TP5 - NOSQL NEO4J RESTAURANT INSPECTIONS

Manon GARDIN

Matias OTTENSEN

Tiphaine KACHKACHI

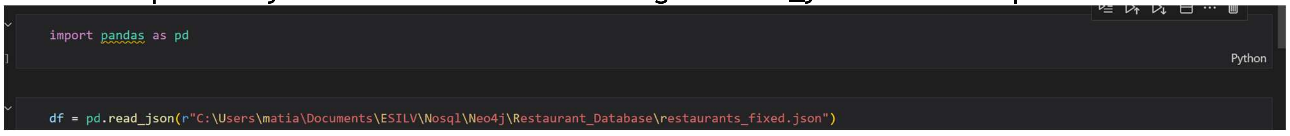
Alexandre GARNIER

Create the Database

Converting JSON to CSV

The first step is to convert our Json file to a csv one. The pyhton library pandas can be used to this end.

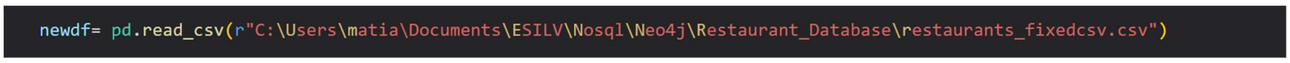
First we import the json file as as dataframe using the read_json method of pandas.



```
import pandas as pd
```

```
df = pd.read_json(r"C:\Users\matia\Documents\ESILV\Nosql\Neo4j\Restaurant_Database\restaurants_fixed.json")
```

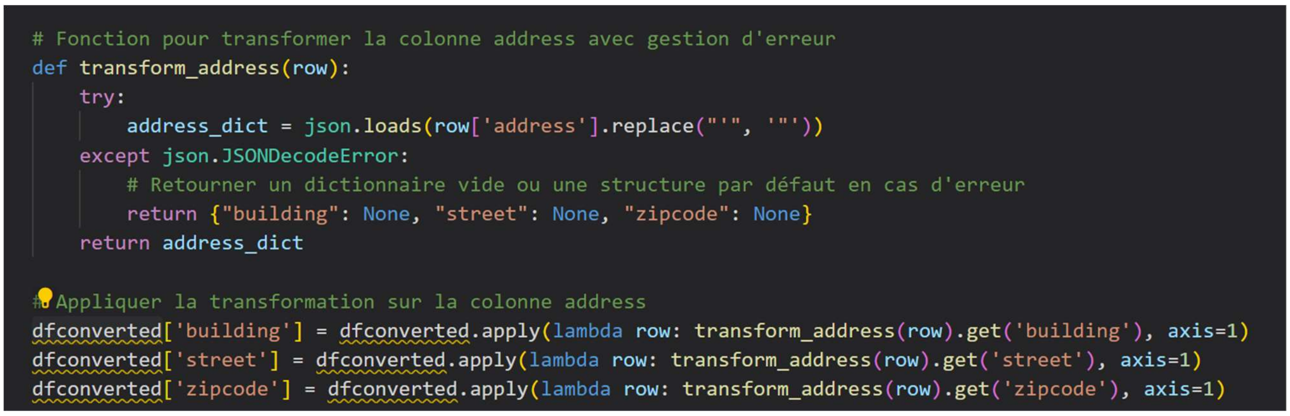
Then we will use the to_csv method to extract the data frame under the csv file



