Inspections Restaurants Report - CCC2-02

Made by: Adrien DJEBAR, Emma FROMAGER, Abdelhak HACIB, Ridge LOWAO, Alex ROUSSEL

1. Introduction to the dataset

How to design schemas in NoSQL databases like Cassandra requires a different mindset compared to traditional SQL databases. In NoSQL, particularly in Cassandra, the schema design is heavily driven by the queries you need to perform. With that in mind, we have to understand our dataset thoroughly before we can design the schema, and also what kind of queries we have to perform.

Here's a peek of how our dataset is structured for a single row :


```
[
  {
    "idRestaurant": 40373938,
    "restaurant": {
      "name": "IHOP",
      "borough": "BRONX",
      "buildingnum": "5655",
      "street": "BROADWAY",
      "zipcode": "10463",
      "phone": "7185494565",
      "cuisineType": "American"
    },
    "inspectionDate": "2016-08-16",
    "violationCode": "04L",
    "violationDescription": "Evidence of mice or live mice present in facility's
food and/or non-food areas.",
    "criticalFlag": "Critical",
    "score": 15,
    "grade": ""
  }
]
```


Considering our dataset is of difficulty "2", we have to provide :

- 6 simple queries
- 2 complex queries
- 1 hard query

We have to design a schema that can answer these queries that we'll be running most frequently. Two approaches can be taken to design the schema:

- Multi-table design: Data is split into multiple tables, which can make simple queries more efficient. Two tables are created, one for *restaurants* which will hold all the relevant data about the restaurants, and one for *inspections* which will hold all the relevant data about the inspection of the said restaurants. Since in CQL there is no JOIN operation, if we would like to do queries that require data from both

tables, we would need materialized views or create a third table that would hold the data from both tables.

- Single table design: All data in one table, which can simplify complex and hard queries that need to access both *restaurant* and *inspection* data simultaneously. However, this approach can make simple queries more complex and less efficient. In this approach, the "restaurant" dictionary is denormalized so that we can access all the data in one table.

But what if we could have the best of both worlds ? We can use a multi-table design for simple queries, and a single table design for complex and hard queries. This way, we can have the best performance for all our queries. We can also use a materialized view to have the best of both worlds, but we'll have to see if it's necessary. NoSQL databases are more of a iterative process than SQL databases, so we'll have to see how our queries perform and adjust our schema accordingly.

2. Designing the schema

Let's take a closer look at how our dataset is structured. For each row, we can separate the data into two parts: the restaurant data and the inspection data. The inspection data hold the date of when the inspection happened, the violation code, a description, a critical flag, a score and a grade. While, the restaurant hold some information that aren't directly related to the inspection, like the name, the borough, the building number, the street, the zipcode, the phone number and the cuisine type. So it seems reasonable either way to split the data into two tables or to "flatten" the "restaurant" dictionary into it's elements.


```
[
  {
    "idRestaurant": 40373938,
    "restaurant": {
      "name": "IHOP",
      "borough": "BRONX",
      "buildingnum": "5655",
      "street": "BROADWAY",
      "zipcode": "10463",
      "phone": "7185494565",
      "cuisineType": "American"
    },
    "inspectionDate": "2016-08-16",
    "violationCode": "04L",
    "violationDescription": "Evidence of mice or live mice present in facility's
food and/or non-food areas.",
    "criticalFlag": "Critical",
    "score": 15,
    "grade": ""
  }
]
```


2.1 Mutli-table design

Here's how we can design the schema using a multi-table design:

```
CREATE TABLE IF NOT EXISTS restaurant (
    idRestaurant INT,
    name TEXT,
    borough TEXT,
    buildingnum TEXT,
    street TEXT,
    zipcode TEXT,
    phone TEXT,
    cuisineType TEXT,
    PRIMARY KEY (idRestaurant)
);

CREATE TABLE IF NOT EXISTS inspection (
    idRestaurant INT,
    inspectionDate DATE,
    violationCode TEXT,
    violationDescription TEXT,
    criticalFlag TEXT,
    score INT,
    grade TEXT,
    PRIMARY KEY (idRestaurant, inspectionDate)
);
```

Both tables have "idRestaurant" as the partition key, which is the unique identifier for each restaurant. This way, we keep a relationship between the two tables. The "inspection" table has "inspectionDate" as the clustering column, which will allow us to sort the data by date.

2.2 Single table design

Here's how we can design the schema using a single table design:

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

In this design, we have denormalized the data from both `restaurant` and `inspection` tables into a single table. Each row in this table would represent an inspection, including all relevant restaurant details. This approach allows us to run complex queries that need information from both entities without the need for joins.

3. Importing the data into Cassandra

3.1 Cleaning the JSON file

Before we can import the data into Cassandra, we have to clean the JSON file. The JSON file is not in a format that Cassandra can understand. However, our JSON objects are not in an array, and they are not separated by commas.

For this, we created a simple Python script to add the missing commas and to put the JSON objects into an array. Here's the script:

```python
# Fix the JSON data not being in an array format
with open('InspectionsRestaurant.json', 'r') as read_obj, open('InspectionsRestaurantFixed.json', 'w') as write_obj:
    write_obj.write('[')

    # Read the file line by line
    for line in read_obj:
        # Add a comma to the end of the line and write
        write_obj.write(line.rstrip('\n') + ',\n')

    # Remove the last comma
    write_obj.seek(write_obj.tell() - 3)

    # Write the end of the file
    write_obj.write(']')
```

If we compare the line count of the original file and the cleaned file with "wc -l", we have the same number of lines.

```
Exercices/_dataset/Cassandra via v3.11.5
17:42:24 ❯ wc -l InspectionsRestaurant.json                    13GiB/31GiB  38ms
442795 InspectionsRestaurant.json

Exercices/_dataset/Cassandra via v3.11.5
17:42:41 ❯ wc -l InspectionsRestaurantFixed.json               13GiB/31GiB  3s
442795 InspectionsRestaurantFixed.json
```

3.2 Converting the JSON file to CSV

To efficiently convert the JSON file to CSV, we can use the library `pandas` in Python to create dataframes that enable to work with the data in a more structured way. We can then save the dataframes to CSV files. Here's the script:

```python
import json
import os

import pandas as pd

# Create the data directory if it does not exist
os.makedirs("csv", exist_ok=True)
```

```
# Read the json file
with open("json/InspectionsRestaurantFixed.json") as f:
    data = json.load(f)

    restaurant_data = []
    inspection_data = []
    inspections_restaurants_data = []

    # Loop through the data once
    for item in data:
        # Create the restaurant data table
        restaurant = item["restaurant"]
        restaurant["idRestaurant"] = item["idRestaurant"]
        restaurant_data.append(restaurant)

        # Create the inspection data table
        inspection = {
            k: v if v != "" else "Not Yet Graded"
            for k, v in item.items()
            if k != "restaurant"
        }
        inspection_data.append(inspection)

        # Create the combined data table
        combined = {**restaurant, **inspection}
        inspections_restaurants_data.append(combined)

    # Create the dataframes
    restaurant_df = pd.DataFrame(restaurant_data)
    inspection_df = pd.DataFrame(inspection_data)
    inspections_restaurants_df = pd.DataFrame(inspections_restaurants_data)

    # Print the line count for each dataframe
    print(f"restaurant_df: {len(restaurant_df)} lines")
    print(f"inspection_df: {len(inspection_df)} lines")
    print(f"inspections_restaurants_df: {len(inspections_restaurants_df)} lines")

    # Save the dataframes to csv files
    restaurant_df.to_csv(
        "csv/restaurant.csv",
        sep=";",
        index=False,
        columns=[
            "idRestaurant",
            "name",
            "borough",
            "buildingnum",
            "street",
            "zipcode",
            "phone",
            "cuisineType",
        ],
    )
```

```
    inspection_df.to_csv(
        "csv/inspection.csv",
        sep=";",
        index=False,
        columns=[
            "idRestaurant",
            "inspectionDate",
            "violationCode",
            "violationDescription",
            "criticalFlag",
            "score",
            "grade",
        ],
    )
    inspections_restaurants_df.to_csv(
        "csv/inspections_restaurants.csv",
        sep=";",
        index=False,
        columns=[
            "idRestaurant",
            "inspectionDate",
            "name",
            "borough",
            "buildingnum",
            "street",
            "zipcode",
            "phone",
            "cuisineType",
            "violationCode",
            "violationDescription",
            "criticalFlag",
            "score",
            "grade",
        ],
    )
```

