NoSQL - Cassandra

ESILV A4 - DIA2

Élèves :
Nour AFFES
Hugo BONNELL
Lucas BLANCHET
Rayan HAMADEH

Enseignants :
Jihane MALI
Pierre LEFEBVRE

6 février 2025

Table des matières

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 table that will hold the database.

Creating the keyspace :

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

USE restaurant_inspections;
```

Creating the table, according to the model of the JSON file

```
CREATE TABLE inspections (
    idRestaurant INT PRIMARY KEY,
    name TEXT,
    borough TEXT,
    buildingnum TEXT,
    street TEXT,
    zipcode TEXT,
    phone TEXT,
    cuisineType TEXT,
    inspectionDate DATE,
    violationCode TEXT,
    violationDescription TEXT,
    criticalFlag TEXT,
    score INT,
    grade TEXT
);
```

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 !

```python
    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 = '../../Database/dataBatch.ndjson'
        output_file = '../../Database/dataBatch.csv'

        # Turn the NDJSON file to a JSON file. (This step is needed if the input is an NDJSON file)
        converted_file = ndjson_to_json(input_file)

        # Open the JSON file and the CSV file
        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)

            # Define the field names for the CSV file
            fieldnames = [
                'idRestaurant', 'name', 'borough', 'buildingnum', 'street', 'zipcode', 'phone',
                'cuisineType', 'inspectionDate', 'violationCode', 'violationDescription',
                'criticalFlag', 'score', 'grade'
            ]
            writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
            writer.writeheader()

            # Process each record in the JSON array
            total_records = len(records)
            for counter, record in enumerate(records, start=1):
                print(f'Processing... {round((counter / total_records) * 100, 2)}%')
                writer.writerow({
                    'idRestaurant': record['idRestaurant'],
                    'name': record['restaurant']['name'],
                    'borough': record['restaurant']['borough'],
                    'buildingnum': record['restaurant']['buildingnum'],
                    'street': record['restaurant']['street'],
                    'zipcode': record['restaurant']['zipcode'],
                    'phone': record['restaurant']['phone'],
                    'cuisineType': record['restaurant']['cuisineType'],
                    'inspectionDate': record['inspectionDate'],
                    'violationCode': record.get('violationCode', ''),
                       'violationDescription': record.get('violationDescription', ''),
                    'criticalFlag': record['criticalFlag'],
                    'score': record['score'],
                    'grade': record['grade']
                })

        print(f"Conversion complete! CSV file saved as '{output_file}'.")


    def ndjson_to_json(ndjson_file):

        # Output JSON file
        json_file = os.path.splitext(ndjson_file)[0] + '.json'

        # Read the NDJSON file and convert to a JSON array
        data = []

        with open(ndjson_file, 'r') as infile:
            counter = 0
            for line in infile:
                counter += 1
                if line.strip():  # Skip empty lines
                    try:
                        # Parse each line as a JSON object
                        record = json.loads(line)
                        data.append(record)
                        print(f"Processing... line {counter}")
                    except json.JSONDecodeError as e:
                        print(f"Skipping invalid JSON line: {line.strip()} - Error: {e}")

        # Write the JSON array to a file
        with open(json_file, 'w', encoding='utf-8') as outfile:
            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 :**

```sql
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

```sql
SELECT * FROM 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

```sql
SELECT * FROM inspections WHERE idRestaurant = 50016731 ;
```

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

```sql
SELECT COUNT(*) FROM inspections;
```

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

```sql
SELECT AVG(score) FROM inspections;
```

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

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

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

```sql
    SELECT MAX(score) FROM inspections;
```

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;
```

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;
```

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);
```

Then we preprocess the data and load it into cassandra :

```python
import pandas as pd
import uuid

# Load CSV file
df = pd.read_csv("dataBatch.csv")

# Define column subsets
inspections_columns = ["idRestaurant", "inspectionDate", "
    violationCode", "violationDescription", "criticalFlag", "
    score", "grade"]
restaurant_columns = ["idRestaurant", "name", "borough", "
    buildingnum", "street", "zipcode", "phone", "cuisineType"]

# Extract tables
inspections = df[inspections_columns].copy()
restaurants = df[restaurant_columns].copy()

# Generate UUIDs for each inspection row
inspections["idInspection"] = [uuid.uuid4() for _ in range(
    len(inspections))]

# Save to new CSV files
inspections.to_csv("inspections.csv", index=False)
restaurants.to_csv("restaurants.csv", index=False)
```

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

Then, simply :

```sql
SELECT * FROM restaurants_inspections;
```

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.