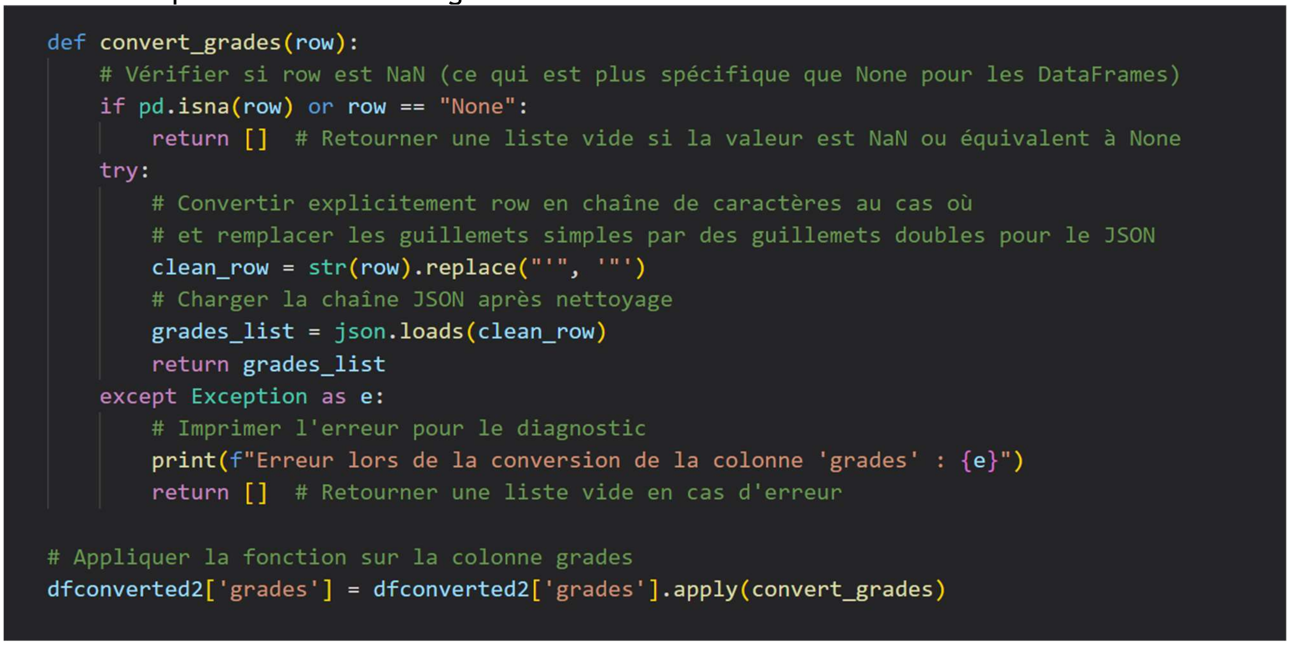
```
newdf= pd.read_csv(r"C:\Users\matia\Documents\ESILV\Nosql\Neo4j\Restaurant_Database\restaurants_fixedcsv.csv")
```

