TP2 – NOSQL CASSANDRA RESTAURANT INSPECTIONS

Manon GARDIN

Matias OTTENSEN

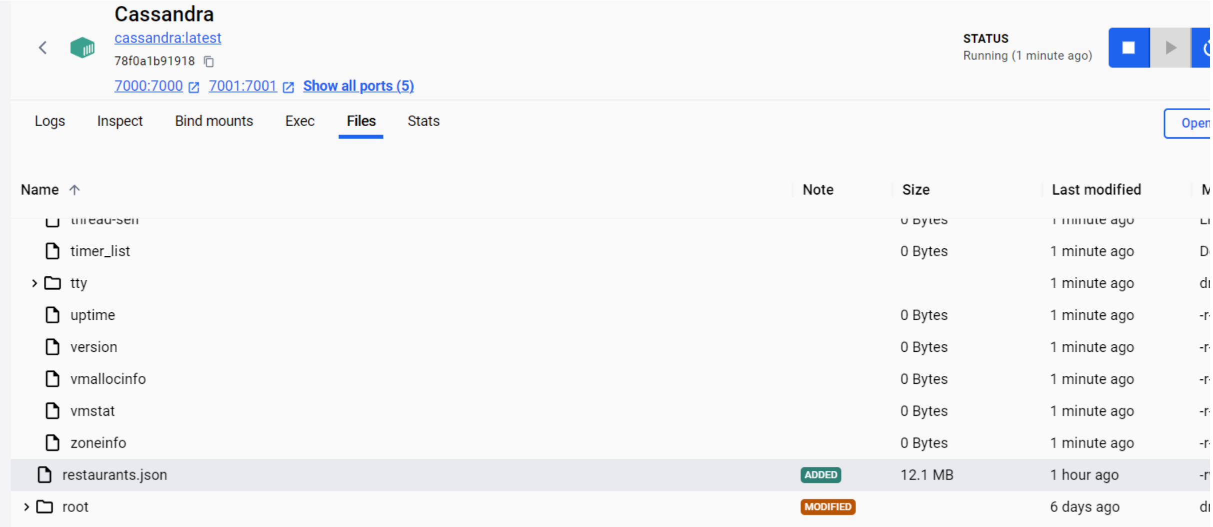
Alexandre GARNIER

Tiphaine KACHKACHI

Chapter 1 – Create the database

Files transfer

We drag and drop the restaurants.json into the files of our Cassandra container.



Name	Note	Size	Last modified	M
thread-self		0 Bytes	1 minute ago	Li
timer_list		0 Bytes	1 minute ago	D
tty			1 minute ago	dr
uptime		0 Bytes	1 minute ago	-r
version		0 Bytes	1 minute ago	-r
vmallocinfo		0 Bytes	1 minute ago	-r
vmstat		0 Bytes	1 minute ago	-r
zoneinfo		0 Bytes	1 minute ago	-r
restaurants.json	ADDED	12.1 MB	1 hour ago	-r
root	MODIFIED		6 days ago	dr

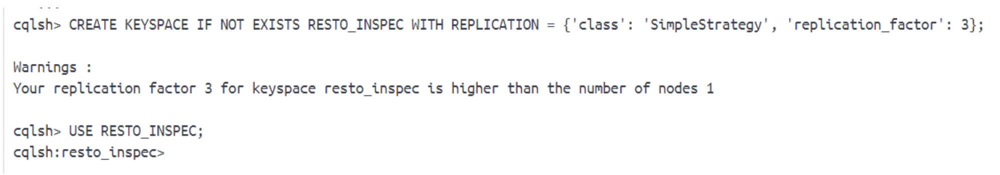
Create the keyspace

In the CLI, use the command :

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

And then,

```
USE RESTO_INSPEC;
```



```
   ...
cqlsh> CREATE KEYSPACE IF NOT EXISTS RESTO_INSPEC WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 3};

Warnings :
Your replication factor 3 for keyspace resto_inspec is higher than the number of nodes 1

cqlsh> USE RESTO_INSPEC;
cqlsh:resto_inspec>
```

The `CREATE KEYSPACE IF NOT EXISTS RESTO_INSPEC` command creates a new namespace called RESTO_INSPEC in Cassandra, which is used to organize and isolate tables from the database, only if the namespace does not already exist. The `WITH REPLICATION = { 'class': 'SimpleStrategy', 'replication_factor': 3 }` clause defines the replication strategy to ensure data reliability and availability by replicating it on three different nodes. Next, the use of `USE RESTO_INSPEC;` selects this keyspace, allowing the creation of tables and the insertion of data into the specified namespace.