If we run the script, we can see that the line count for each dataframe is the same as the original JSON file.

```
🏠 Exercices\_dataset\Cassandra via  v3.11.5
🌱  20:37:24 ❯ python -u "d:\OneDrive - De Vinci\Cours\ESILV\
Année 4\Semestre 2\Core Computer Science\NoSQL\Exercices\_dat
aset\Cassandra\create_csv.py"
restaurant_df: 442795 lines
inspection_df: 442795 lines
restaurant_inspection_df: 442795 lines
```

We can then compress the CSV files into a tar.gz file to make it easier to copy to the Cassandra container.

```
tar -czvf inspections_restaurant.tar.gz *
```

3.3 Importing the CSV files into Cassandra

Now that we have the CSV files, we can import them into Cassandra. We can use a CQL script that creates the tables and imports the data into the tables. For it, we have to copy the CSV files and the CQL script to the Cassandra container. We can use the `docker cp` command to copy the files to the container.

```
docker cp /path/to/inspections_restaurant.tar.gz cassandra_nosql:/home
docker cp /path/to/create_import_table.cql cassandra_nosql:/home
```

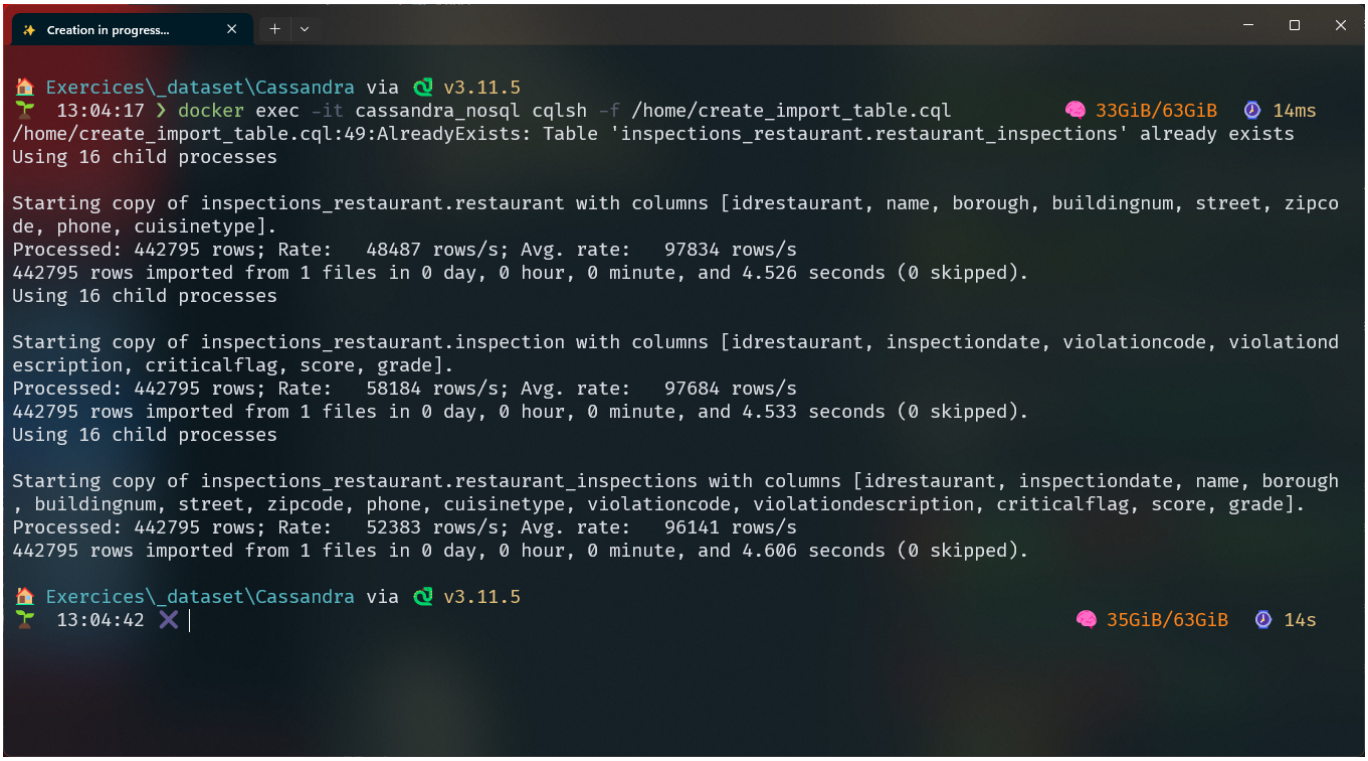
We can then extract the tar.gz file and import the data into Cassandra.

```
tar xzvf inspections_restaurant.tar.gz -C ./inspections_restaurant
```

We can then use the `cqlsh` command to run the CQL script.

```
docker exec -it cassandra_nosql cqlsh -f /home/create_import_table.cql
```

And that's it! We have imported the data into Cassandra.



The content of the `create_import_table.cql` file is as follows:

```
-- Description: This script creates the keyspace and the table for the restaurant inspections data.
CREATE KEYSPACE IF NOT EXISTS inspections_restaurant
WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
```

```
-- Create the table for the restaurant inspections data.
USE inspections_restaurant;

-- Create the table for the restaurant inspections data.
CREATE TABLE IF NOT EXISTS restaurant (
    idRestaurant INT,
    name TEXT,
    borough TEXT,
    buildingnum TEXT,
    street TEXT,
    zipcode TEXT,
    phone TEXT,
    cuisineType TEXT,
    PRIMARY KEY (idRestaurant)
);

CREATE TABLE IF NOT EXISTS inspection (
    idRestaurant INT,
    inspectionDate DATE,
    violationCode TEXT,
    violationDescription TEXT,
    criticalFlag TEXT,
    score INT,
    grade TEXT,
    PRIMARY KEY (idRestaurant, inspectionDate)
);

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

-- Import the data into the table.
COPY restaurant (idRestaurant, name, borough, buildingnum, street, zipcode, phone,
cuisineType)
FROM '/home/inspections_restaurants/restaurant.csv' WITH HEADER=TRUE AND
DELIMITER=';';

COPY inspection (idRestaurant, inspectionDate, violationCode,
violationDescription, criticalFlag, score, grade)
```

```
FROM '/home/inspections_restaurants/inspection.csv' WITH HEADER=TRUE AND
DELIMITER=';';

COPY inspections_restaurants (idRestaurant, inspectionDate, name, borough,
buildingnum, street, zipcode, phone, cuisineType, violationCode,
violationDescription, criticalFlag, score, grade)
FROM '/home/inspections_restaurants/inspections_restaurants.csv' WITH HEADER=TRUE
AND DELIMITER=';';
```

3.4 Automating the process

We can automate the process of importing the data into Cassandra by creating a shell script that does all the steps for us. First clone the GitHub repository and execute the `execute_pipeline.sh` script:

```
#!/bin/bash

# Prompt the user for the Docker image name
echo "Please enter the Docker image name for Cassandra:"
read docker_name

# Check if the Docker image name is empty
if [ -z "$docker_name" ]; then
    echo "No Docker image name provided. Exiting."
    exit 1
fi

echo "Executing pipeline for Cassandra..."

echo -e "[INFO] Correcting array comma in json file..."
python fix_array_comma.py
echo -e "[INFO] Done! ✔"

echo -e "\n[INFO] Creating csv files from json for Cassandra readable files..."
python create_csv.py
echo -e "[INFO] Done! ✔"

echo -e "\n[INFO] Compressing csv files to tar.gz for transfer in the docker..."
(cd csv && tar -czvf ../inspections_restaurants.tar.gz *)

echo -e "\n[INFO] Copying tar.gz file and script to docker..."
docker cp inspections_restaurants.tar.gz $docker_name:/home
docker cp create_import_table.cql $docker_name:/home

echo -e "\n[INFO] Extracting tar.gz file in docker..."
docker exec -it $docker_name mkdir /home/inspections_restaurants
docker exec -it $docker_name tar -xzvf /home/inspections_restaurants.tar.gz -C
/home/inspections_restaurants

echo -e "\n[INFO] Creating Cassandra tables and import data..."
docker exec -it $docker_name cqlsh -f /home/create_import_table.cql
echo -e "[INFO] Done! ✔"
```

```
echo -e "\n[INFO] Executing pipeline... Done! ✔"
```