Treating the data

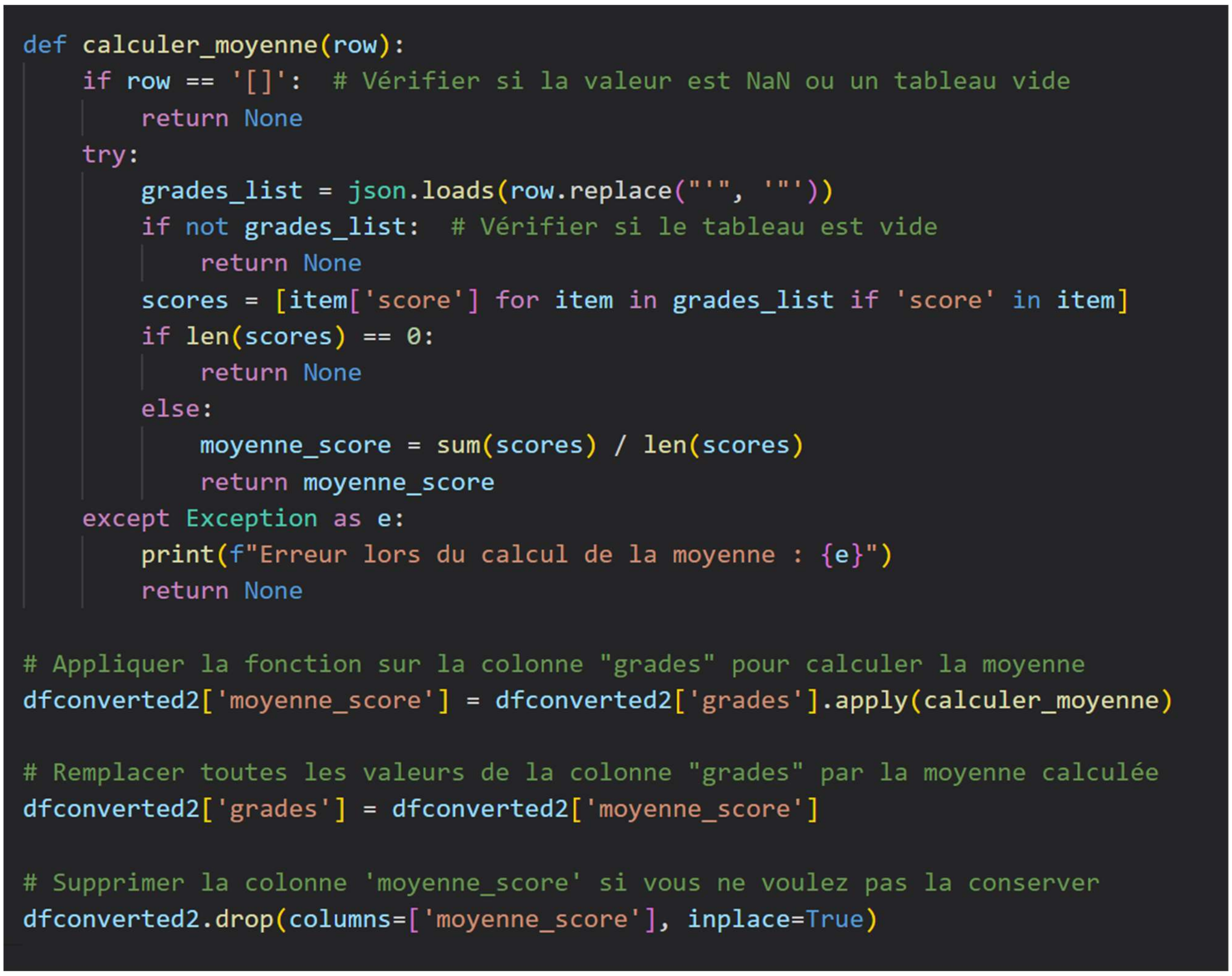
Before we export our csv file, we have some step to follow to allow NEO4J to process it.
First we split the address object into a building, a street and a zipcode, which will all be strings.


```python
# Fonction pour transformer la colonne address avec gestion d'erreur
def transform_address(row):
    try:
        address_dict = json.loads(row['address'].replace("'", '"'))
    except json.JSONDecodeError:
        # Retourner un dictionnaire vide ou une structure par défaut en cas d'erreur
        return {"building": None, "street": None, "zipcode": None}
    return address_dict

# Appliquer la transformation sur la colonne address
dfconverted['building'] = dfconverted.apply(lambda row: transform_address(row).get('building'), axis=1)
dfconverted['street'] = dfconverted.apply(lambda row: transform_address(row).get('street'), axis=1)
dfconverted['zipcode'] = dfconverted.apply(lambda row: transform_address(row).get('zipcode'), axis=1)
```

Then we adapt the format of the grades field to allow NEO4J to read it.


```python
def convert_grades(row):
    # Vérifier si row est NaN (ce qui est plus spécifique que None pour les DataFrames)
    if pd.isna(row) or row == "None":
        return []  # Retourner une liste vide si la valeur est NaN ou équivalent à None
    try:
        # Convertir explicitement row en chaîne de caractères au cas où
        # et remplacer les guillemets simples par des guillemets doubles pour le JSON
        clean_row = str(row).replace("'", '"')
        # Charger la chaîne JSON après nettoyage
        grades_list = json.loads(clean_row)
        return grades_list
    except Exception as e:
        # Imprimer l'erreur pour le diagnostic
        print(f"Erreur lors de la conversion de la colonne 'grades' : {e}")
        return []  # Retourner une liste vide en cas d'erreur

# Appliquer la fonction sur la colonne grades
dfconverted2['grades'] = dfconverted2['grades'].apply(convert_grades)
```

The next step is to replace the grades by their average to get a single float for NEO4J to process.