Create Tables

Let’s understand **the JSON structure** of our dataset with the structure of one insert :

```
[
        {
                "address":
                        {
                                "building": "1007",
                                "coord": {
                                        "type": "Point",
                                        "coordinates" : [-73.856077, 40.848447]
                                },
                                "street": "Morris Park Ave",
                                "zipcode": "10462"
                        },
                "borough": "Bronx",
                "cuisine": "Bakery",
                "grades": [
                                {
                                        "date": {
                                                "$date": 1393804800000
                                        },
                                        "grade": "A",
                                        "score": 2
                                },
                                {
                                        "date": {
                                                "$date": 1378857600000
                                        },
                                        "grade": "A",
                                        "score": 6
                                },
                                {
                                        "date": {
                                                "$date": 1358985600000
                                        },
                                        "grade": "A",
                                        "score": 10
                                },
                                {
                                        "date": {
```

```
                                        "$date": 1322006400000
                                  },
                                  "grade": "A",
                                  "score": 9
                           },
                           {
                                  "date": {
                                        "$date": 1299715200000
                                  },
                                  "grade": "B",
                                  "score": 14
                           }
                    ],
              "name":  "Morris Park Bake Shop",
              "restaurant_id": "30075445"
       },
       INSERT2…
]
```

Visually, we can see that the first property address will need a full table containing building, coord, street and zipcode. In each coord, we have a type of coordinate and the X and Y coordinates. In order to not create another table, we can add them as different attributes.

Also the property grade is an array with different elements. We can conclude that we will need a grades table with the three attributes grade, date and score. One restaurant can have several grades, so we add the restaurant_id attributes as the equivalent of a foreign key.

And finally the table restaurants that will have all the other attributes.

We chose to add the restaurant_id attributes in every table so that we don't need to find a way to join the table in the queries.

Create the tables in file CreaTable.sql :

```
CREATE TABLE restaurants (
      restaurant_id text PRIMARY KEY,
      name text,
      borough text,
      cuisine text
);
ALTER TABLE restaurants WITH GC_GRACE_SECONDS=0;


CREATE TABLE addresses (
    address_id text PRIMARY KEY,
    building text,
    street text,
    zipcode text,
    coord_type text,
    coord_X float,
    coord_Y float
);
ALTER TABLE addresses WITH GC_GRACE_SECONDS=0;

CREATE TABLE grades (
    restaurant_id text,
    date timestamp,
    grade text,
    score int,
    PRIMARY KEY (restaurant_id, date)
);
ALTER TABLE grades WITH GC_GRACE_SECONDS=0;
```

Then, we added some additional tables, like restaurants_by_borough which uses two partitions keys and will allow us to use a GROUP BY. We will see this in the next complex queries.

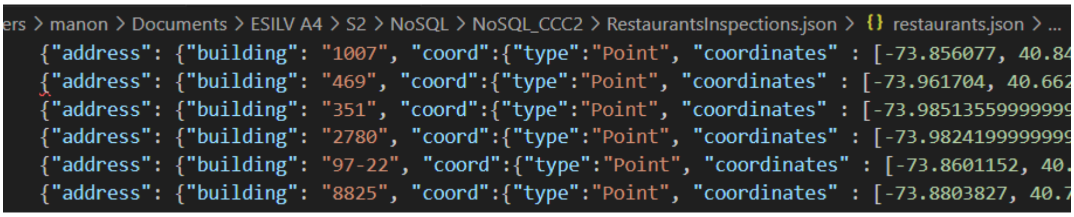
```
CREATE TABLE restaurants_by_borough (
    borough text,
    restaurant_id text,
    PRIMARY KEY (borough, restaurant_id)
);
ALTER TABLE restaurants_by_borough WITH GC_GRACE_SECONDS = 0;
```