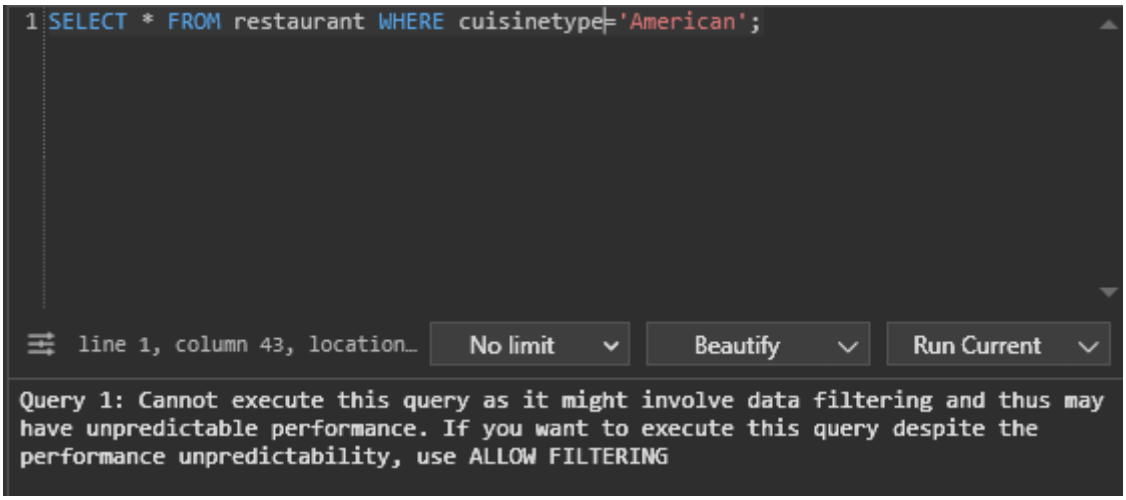
4. Running the queries

Now that we have imported the data into Cassandra, we can run the queries.

4.1 Simple queries

Let's do a simple query. We want to list all the restaurant which serves American cuisine. However, something interesting is going to happen.

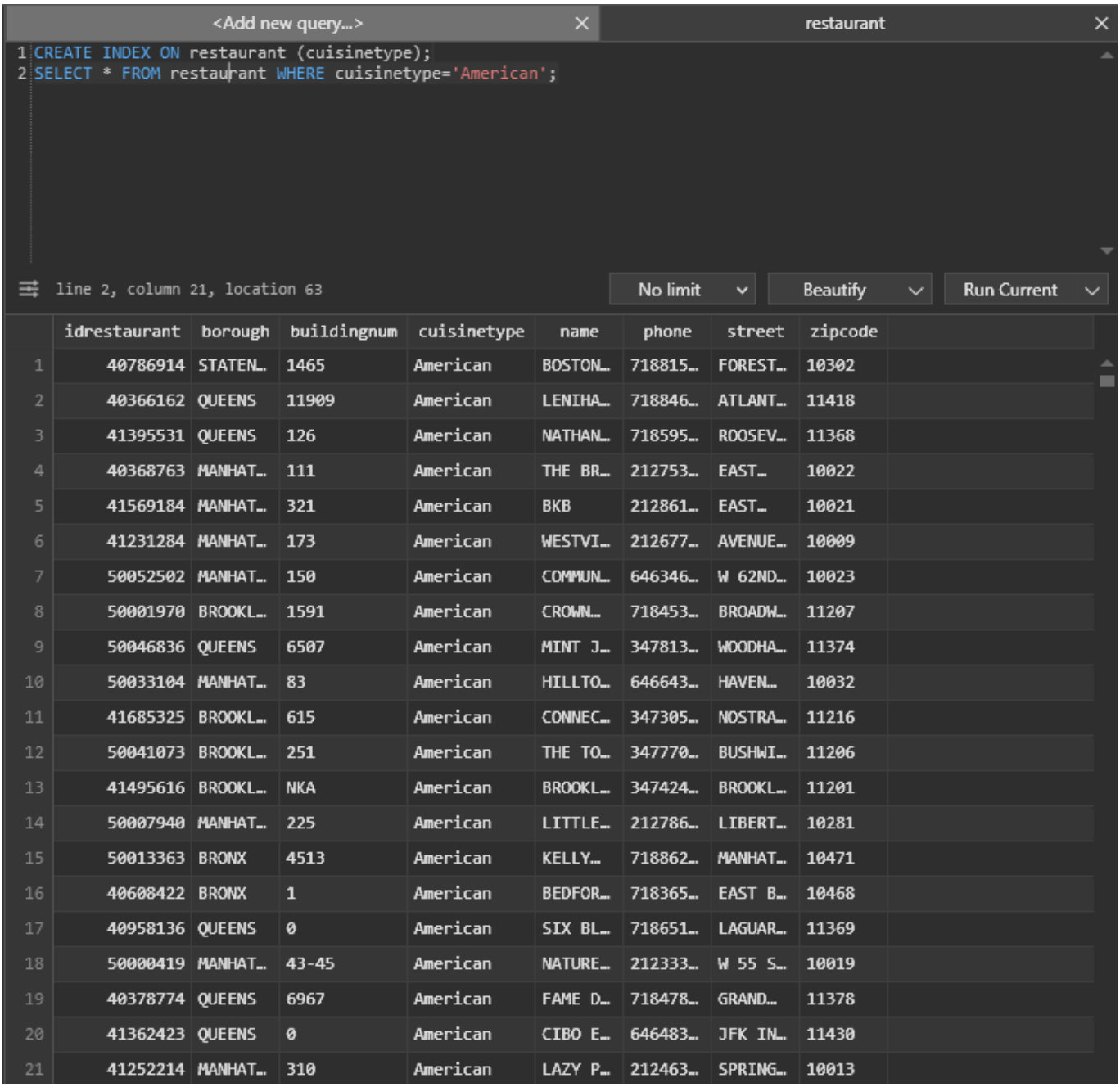
```
SELECT * FROM restaurant WHERE name='American';
```



Without `ALLOW FILTERING`, we can't run the query. This is because we are trying to filter on a non-primary key column. We can fix this by creating an index on the `cuisineType` column. But why is this happening? This is because `ALLOW FILTERING` does a `BROADCAST` operation to all the nodes in the cluster, to query all the data in the table. To prevent this, we can create an index on the column we want to filter on, allowing to put all the relevant data on the same node. This way, the query with an exact match (i.e where `cuisineType='American'`) will be faster ! 😊

Now that we have created the index, we can run the query again.

```
CREATE INDEX ON restaurant (cuisinetype);
SELECT * FROM restaurant WHERE cuisinetype='American';
```

