['street'],
43
                        'zipcode': record['restaurant']['zipcode'],
44
                        'phone': record['restaurant']['phone'],
45
                        'cuisineType': record['restaurant']['
46
                           cuisineType'],
                        'inspectionDate': record['inspectionDate'],
47
                        'violationCode': record.get('violationCode',
48
                           ''),
                        'violationDescription': record.get('
                           violationDescription', ''),
                        'criticalFlag': record['criticalFlag'],
                        'score': record['score'],
                        'grade': record['grade']
                   })
54
```



```
print(f"Conversion complete! CSV file saved as '{
              output file}'.")
56
           #
57
           # STEP 3
58
           #
59
           # Load the newly created CSV file
60
           df = pd.read_csv(output_file)
61
62
           # Define column subsets
63
           inspections_columns = ["idRestaurant", "inspectionDate",
              "violationCode", "violationDescription", "criticalFlag"
              , "score", "grade"]
           restaurant_columns = ["idRestaurant", "name", "borough",
              "buildingnum", "street", "zipcode", "phone", "
              cuisineType"]
           # Extract tables
67
           inspections = df[inspections_columns].copy()
68
           restaurants = df[restaurant_columns].copy()
           # Generate UUIDs for each inspection row
           inspections["idInspection"] = [uuid.uuid4() for _ in
              range(len(inspections))]
73
           # Save to new CSV files
74
           inspections.to_csv(f"{output_folder}inspections.csv",
              index=False)
           restaurants.to_csv(f"{output_folder}restaurants.csv",
              index=False)
           # Last CSV for the Hard Query... see Hard Query section !
78
           inspections_restaurants_columns = ['idRestaurant', 'name'
79
              , 'borough', 'inspectionDate', 'grade']
           inspections_restaurants = df[
80
              inspections_restaurants_columns]
           inspections_restaurants.to_csv(f"{output_folder}
81
              inspections_restaurants.csv", index=False)
82
           print("Files saved.")
       # FUNCTION USED BY STEP 1
85
       def ndjson_to_json(ndjson_file):
86
87
           # Output JSON file
88
           json_file = os.path.splitext(ndjson_file)[0] + '.json'
```



```
90
            # Read the NDJSON file and convert to a JSON array
91
           data = []
92
93
           with open(ndjson_file, 'r') as infile:
94
                counter = 0
95
                for line in infile:
96
                    counter += 1
97
                    if line.strip(): # Skip empty lines
98
                        try:
99
                             # Parse each line as a JSON object
100
                             record = json.loads(line)
                             data.append(record)
                             print(f"Processing... line {counter}")
                        except json.JSONDecodeError as e:
104
                             print(f"Skipping invalid JSON line: {line
                                .strip()} - Error: {e}")
106
            # Write the JSON array to a file
107
           with open(json_file, 'w', encoding='utf-8') as outfile:
108
                json.dump(data, outfile, indent=4)
           print(f"Converted NDJSON file '{ndjson_file}' to JSON
               file '{json_file}'.")
           return json_file
```

Once the .csv file is generated, we can use the cql shell to load the data into Cassandra. We shall move the .csv file to cassandra's container (if using docker) and run this command in the cql shell :

• Moving the .csv file to Cassandra's container :

```
docker cp PATH_TO_FOLDER/your_file.csv Cassandra:/
```

• Copying the data from the CSV to the database :

```
COPY restaurants (idRestaurant, name, borough, buildingnum,
street, zipcode, phone, cuisineType)

FROM 'inspections.csv'
WITH HEADER = TRUE;

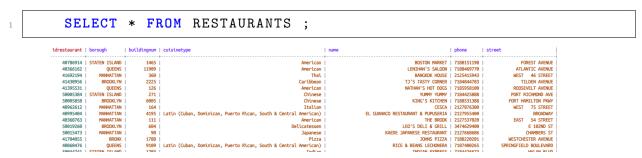
COPY inspections (idRestaurant, inspectionDate, violationCode, violationDescription, criticalFlag, score, grade)

FROM 'inspections.csv'
WITH HEADER = TRUE;
```



## 2 Simple Queries

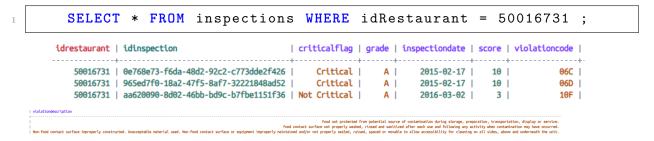
## 2.1 Listing the restaurants



This query retrieves the names and details of all the restaurants in the dataset. It's a simple SELECT query to show the data under the restaurant field of each record in the dataset.

It's a straightforward request for all restaurant details in the data without any filters or aggregation. This can be done by extracting the name, borough, buildingnum, street, zipcode, and phone fields from each restaurant record.

#### 2.2 Get inspection details from a specific restaurant



This query would retrieve the inspection details (e.g., violation code, description, date, score, grade) for a particular restaurant based on its idRestaurant. You would need a WHERE clause to filter by idRestaurant.

You can access inspection details for a specific restaurant by using the unique idRestaurant as a filter. This ensures that only the data of interest for the specified restaurant is retrieved, like violationCode, violationDescription, inspectionDate, etc.



## 2.3 Count the total number of inspections

```
SELECT COUNT(*) FROM inspections;

count
-----
442795
```

This query would simply count how many inspections are present in the dataset. You could count the records directly, as each entry corresponds to a unique inspection.

The COUNT(\*) function counts the number of entries in the dataset. Each restaurant inspection corresponds to one entry, so counting these will give you the total number of inspections.

#### 2.4 Get the average inspection score