And restaurants_with_scores that will contain the restaurants' information and their last score and last grade. This will allow us to do the equivalent of a join query.

```
CREATE TABLE restaurants_with_scores (
    restaurant_id text PRIMARY KEY,
    name text,
    borough text,
    cuisine text,
    last_score int,
    last_grade text
);
ALTER TABLE restaurants_with_scores WITH GC_GRACE_SECONDS = 0;
```

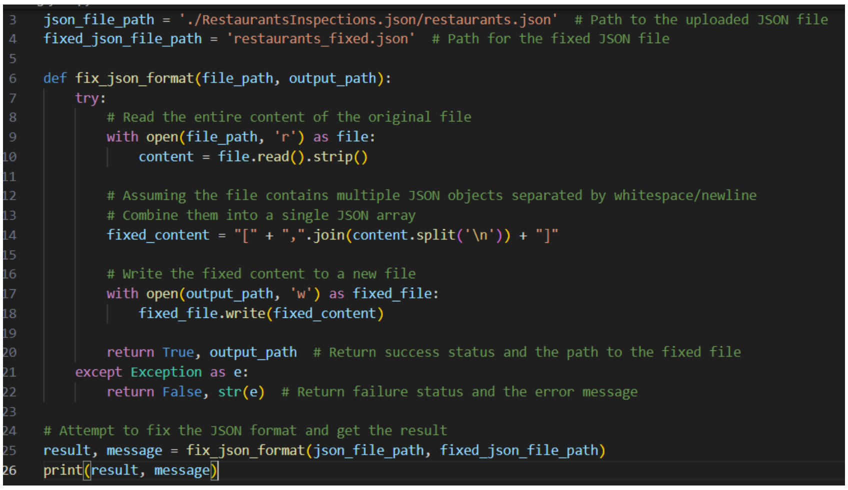
Fixing Json file

We found out that the format of the Json is not correct, so we needed to do a script to correct the file. What was wrong in that file ?


ers > manon > Documents > ESILV A4 > S2 > NoSQL > NoSQL_CCC2 > RestaurantsInspections.json > {} restaurants.json > ...

```
{"address": {"building": "1007", "coord":{"type":"Point", "coordinates" : [-73.856077, 40.84
{"address": {"building": "469", "coord":{"type":"Point", "coordinates" : [-73.961704, 40.662
{"address": {"building": "351", "coord":{"type":"Point", "coordinates" : [-73.98513559999999
{"address": {"building": "2780", "coord":{"type":"Point", "coordinates" : [-73.98241999999999
{"address": {"building": "97-22", "coord":{"type":"Point", "coordinates" : [-73.8601152, 40.
{"address": {"building": "8825", "coord":{"type":"Point", "coordinates" : [-73.8803827, 40.7
```

The file lack the first and last '['. We did the fixing_json.py :



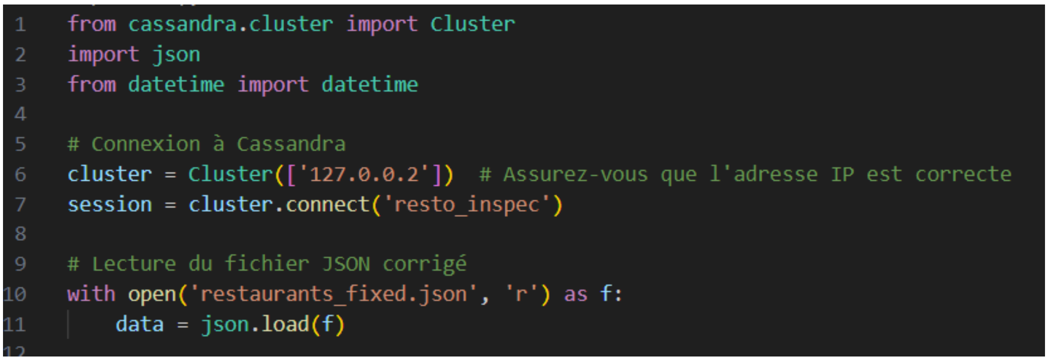
```python
json_file_path = './RestaurantsInspections.json/restaurants.json'  # Path to the uploaded JSON file
fixed_json_file_path = 'restaurants_fixed.json'  # Path for the fixed JSON file

def fix_json_format(file_path, output_path):
    try:
        # Read the entire content of the original file
        with open(file_path, 'r') as file:
            content = file.read().strip()

        # Assuming the file contains multiple JSON objects separated by whitespace/newline
        # Combine them into a single JSON array
        fixed_content = "[" + ",".join(content.split('\n')) + "]"

        # Write the fixed content to a new file
        with open(output_path, 'w') as fixed_file:
            fixed_file.write(fixed_content)

        return True, output_path  # Return success status and the path to the fixed file
    except Exception as e:
        return False, str(e)  # Return failure status and the error message

# Attempt to fix the JSON format and get the result
result, message = fix_json_format(json_file_path, fixed_json_file_path)
print(result, message)
```