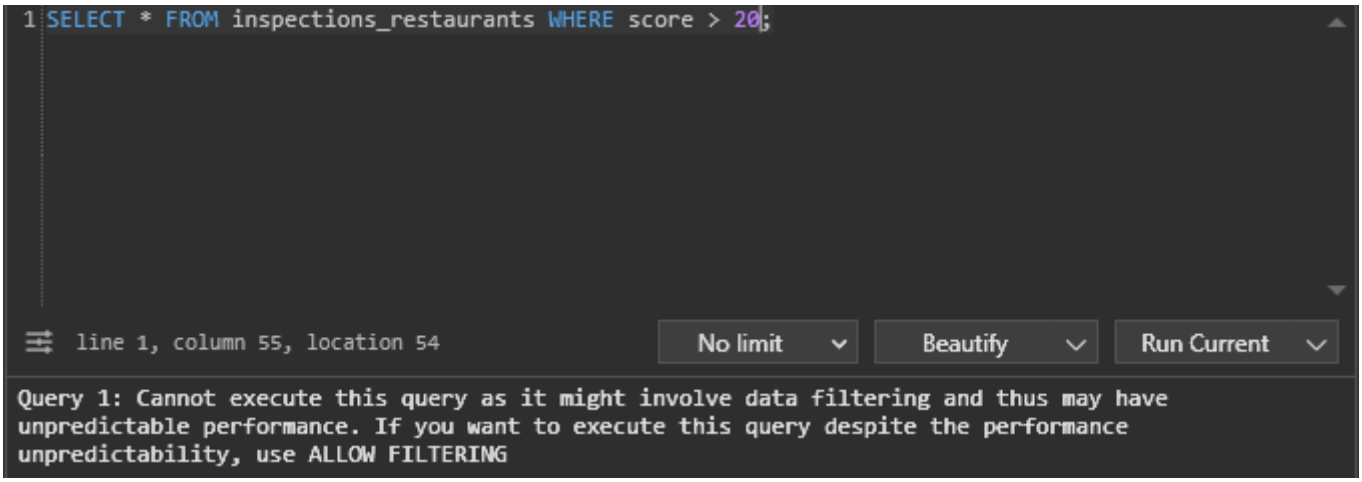


	idrestaurant	borough	buildingnum	cuisinetype	name	phone	street	zipcode
1	40786914	STATEN…	1465	American	BOSTON…	718815…	FOREST…	10302
2	40366162	QUEENS	11909	American	LENIHA…	718846…	ATLANT…	11418
3	41395531	QUEENS	126	American	NATHAN…	718595…	ROOSEV…	11368
4	40368763	MANHAT…	111	American	THE BR…	212753…	EAST…	10022
5	41569184	MANHAT…	321	American	BKB	212861…	EAST…	10021
6	41231284	MANHAT…	173	American	WESTVI…	212677…	AVENUE…	10009
7	50052502	MANHAT…	150	American	COMMUN…	646346…	W 62ND…	10023
8	50001970	BROOKL…	1591	American	CROWN…	718453…	BROADW…	11207
9	50046836	QUEENS	6507	American	MINT J…	347813…	WOODHA…	11374
10	50033104	MANHAT…	83	American	HILLTO…	646643…	HAVEN…	10032
11	41685325	BROOKL…	615	American	CONNEC…	347305…	NOSTRA…	11216
12	50041073	BROOKL…	251	American	THE TO…	347770…	BUSHWI…	11206
13	41495616	BROOKL…	NKA	American	BROOKL…	347424…	BROOKL…	11201
14	50007940	MANHAT…	225	American	LITTLE…	212786…	LIBERT…	10281
15	50013363	BRONX	4513	American	KELLY…	718862…	MANHAT…	10471
16	40608422	BRONX	1	American	BEDFOR…	718365…	EAST B…	10468
17	40958136	QUEENS	0	American	SIX BL…	718651…	LAGUAR…	11369
18	50000419	MANHAT…	43-45	American	NATURE…	212333…	W 55 S…	10019
19	40378774	QUEENS	6967	American	FAME D…	718478…	GRAND…	11378
20	41362423	QUEENS	0	American	CIBO E…	646483…	JFK IN…	11430
21	41252214	MANHAT…	310	American	LAZY P…	212463…	SPRING…	10013

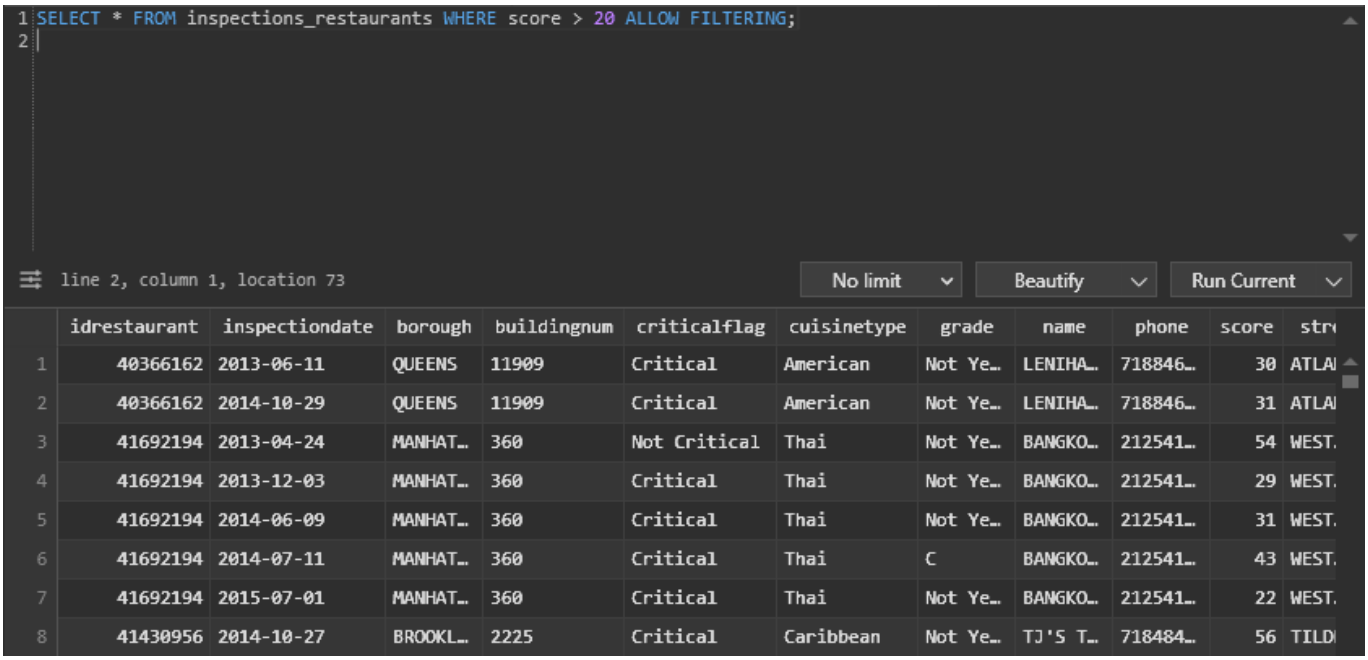
One thing to consider is that creating an index can be expensive in terms of storage and performance. It's a trade-off between storage and performance. We have to consider the size of the data and the number of queries we are going to run. If we are going to run a lot of queries that filter on the `cuisineType` column, then it might be worth it to create an index or a table/materialized view that has the `cuisineType` as the partition key.

- Now, what if we want to see all the restaurants with a score greater than 20? We could create an index on the `score` column, just like before ..? But we can't. Why ? Actually, non-matching query like ">=", "<=", ">", "<" still requires to use `ALLOW FILTERING` because Cassandra still have to filter and then `BROADCAST`.

```
SELECT * FROM inspections_restaurants WHERE score > 20;
```




```
SELECT * FROM inspections_restaurants WHERE score > 20 ALLOW FILTERING;
```



	idrestaurant	inspectiondate	borough	buildingnum	criticalflag	cuisinetype	grade	name	phone	score	str
1	40366162	2013-06-11	QUEENS	11909	Critical	American	Not Ye…	LENIHA…	718846…	30	ATLA
2	40366162	2014-10-29	QUEENS	11909	Critical	American	Not Ye…	LENIHA…	718846…	31	ATLA
3	41692194	2013-04-24	MANHAT…	360	Not Critical	Thai	Not Ye…	BANGKO…	212541…	54	WEST.
4	41692194	2013-12-03	MANHAT…	360	Critical	Thai	Not Ye…	BANGKO…	212541…	29	WEST.
5	41692194	2014-06-09	MANHAT…	360	Critical	Thai	Not Ye…	BANGKO…	212541…	31	WEST.
6	41692194	2014-07-11	MANHAT…	360	Critical	Thai	C	BANGKO…	212541…	43	WEST.
7	41692194	2015-07-01	MANHAT…	360	Critical	Thai	Not Ye…	BANGKO…	212541…	22	WEST.
8	41430956	2014-10-27	BROOKL…	2225	Critical	Caribbean	Not Ye…	TJ'S T…	718484…	56	TILD