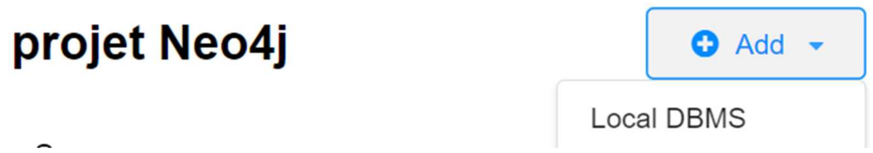


```python
def calculer_moyenne(row):
    if row == '[]':  # Vérifier si la valeur est NaN ou un tableau vide
        return None
    try:
        grades_list = json.loads(row.replace("'", '"'))
        if not grades_list:  # Vérifier si le tableau est vide
            return None
        scores = [item['score'] for item in grades_list if 'score' in item]
        if len(scores) == 0:
            return None
        else:
            moyenne_score = sum(scores) / len(scores)
            return moyenne_score
    except Exception as e:
        print(f"Erreur lors du calcul de la moyenne : {e}")
        return None

# Appliquer la fonction sur la colonne "grades" pour calculer la moyenne
dfconverted2['moyenne_score'] = dfconverted2['grades'].apply(calculer_moyenne)

# Remplacer toutes les valeurs de la colonne "grades" par la moyenne calculée
dfconverted2['grades'] = dfconverted2['moyenne_score']

# Supprimer la colonne 'moyenne_score' si vous ne voulez pas la conserver
dfconverted2.drop(columns=['moyenne_score'], inplace=True)
```

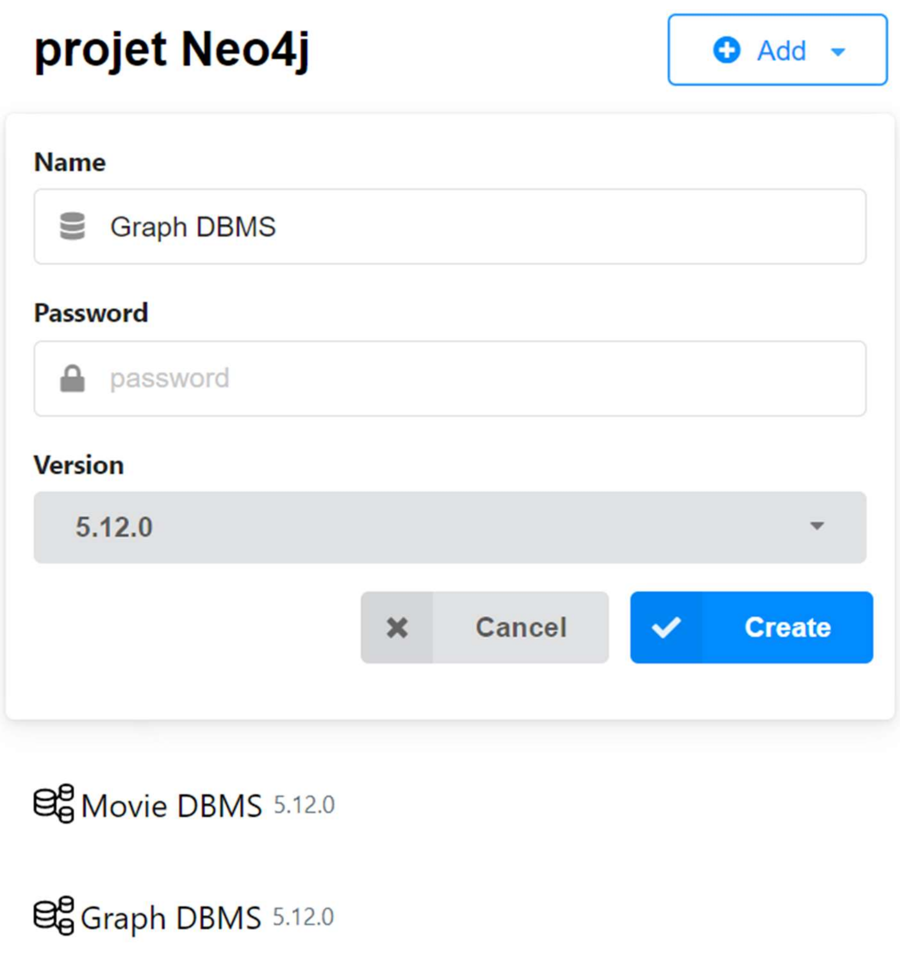
Last step would be to extract the now readable csv file.