In which we created a whole new file “restaurants_fixed.json”. **We added the '[' character and we join all the lines of the original files separated by a comma.**

After that, the data didn’t really need any kind of cleaning.

Import the data

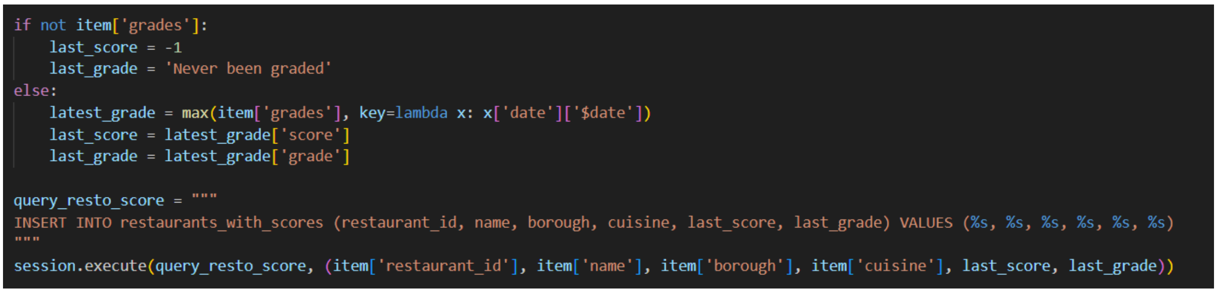
Then, we execute the data_importation.py file in the Cassandra container. In this code below, we setup the connection to the database :



```python
from cassandra.cluster import Cluster
import json
from datetime import datetime

# Connexion à Cassandra
cluster = Cluster(['127.0.0.2'])  # Assurez-vous que l'adresse IP est correcte
session = cluster.connect('resto_inspec')

# Lecture du fichier JSON corrigé
with open('restaurants_fixed.json', 'r') as f:
    data = json.load(f)
```

And then, we add the data with a query, for each table. Nothing special for the first three tables. We chose to not change any values, even if it is null.

But, for the additional tables, like the "restaurants_with_scores" one, we chose to take care of the null values, especially for the last_score and last_grade that will become -1 fir the score, and ‘Never been graded’ for the grade, that is easier to read and understand.



```python
if not item['grades']:
    last_score = -1
    last_grade = 'Never been graded'
else:
    latest_grade = max(item['grades'], key=lambda x: x['date']['$date'])
    last_score = latest_grade['score']
    last_grade = latest_grade['grade']

query_resto_score = """
INSERT INTO restaurants_with_scores (restaurant_id, name, borough, cuisine, last_score, last_grade) VALUES (%s, %s, %s, %s, %s, %s)
"""
session.execute(query_resto_score, (item['restaurant_id'], item['name'], item['borough'], item['cuisine'], last_score, last_grade))
```

Here is the command to execute the file in the container : `docker exec -it Cassandra python3 data_importation.py`

Chapter 2 – Querying Cassandra

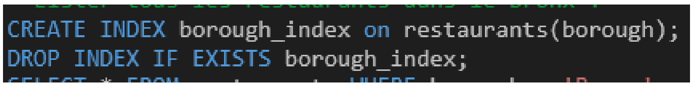
Simple Queries

List of restaurants located in Bronx :