- The grades of indian restaurants.

```
CREATE INDEX ON inspections_restaurants (cuisineType);
SELECT idrestaurant, name, cuisinetype, grade, score FROM inspections_restaurants
WHERE cuisinetype = 'Indian';
```

Query 1 | Query 2

	idrestaurant	name	cuisinetype	grade	score
1	50044741	INDIAN EXPRESS	Indian	Not Yet Graded	0
2	50044741	INDIAN EXPRESS	Indian	A	12
3	50049046	NAAN & GRILL	Indian	Not Yet Graded	38
4	50049046	NAAN & GRILL	Indian	A	12
5	50017852	CHOTE NAWAB	Indian	A	9
6	50017852	CHOTE NAWAB	Indian	Not Yet Graded	21
7	50017852	CHOTE NAWAB	Indian	A	5
8	41487281	GOLDEN TERRACE BANQUET HALL	Indian	A	7
9	41487281	GOLDEN TERRACE BANQUET HALL	Indian	A	5
10	41487281	GOLDEN TERRACE BANQUET HALL	Indian	Not Yet Graded	24
11	41487281	GOLDEN TERRACE BANQUET HALL	Indian	A	12
12	41487281	GOLDEN TERRACE BANQUET HALL	Indian	Not Yet Graded	19
13	41487281	GOLDEN TERRACE BANQUET HALL	Indian	B	15
14	41487281	GOLDEN TERRACE BANQUET HALL	Indian	Not Yet Graded	18
15	41487281	GOLDEN TERRACE BANQUET HALL	Indian	A	9
16	50050501	BAWARCHI INDIAN CUISINE	Indian	A	13
17	40394066	BOMBAY MASALA	Indian	Not Yet Graded	24
18	40394066	BOMBAY MASALA	Indian	A	13
19	40394066	BOMBAY MASALA	Indian	Not Yet Graded	0
20	40394066	BOMBAY MASALA	Indian	A	5
21	40394066	BOMBAY MASALA	Indian	Not Yet Graded	19
22	40394066	BOMBAY MASALA	Indian	A	10
23	40394066	BOMBAY MASALA	Indian	A	9

Data | Message | Chart 19 ms 2,195 rows Export...

- Show the score and grade with the violation description for each restaurant. It might be interesting to see if there is a correlation between the grade and the score.

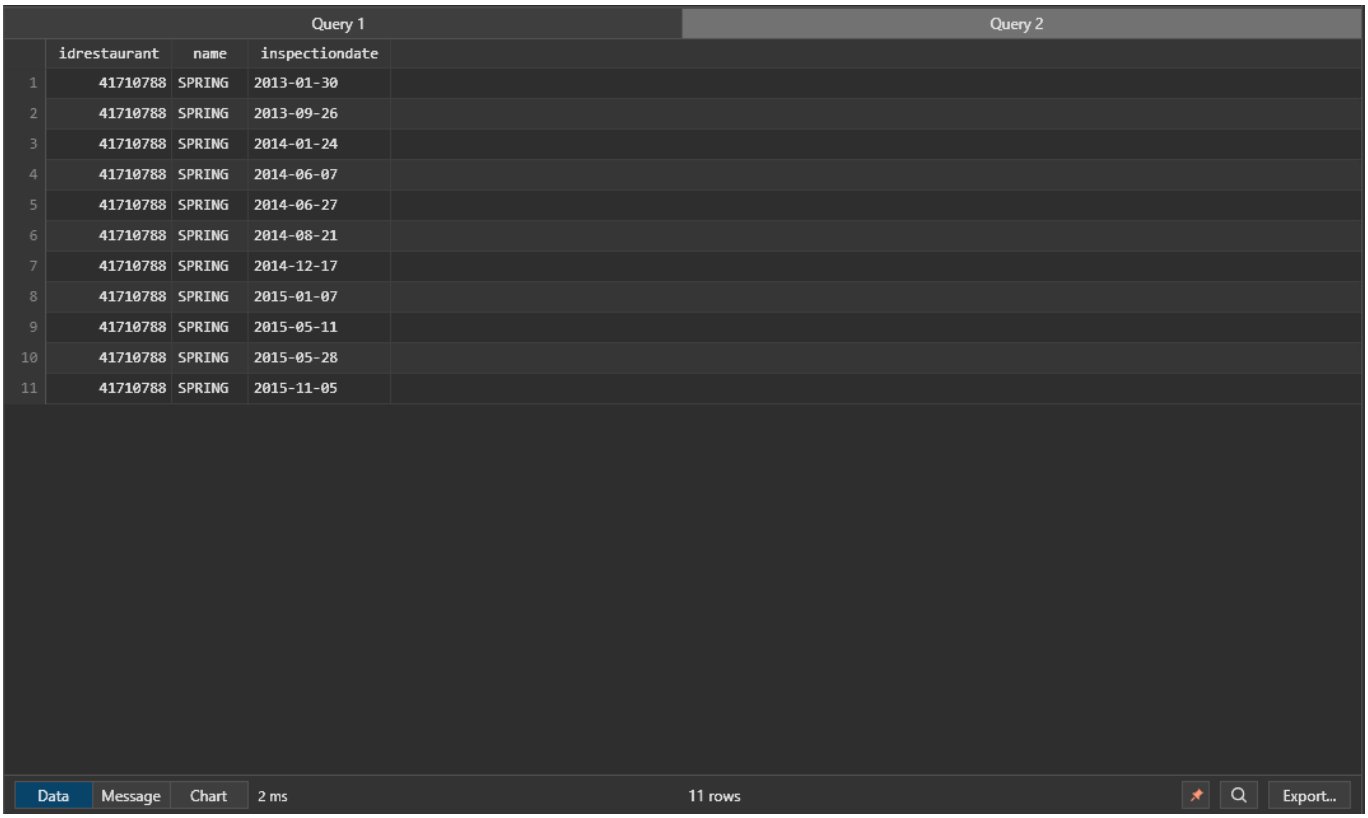
```
SELECT  name, violationdescription, score, grade FROM inspections_restaurants;
```

	name	violationdescription	score	grade
1	BOSTON MARKET	Wiping cloths soiled or not stored in sanitiz…	12	A
2	BOSTON MARKET	Food contact surface not properly washed, rin…	12	A
3	BOSTON MARKET	Sanitized equipment or utensil, including in-…	7	A
4	LENIHAN'S SALOON	Evidence of mice or live mice present in faci…	30	Not Yet Graded
5	LENIHAN'S SALOON	Lighting inadequate; permanent lighting not p…	16	B
6	LENIHAN'S SALOON	Plumbing not properly installed or maintained…	15	Not Yet Graded
7	LENIHAN'S SALOON	Filth flies or food/refuse/sewage-associated…	13	A
8	LENIHAN'S SALOON	Evidence of mice or live mice present in faci…	31	Not Yet Graded
9	LENIHAN'S SALOON	Evidence of mice or live mice present in faci…	9	A
10	LENIHAN'S SALOON	Non-food contact surface improperly construct…	13	A
11	LENIHAN'S SALOON	Non-food contact surface improperly construct…	2	A
12	BANGKOK HOUSE	Mechanical or natural ventilation system not…	54	Not Yet Graded
13	BANGKOK HOUSE	Non-food contact surface improperly construct…	11	P
14	BANGKOK HOUSE	Cold food item held above 41Â° F (smoked fish…	15	B
15	BANGKOK HOUSE	Food contact surface not properly washed, rin…	29	Not Yet Graded
16	BANGKOK HOUSE	Food contact surface not properly washed, rin…	9	A
17	BANGKOK HOUSE	Cold food item held above 41Â° F (smoked fish…	31	Not Yet Graded
18	BANGKOK HOUSE	Hot food item not held at or above 140Â° F.	43	C
19	BANGKOK HOUSE	Raw, cooked or prepared food is adulterated,…	15	Not Yet Graded
20	BANGKOK HOUSE	Live roaches present in facility's food and/o…	6	A
21	BANGKOK HOUSE	Cold food item held above 41Â° F (smoked fish…	22	Not Yet Graded
22	BANGKOK HOUSE	Evidence of rats or live rats present in faci…	11	A
23	BANGKOK HOUSE	Food contact surface not properly washed, rin…	13	A
24	TJ'S TASTY CORNER	Food Protection Certificate not held by super…	13	A
25	TJ'S TASTY CORNER	Evidence of mice or live mice present in faci…	18	Not Yet Graded
26	TJ'S TASTY CORNER	Hot food item not held at or above 140Â° F.	17	B
27	TJ'S TASTY CORNER	Evidence of mice or live mice present in faci…	56	Not Yet Graded
28	TJ'S TASTY CORNER	Facility not vermin proof. Harborage or condi…	18	B
29	TJ'S TASTY CORNER	Hot food item not held at or above 140Â° F.	17	Not Yet Graded