Creating a graph database

We first need a database to import data on it. On your project, click the add button, then local DBMS:



Then name your database, give it some password and click the create button.



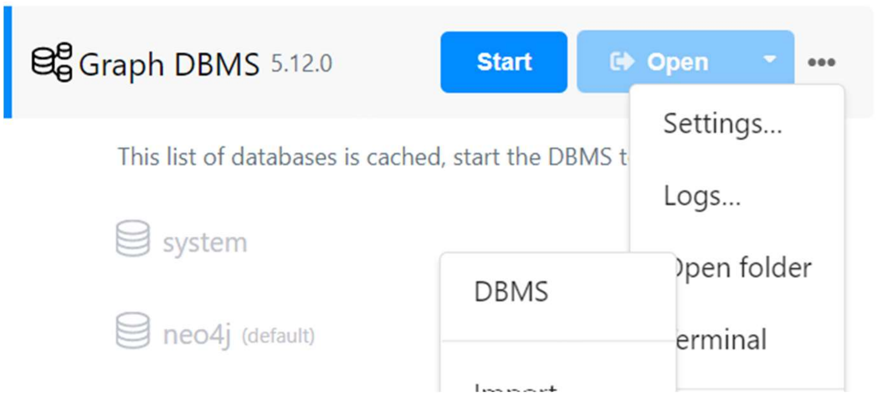
Name: Graph DBMS

Password: password

Version: 5.12.0

Importing the data

Once the database has been created, click the 3 dots, then open folder and Import, It will open the folder you need to put your csv into.



This list of databases is cached, start the DBMS t

Queries

Since our dataset is a difficult one, we are going to do 8 easy queries, 1 complex and 1 hard. We also added an extra Complex and Hard querry.

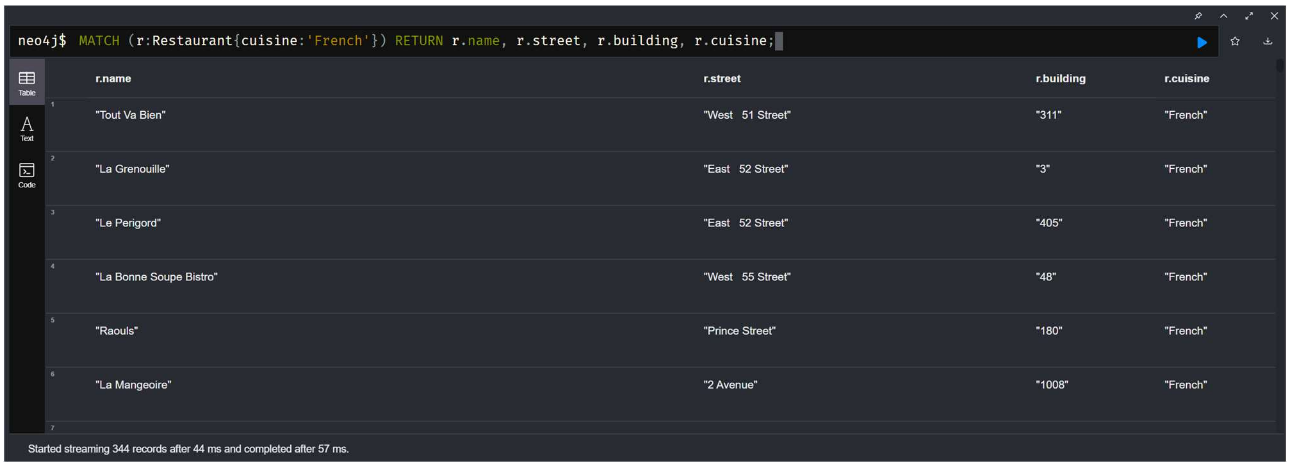
Easy Queries

Query 1 : Get all french restaurants

The expected output of this query is to get only the name and address of all French restaurants and of no other resataurants.