*Query: SELECT * FROM restaurants WHERE borough = 'Bronx';*

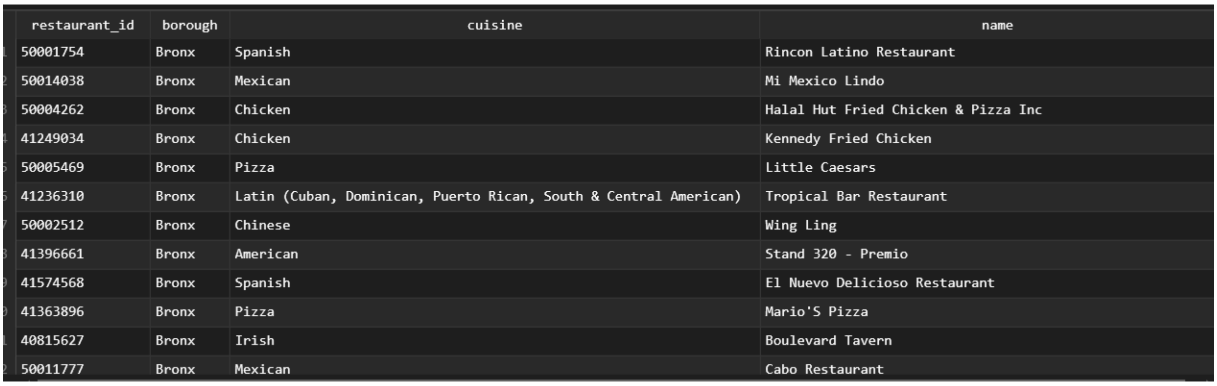
This query selects every column from the "restaurants" table where the value of the "borough" column is 'Bronx'.

This query won't be executed since it would take too much resources. We need to either use ALLOW FILTERING or create an index on the borough column of restaurants :



```
CREATE INDEX borough_index on restaurants(borough);
DROP INDEX IF EXISTS borough_index;
```

Result:



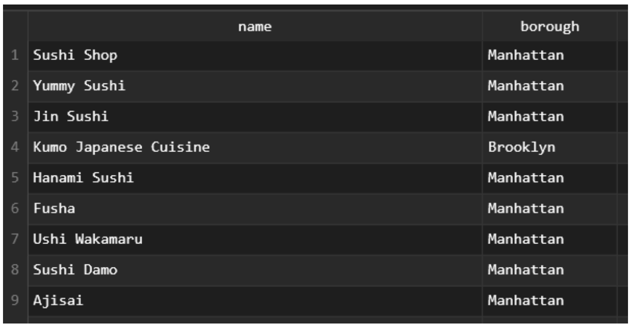
restaurant_id	borough	cuisine	name
50001754	Bronx	Spanish	Rincon Latino Restaurant
50014038	Bronx	Mexican	Mi Mexico Lindo
50004262	Bronx	Chicken	Halal Hut Fried Chicken & Pizza Inc
41249034	Bronx	Chicken	Kennedy Fried Chicken
50005469	Bronx	Pizza	Little Caesars
41236310	Bronx	Latin (Cuban, Dominican, Puerto Rican, South & Central American)	Tropical Bar Restaurant
50002512	Bronx	Chinese	Wing Ling
41396661	Bronx	American	Stand 320 - Premio
41574568	Bronx	Spanish	El Nuevo Delicioso Restaurant
41363896	Bronx	Pizza	Mario'S Pizza
40815627	Bronx	Irish	Boulevard Tavern
50011777	Bronx	Mexican	Cabo Restaurant

List of Japanese restaurants' name and borough

SELECT name, borough FROM restaurants WHERE cuisine='Japanese' ALLOW FILTERING;

This query selects the Japanese restaurants and more particularly the name and the borough in which they are located. For the same reason of the previous query, we use ALLOW FILTERING here.