Data | Message | Chart 2.963 s 150,882 rows

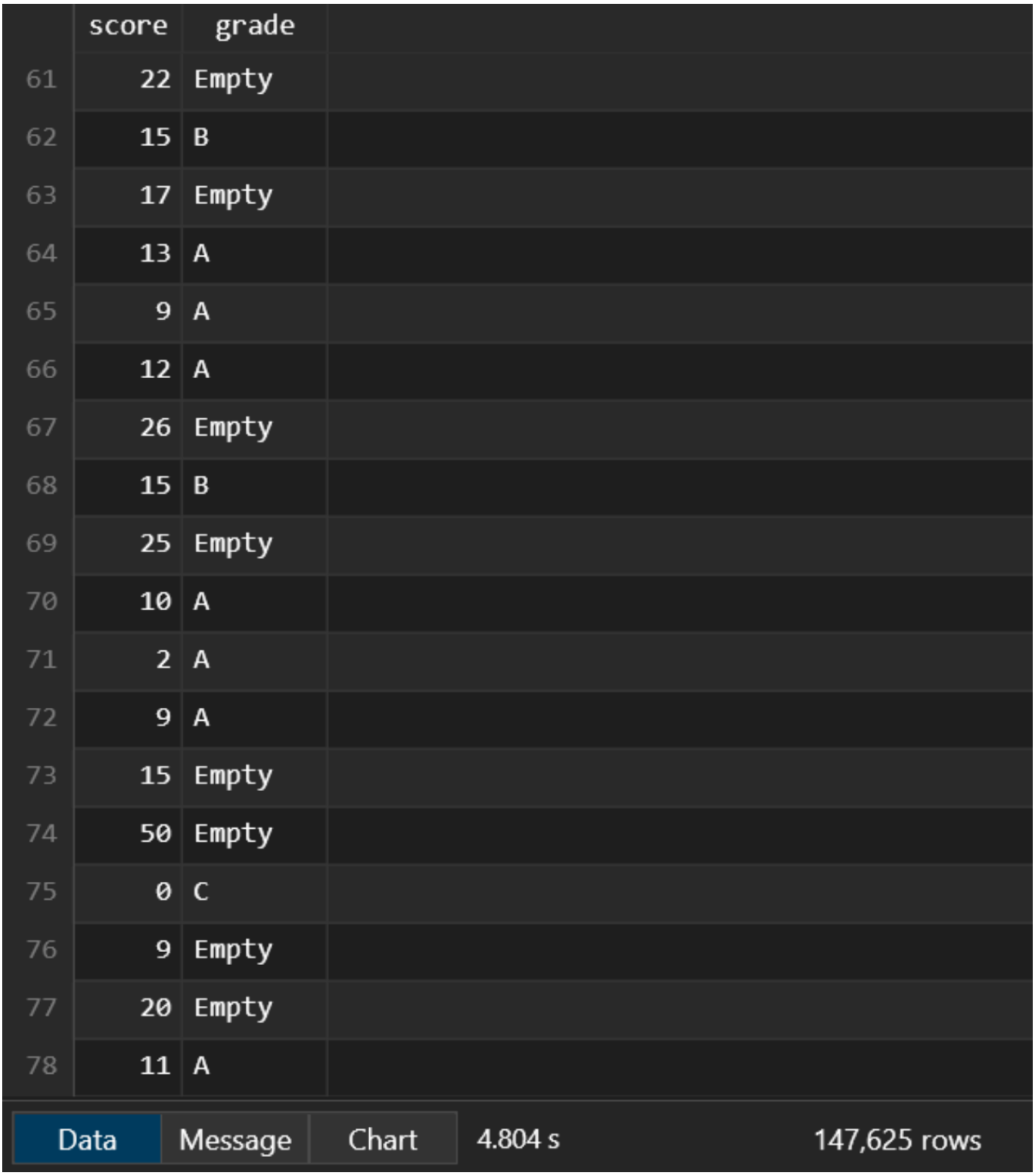
- Display the inspection dates for a specific restaurant named `SPRING`.

```
CREATE INDEX ON inspections_restaurants (name);
SELECT idRestaurant, name, inspectionDate FROM inspections_restaurants WHERE name
= 'SPRING';
```



- Grades and scores: We want to see the relationship between grades and scores.

```
SELECT score, grade
FROM inspections_restaurants;
```



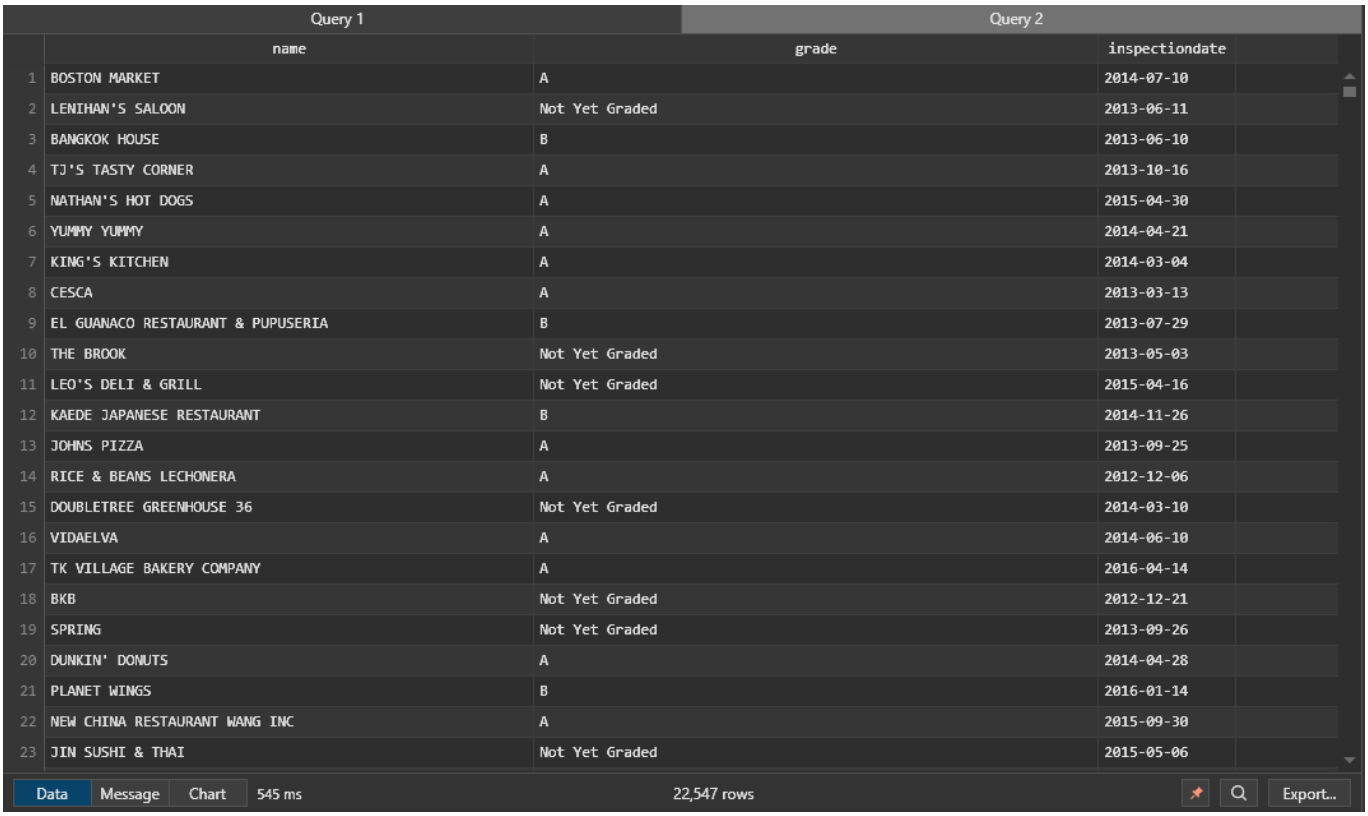
	score	grade
61	22	Empty
62	15	B
63	17	Empty
64	13	A
65	9	A
66	12	A
67	26	Empty
68	15	B
69	25	Empty
70	10	A
71	2	A
72	9	A
73	15	Empty
74	50	Empty
75	0	C
76	9	Empty
77	20	Empty
78	11	A