<u>Querry:</u>
MATCH (r:Restaurant{cuisine:'French'}) RETURN r.name, r.street, r.building, r.cuisine;

The MATCH is used to filter restaurants. 'r' is the name of the variable used to deal with the restaurant. The section in bracket {} used to select nodes based on their properties' values. Typically, the syntax would be properties:expectedValue, with expectedValue the value the property is equal to to be selected. In that case the cuisine type should be French, so we simply write cuisine:'French'.
Lastly the RETURN specifies what properties to return. To meet the expectations of the querry, we need to return the name, street and building (street number) of every French restaurant. We also add the cuisine type in the output to check no other cuisine type made it in.



```
neo4j$ MATCH (r:Restaurant{cuisine:'French'}) RETURN r.name, r.street, r.building, r.cuisine;

   r.name                    r.street            r.building   r.cuisine
1  "Tout Va Bien"            "West  51 Street"   "311"        "French"
2  "La Grenouille"           "East  52 Street"   "3"          "French"
3  "Le Perigord"             "East  52 Street"   "405"        "French"
4  "La Bonne Soupe Bistro"   "West  55 Street"   "48"         "French"
5  "Raouls"                  "Prince Street"     "180"        "French"
6  "La Mangeoire"            "2 Avenue"          "1008"       "French"
7

Started streaming 344 records after 44 ms and completed after 57 ms.
```

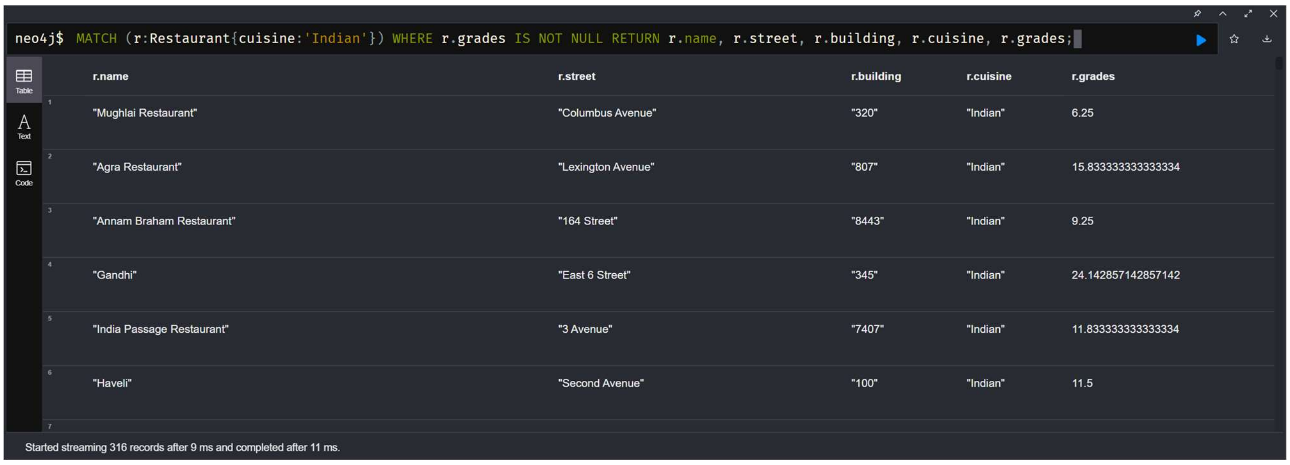
We got 344 results, as expected, we only find all the restaurants' name seem french in the output. Moreover the form of the output matches the expectation of the restaurant's name and address, and all restaurants are French ones.

Query 2 : Get all graded indian restaurants name and address.

The expected result of this querry should have the same collumns as the previous ones, with the added criterion that restaurants with no grades shouldn't appear.