Result : The borough of all Japanese restaurant.



name	borough
Sushi Shop	Manhattan
Yummy Sushi	Manhattan
Jin Sushi	Manhattan
Kumo Japanese Cuisine	Brooklyn
Hanami Sushi	Manhattan
Fusha	Manhattan
Ushi Wakamaru	Manhattan
Sushi Damo	Manhattan
Ajisai	Manhattan

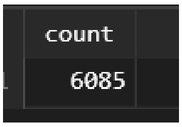
List the restaurants that are located in Brooklyn

Query: SELECT COUNT(*) FROM restaurants WHERE borough = 'Brooklyn';

This query counts the number of rows in the "restaurants" table where the value of the "borough" column is 'Brooklyn'.

Since we already have an index on borough that we created earlier, we don't need to use ALLOW FILTERING.

Result: We have 6805 restaurants located in Brooklyn.



	count
	6085

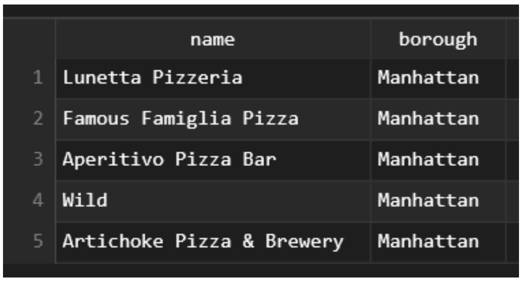
List 5 pizzerias of Manhattan

Query : SELECT name, borough FROM restaurants WHERE borough = 'Manhattan' AND cuisine = 'Pizza' LIMIT 5 ALLOW FILTERING;

This query selects the "name" and "borough" columns from the "restaurants" table where the value of the "borough" column is 'Manhattan' and "cuisine" is 'Pizza'.

We use LIMIT 5 to limit the result to 5 rows.

Result : The names and boroughs of the first 5 restaurants located in Manhattan.



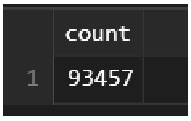
	name	borough
1	Lunetta Pizzeria	Manhattan
2	Famous Famiglia Pizza	Manhattan
3	Aperitivo Pizza Bar	Manhattan
4	Wild	Manhattan
5	Artichoke Pizza & Brewery	Manhattan

Get the number of grades attributed to restaurants

Query: SELECT COUNT(*) FROM grades;

This query will get the number of lines in the grades table.

Result: We have 93457 lines.



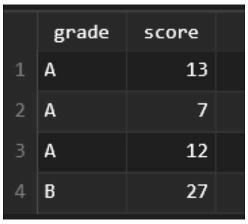
	count
1	93457

Get the grades of a specific restaurant

Query: SELECT grade, score FROM grades WHERE restaurant_id ='41701180';

We target a specific restaurant with its id in the WHERE condition.

Result: The restaurant with id 41701180 has been given 4 grades which are three A and one B.



	grade	score
1	A	13
2	A	7
3	A	12
4	B	27

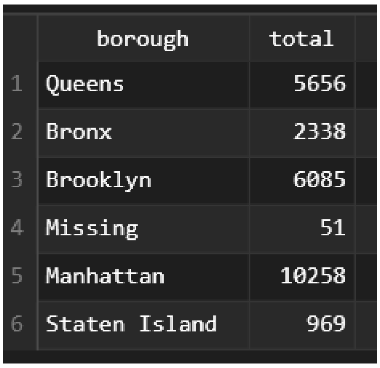
Complex queries

List of the borough and the number of restaurants in it

SELECT borough, COUNT(*) AS total FROM restaurants_by_borough GROUP BY borough;

For this query, we managed to use a GROUP BY on a new table “restaurants_by_borough”. This table have been created so that the GROUP BY on borough is possible. Borough is not a partition key on the original “restaurants” table. In this new table, it is.