We can see that a grade of A generally correlates with a low score, B is in between and C generally correlates with a high score (bad).

4.2 Complex queries

- Display the name and the grade of restaurants that have a `criticalFlag` set to `Critical`.

```
CREATE INDEX ON inspections_restaurants (criticalflag);
SELECT name, grade, inspectiondate FROM inspections_restaurants  WHERE
criticalflag = 'Critical' GROUP BY idRestaurant;
```



Query 1 | Query 2

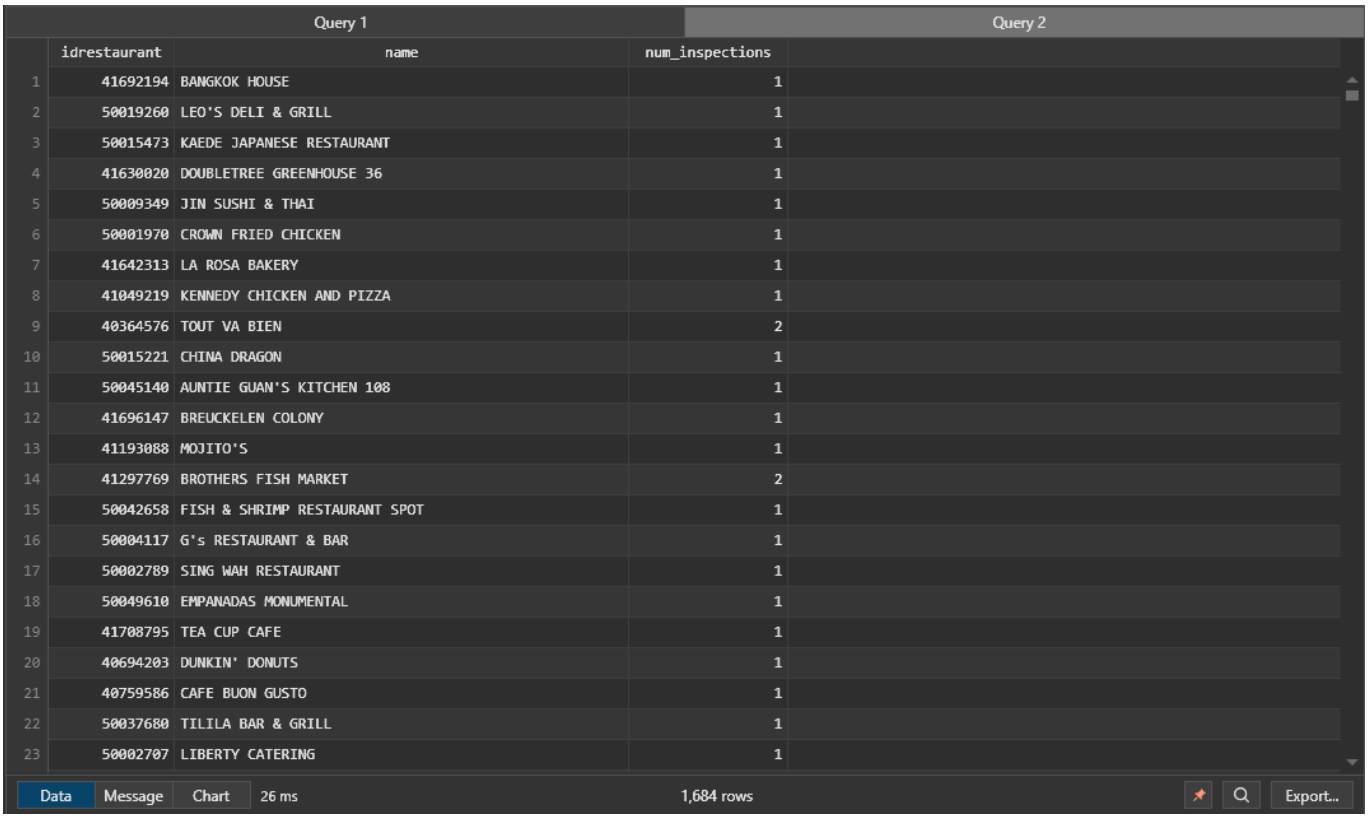
	name	grade	inspectiondate
1	BOSTON MARKET	A	2014-07-10
2	LENIHAN'S SALOON	Not Yet Graded	2013-06-11
3	BANGKOK HOUSE	B	2013-06-10
4	TJ'S TASTY CORNER	A	2013-10-16
5	NATHAN'S HOT DOGS	A	2015-04-30
6	YUMMY YUMMY	A	2014-04-21
7	KING'S KITCHEN	A	2014-03-04
8	CESCA	A	2013-03-13
9	EL GUANACO RESTAURANT & PUPUSERIA	B	2013-07-29
10	THE BROOK	Not Yet Graded	2013-05-03
11	LEO'S DELI & GRILL	Not Yet Graded	2015-04-16
12	KAEDE JAPANESE RESTAURANT	B	2014-11-26
13	JOHNS PIZZA	A	2013-09-25
14	RICE & BEANS LECHONERA	A	2012-12-06
15	DOUBLETREE GREENHOUSE 36	Not Yet Graded	2014-03-10
16	VIDAELVA	A	2014-06-10
17	TK VILLAGE BAKERY COMPANY	A	2016-04-14
18	BKB	Not Yet Graded	2012-12-21
19	SPRING	Not Yet Graded	2013-09-26
20	DUNKIN' DONUTS	A	2014-04-28
21	PLANET WINGS	B	2016-01-14
22	NEW CHINA RESTAURANT WANG INC	A	2015-09-30
23	JIN SUSHI & THAI	Not Yet Graded	2015-05-06

Data | Message | Chart | 545 ms | 22,547 rows | Export...

Funnily enough, we can see restaurants with a `criticalFlag` set to `Critical` have a grade of `A` or `B`. This is interesting because we would expect a `C` grade for restaurants with a `criticalFlag` set to `Critical`. There's also a lot of restaurants that simply don't have a grade, so we can't really know if they are good or bad.