```
SELECT AVG(score) FROM inspections;

system.avg(score)

17
```

This query calculates the average inspection score across all restaurants. By using the AVG(score) function, you will get the mean score value.

The dataset has a score field that indicates the inspection score for each restaurant. The AVG(score) function computes the average of all these scores to give a measure of how well restaurants are performing on average during their inspections.

#### 2.5 Count the restaurants that are in Manhattan

```
SELECT COUNT(*) FROM restaurants WHERE borough = 'MANHATTAN'
ALLOW FILTERING;
```

## 10407

To count how many restaurants are located in Manhattan, you can use a WHERE clause filtering by the borough field.

By filtering records where borough = 'MANHATTAN', you can count all restaurants within Manhattan. This is simply a count of how many records satisfy this condition.



## 2.6 Get the maximum score of an inspection

```
SELECT MAX(score) FROM inspections;

system.max(score)

156
```

This query finds the highest inspection score across all inspections in the dataset. Using the MAX(score) function will give the maximum inspection score.

The MAX(score) function gives the highest inspection score from the score field. This helps in identifying the best inspection result among all the restaurants.

## 3 Complex Queries

#### 3.1 Get the number of inspections per restaurant

```
SELECT idRestaurant, COUNT(*) FROM inspections GROUP BY
  idRestaurant;
```

idrestaurant		
40786914		
40366162	25	
41692194	50	
41430956	55	
41395531	4	
50005384	8	
50005858	17	
40962612	35	

This query requires counting the number of inspections per restaurant. Since each entry represents one inspection, we can group by idRestaurant and count the number of inspections for each restaurant.

Justification: To achieve this, you can group the data by the idRestaurant field and use the COUNT(\*) function. This will provide the number of inspections per restaurant.



### 3.2 Get the average score of inspections per restaurant

```
SELECT idRestaurant, AVG(score) FROM inspections GROUP BY
idRestaurant;
```

```
| system.avg(score) | 49786914 | 10 | 49366162 | 18 | 41692194 | 26 | 41430956 | 30 | 41395531 | 6 | 50005384 | 9 | 50005858 | 9 | 40962612 | 19
```

This query calculates the average score of inspections for each restaurant. By grouping by idRestaurant and using AVG(score), you'll get the average inspection score for each restaurant.

Justification: To find the average inspection score for each restaurant, you group by idRestaurant and apply the AVG(score) function. This will give you a breakdown of how each restaurant is performing on average during their inspections.

## 4 Hard Query

## 4.1 Get restaurant details with their inspection date

First we create a denormalized table.

```
CREATE TABLE restaurants_inspections(

idRestaurant INT,

name TEXT,

borough TEXT,

inspectionDate TEXT,

grade TEXT,

PRIMARY KEY (idRestaurant, inspectionDate)

WITH CLUSTERING ORDER BY (inspectionDate DESC);
```

Remember, data was preprocessed into a inspections restaurants.csv file. See these lines from the preprocessing code :

```
COPY restaurants_inspections(idRestaurant, name, borough, inspectionDate, grade)
FROM 'restaurants_inspections.csv';
```



#### Then, simply:

SELECT \* FROM restaurants\_inspections;

idrestaurant	inspectiondate	borough	_		
40786914	2016-08-10	STATEN ISLAND		•	BOSTON MARKET
40786914	2015-07-01	STATEN ISLAND	Α		BOSTON MARKET
40786914	2014-07-10	STATEN ISLAND	Α		BOSTON MARKET
40366162	2016-05-11	QUEENS	Α	LEN	IHAN'S SALOON
40366162	2015-04-23	QUEENS	Α	LEN	IHAN'S SALOON
	2014-11-20				IHAN'S SALOON
40366162	2014-10-29	QUEENS	null	LEN	IHAN'S SALOON
40366162	2014-02-15	QUEENS	Α	LEN	IHAN'S SALOON
40366162	2013-12-24	QUEENS	null	LEN	IHAN'S SALOON
40366162	2013-07-01	QUEENS	В	LEN	IHAN'S SALOON
40366162	2013-06-11	QUEENS	null	LEN	IHAN'S SALOON
41692194	2016-01-27	MANHATTAN	Α		BANGKOK HOUSE
41692194	2015-07-14	MANHATTAN	Α		BANGKOK HOUSE
41692194	2015-07-01	MANHATTAN	null		BANGKOK HOUSE
41692194	2014-12-29	MANHATTAN	Α		BANGKOK HOUSE
41692194	2014-12-04	MANHATTAN	null		BANGKOK HOUSE
41692194	2014-07-11	MANHATTAN	C		BANGKOK HOUSE
41692194	2014-06-09	MANHATTAN	null		BANGKOK HOUSE
41692194	2013-12-12	MANHATTAN	Α		BANGKOK HOUSE
41692194	2013-12-03	MANHATTAN	null		BANGKOK HOUSE
41692194	2013-06-10	MANHATTAN	В		BANGKOK HOUSE
41692194	2013-04-26	MANHATTAN	P		BANGKOK HOUSE

This query requires fetching the restaurant details along with the corresponding inspection date. It would combine both restaurant information and the inspectionDate for each entry in the dataset.

Justification: You will need to select the restaurant fields (like name, borough, etc.) and also include the inspectionDate for each record. This will give a comprehensive list of restaurants alongside their most recent inspection date, showing how the inspection timeline aligns with restaurant information.