Result:



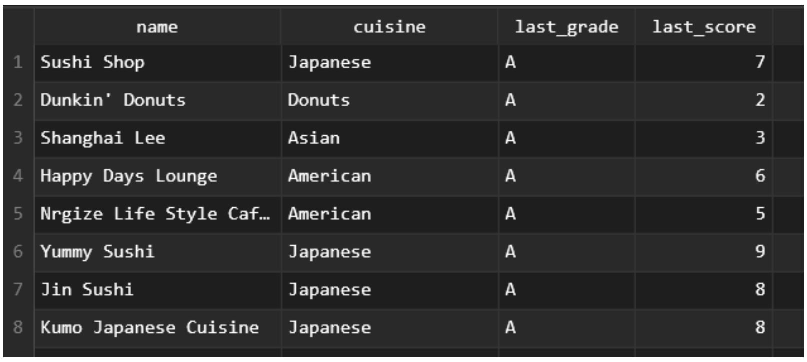
	borough	total
1	Queens	5656
2	Bronx	2338
3	Brooklyn	6085
4	Missing	51
5	Manhattan	10258
6	Staten Island	969

List of restaurants’ name and cuisine which had a really good last grade

Query: SELECT name, cuisine, last_grade, last_score FROM restaurants_with_scores WHERE last_grade = 'A' ALLOW FILTERING;

This query selects the "name" and "cuisine" columns from the "restaurants" table where the last grade is A. We used the table "restaurants_with_scores" that we created that contains the restaurants attributes and the grades attributes. We don’t forget to use ALLOW FILTERING because we have no indexes on last_grade.

Result: A list of restaurants' names and their cuisines with their grade (only A)

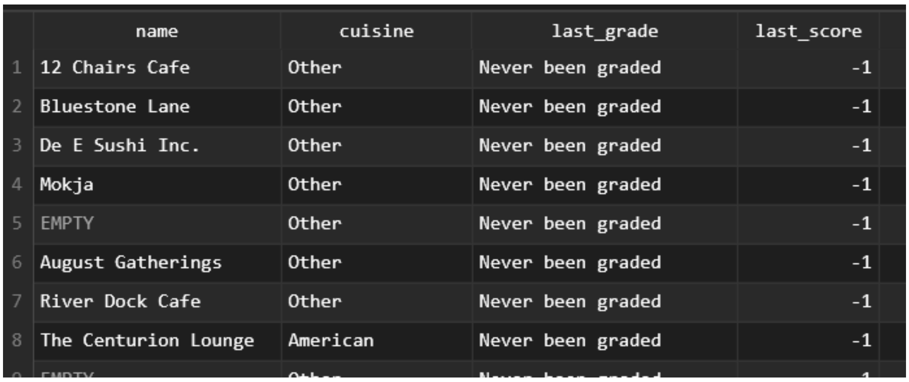


	name	cuisine	last_grade	last_score
1	Sushi Shop	Japanese	A	7
2	Dunkin' Donuts	Donuts	A	2
3	Shanghai Lee	Asian	A	3
4	Happy Days Lounge	American	A	6
5	Nrgize Life Style Caf…	American	A	5
6	Yummy Sushi	Japanese	A	9
7	Jin Sushi	Japanese	A	8
8	Kumo Japanese Cuisine	Japanese	A	8

The same way, we can observe the restaurants that have never been graded. The values were originally null, but we edited it in the importation file.

Query: SELECT name, cuisine, last_grade, last_score FROM restaurants_with_scores WHERE last_grade = 'Never been graded' ALLOW FILTERING;

Result :



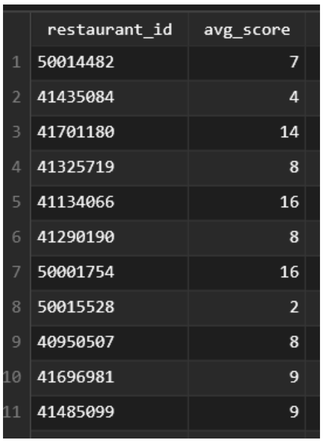
	name	cuisine	last_grade	last_score
1	12 Chairs Cafe	Other	Never been graded	-1
2	Bluestone Lane	Other	Never been graded	-1
3	De E Sushi Inc.	Other	Never been graded	-1
4	Mokja	Other	Never been graded	-1
5	EMPTY	Other	Never been graded	-1
6	August Gatherings	Other	Never been graded	-1
7	River Dock Cafe	Other	Never been graded	-1
8	The Centurion Lounge	American	Never been graded	-1
9	EMPTY	Other	Never been graded	-1

Calculate the average score obtained for each restaurant

SELECT restaurant_id, AVG(score) AS avg_score FROM grades GROUP BY restaurant_id;

In this query we used a group by and the function AVG to compute the average score of the restaurant.