- Number of inspections for restaurants that have a 'C' grade

```
CREATE INDEX ON inspections_restaurants (grade);
SELECT idRestaurant, name, COUNT(inspectionDate) AS num_inspections FROM
inspections_restaurants WHERE grade = 'C' GROUP BY idRestaurant;
```



Query 1 | Query 2

	idrestaurant	name	num_inspections
1	41692194	BANGKOK HOUSE	1
2	50019260	LEO'S DELI & GRILL	1
3	50015473	KAEDE JAPANESE RESTAURANT	1
4	41630020	DOUBLETREE GREENHOUSE 36	1
5	50009349	JIN SUSHI & THAI	1
6	50001970	CROWN FRIED CHICKEN	1
7	41642313	LA ROSA BAKERY	1
8	41049219	KENNEDY CHICKEN AND PIZZA	1
9	40364576	TOUT VA BIEN	2
10	50015221	CHINA DRAGON	1
11	50045140	AUNTIE GUAN'S KITCHEN 108	1
12	41696147	BREUCKELEN COLONY	1
13	41193088	MOJITO'S	1
14	41297769	BROTHERS FISH MARKET	2
15	50042658	FISH & SHRIMP RESTAURANT SPOT	1
16	50004117	G's RESTAURANT & BAR	1
17	50002789	SING WAH RESTAURANT	1
18	50049610	EMPANADAS MONUMENTAL	1
19	41708795	TEA CUP CAFE	1
20	40694203	DUNKIN' DONUTS	1
21	40759586	CAFE BUON GUSTO	1
22	50037680	TILILA BAR & GRILL	1
23	50002707	LIBERTY CATERING	1

Data | Message | Chart | 26 ms | 1,684 rows | Export...

We can see something interesting. Indeed, the number of inspections for "bad restaurants" (because they have a C mark) is generally equal to 1. This is interesting because we would expect a "bad restaurant" to have more inspections than a "good restaurant". This is a good example of how the data can be misleading.

4.3 Hard query

- Creation of the state function

```
CREATE OR REPLACE FUNCTION avgState(state tuple<int, int>, val int)
CALLED ON NULL INPUT RETURNS tuple<int, int> LANGUAGE java AS '
if (val != null) {
    state.setInt(0, state.getInt(0) + val);
    state.setInt(1, state.getInt(1) + 1);
}
return state;';
```

- Creation of the final function to calculate the average

```
CREATE OR REPLACE FUNCTION avgFinal(state tuple<int, int>)
CALLED ON NULL INPUT RETURNS double
LANGUAGE java AS '
int sum = state.getInt(0);
int count = state.getInt(1);
return (double) sum / count;
';
```

- Creation of the UDA function

```
CREATE OR REPLACE AGGREGATE average ( int )
SFUNC avgState STYPE tuple<int,int>
FINALFUNC avgFinal INITCOND (0,0);
```

- We display the average score of each restaurant using our newly created UDA function "average"

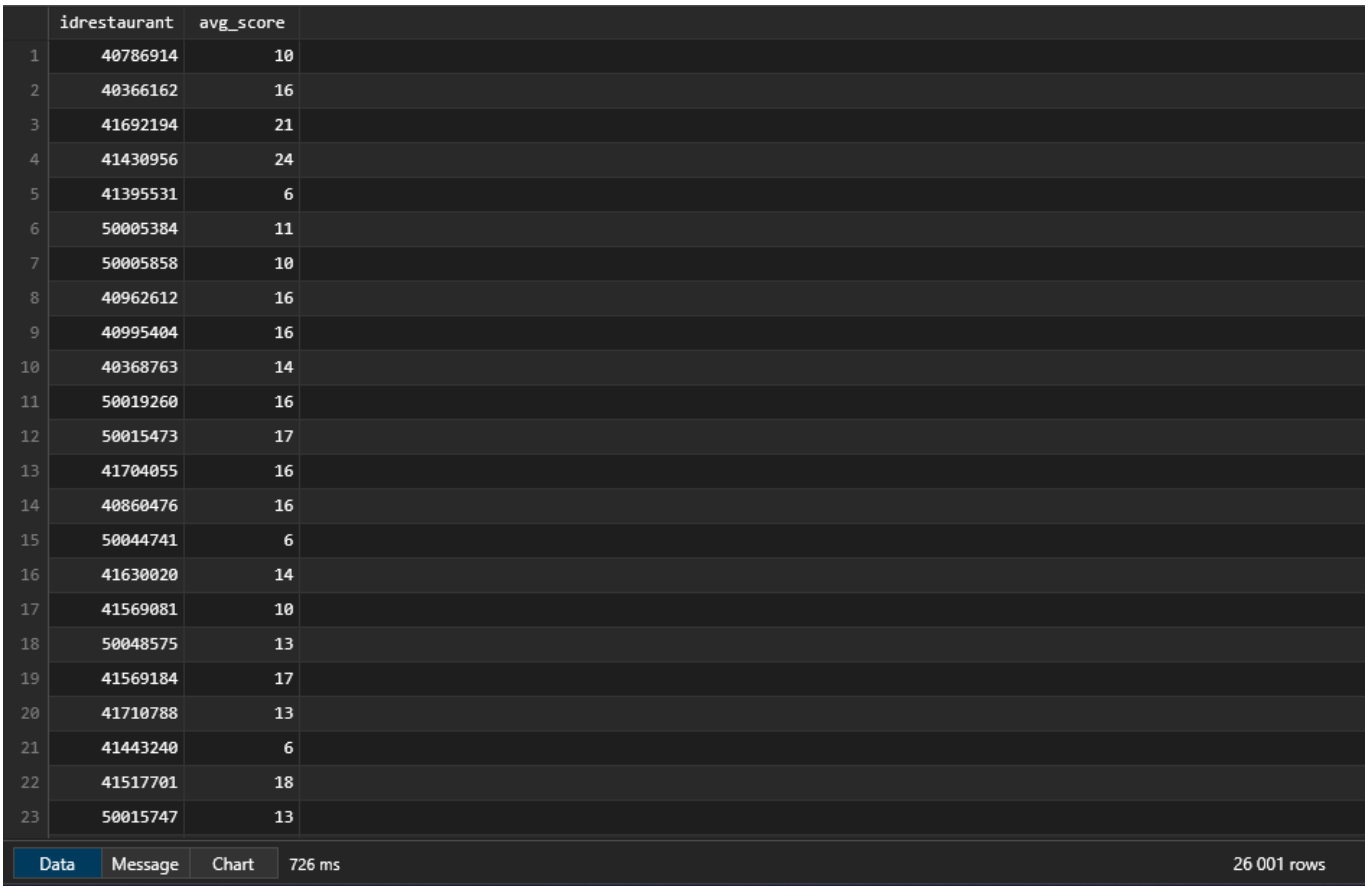
```
SELECT idrestaurant, average(score) AS total_score
FROM restaurant_inspections
GROUP BY idrestaurant;
```

	idrestaurant	avg_score
1	40786914	10,3333333333333
2	40366162	16,125
3	41692194	21,5833333333333
4	41430956	24,5833333333333
5	41395531	6
6	50005384	11
7	50005858	10
8	40962612	16
9	40995404	16,625
10	40368763	14,5
11	50019260	16
12	50015473	17,6666666666667
13	41704055	16,875
14	40860476	16,7142857142857
15	50044741	6
16	41630020	14,25
17	41569081	10,75
18	50048575	13
19	41569184	17
20	41710788	13,4545454545455
21	41443240	6,5
22	41517701	18
23	50015747	13,3333333333333

Data Message Chart 9,393 s 26 001 rows

- We do the same but this time using the included aggregate function "AVG"

```
SELECT idrestaurant, AVG(score) AS avg_score
FROM restaurant_inspections
GROUP BY idrestaurant;
```



	idrestaurant	avg_score
1	40786914	10
2	40366162	16
3	41692194	21
4	41430956	24
5	41395531	6
6	50005384	11
7	50005858	10
8	40962612	16
9	40995404	16
10	40368763	14
11	50019260	16
12	50015473	17
13	41704055	16
14	40860476	16
15	50044741	6
16	41630020	14
17	41569081	10
18	50048575	13
19	41569184	17
20	41710788	13
21	41443240	6
22	41517701	18
23	50015747	13

Data Message Chart 726 ms 26 001 rows

At first, we can see that using our UDA function "average", we achieve better results leading us to believe that UDA functions would be the go too if we need precision. But that precision comes with a price. If we compare the time it takes for each query to run and return results, a huge gap in compute time is visible between our UDA function and the onboard aggregate function.

5. Conclusion

In conclusion, we have successfully imported the data into Cassandra and ran the queries. We have designed the schema using a multi-table design and a single table design and automated the process of importing the data into Cassandra. We have run simple, complex and hard queries to pratice and better understand CQL as a whole.

Thanks for reading! 🚀