Result:

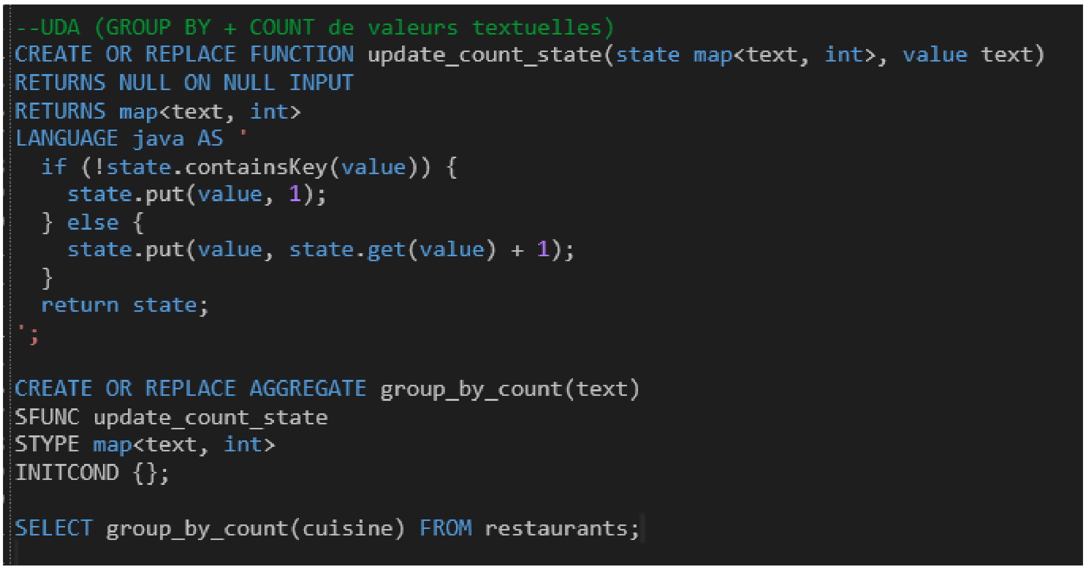


restaurant_id	avg_score
50014482	7
41435084	4
41701180	14
41325719	8
41134066	16
41290190	8
50001754	16
50015528	2
40950507	8
41696981	9
41485099	9

Hard queries

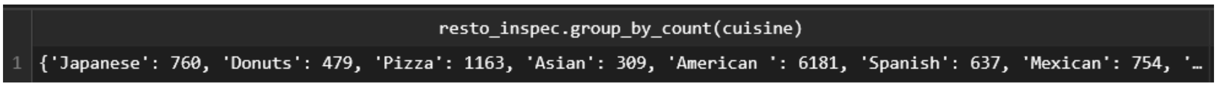
List the number of restaurants by cuisine

We created an UDA. We didn't need to create a TYPE because we used the MAP<text, int>. This function checks if the text value is already present in the map. If so, it increments the associated counter, otherwise it adds the value with a counter initialized to 1.



```
--UDA (GROUP BY + COUNT de valeurs textuelles)
CREATE OR REPLACE FUNCTION update_count_state(state map<text, int>, value text)
RETURNS NULL ON NULL INPUT
RETURNS map<text, int>
LANGUAGE java AS '
  if (!state.containsKey(value)) {
    state.put(value, 1);
  } else {
    state.put(value, state.get(value) + 1);
  }
  return state;
';

CREATE OR REPLACE AGGREGATE group_by_count(text)
SFUNC update_count_state
STYPE map<text, int>
INITCOND {};

SELECT group_by_count(cuisine) FROM restaurants;
```

Result : We have a Json result with the cuisine attributes and the number of restaurants associated.



```
resto_inspec.group_by_count(cuisine)
{'Japanese': 760, 'Donuts': 479, 'Pizza': 1163, 'Asian': 309, 'American ': 6181, 'Spanish': 637, 'Mexican': 754, '…
```