<u>Querry:</u>
MATCH (r:Restaurant{cuisine:'Indian'}) WHERE r.grades IS NOT NULL RETURN r.name, r.street, r.building, r.cuisine, r.grades;

The only structural difference in this querry relatively to the previous one is the addition of a WHERE section which can be used to further detail the conditions on the properties. Here we simply want to check that the grades section isn't empty, hence the IS NOT NULL following the property, which does exactly what it says.
We also added the cuisine type and the grade in the output to check all restaurants are both graded and Indian.



```
neo4j$ MATCH (r:Restaurant{cuisine:'Indian'}) WHERE r.grades IS NOT NULL RETURN r.name, r.street, r.building, r.cuisine, r.grades;
```

	r.name	r.street	r.building	r.cuisine	r.grades
1	"Mughlai Restaurant"	"Columbus Avenue"	"320"	"Indian"	6.25
2	"Agra Restaurant"	"Lexington Avenue"	"807"	"Indian"	15.833333333333334
3	"Annam Braham Restaurant"	"164 Street"	"8443"	"Indian"	9.25
4	"Gandhi"	"East 6 Street"	"345"	"Indian"	24.142857142857142
5	"India Passage Restaurant"	"3 Avenue"	"7407"	"Indian"	11.833333333333334
6	"Haveli"	"Second Avenue"	"100"	"Indian"	11.5
7					

Started streaming 316 records after 9 ms and completed after 11 ms.

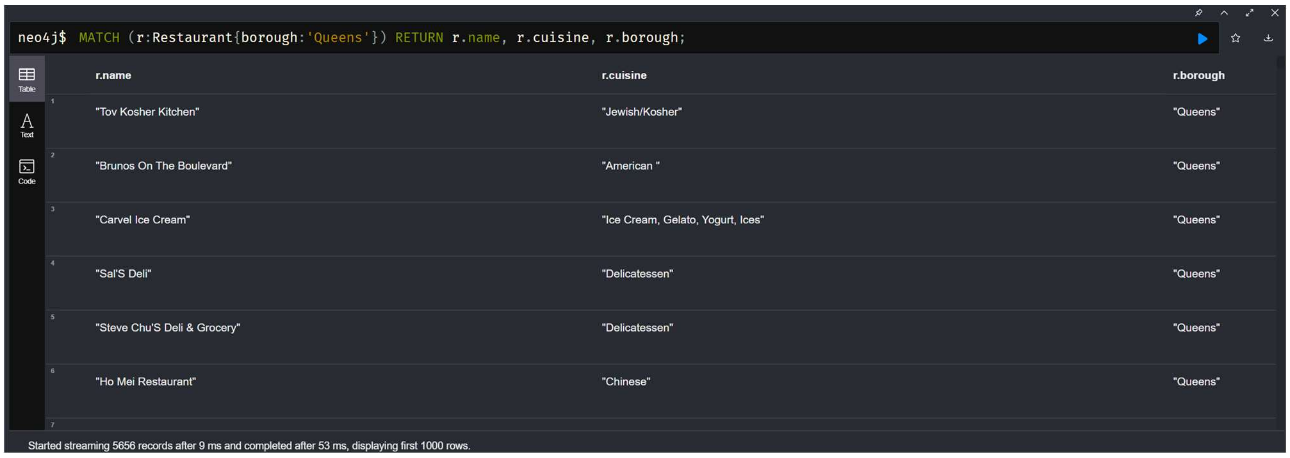
As expected, the results contains all the desired columns, and all restaurants should have been graded at least once.

Query 3 : Get all the Queens' restaurants' name as well as cuisine

The output should only be composed of restaurants which happen to be in the Queens with their respective cuisine type.

<u>Querry:</u>
MATCH (r:Restaurant{borough:'Queens'}) RETURN r.name, r.cuisine, r.borough;

The structure is very much alike that of the first querry, exception made for the output which should be made of the restaurants' name and cuisine. The borough in the output is used to check the restaurants are indeed in the Queens.



```
neo4j$ MATCH (r:Restaurant{borough:'Queens'}) RETURN r.name, r.cuisine, r.borough;
```

	r.name	r.cuisine	r.borough
1	"Tov Kosher Kitchen"	"Jewish/Kosher"	"Queens"
2	"Brunos On The Boulevard"	"American "	"Queens"
3	"Carvel Ice Cream"	"Ice Cream, Gelato, Yogurt, Ices"	"Queens"
4	"Sal'S Deli"	"Delicatessen"	"Queens"
5	"Steve Chu'S Deli & Grocery"	"Delicatessen"	"Queens"
6	"Ho Mei Restaurant"	"Chinese"	"Queens"
7			

Started streaming 5656 records after 9 ms and completed after 53 ms, displaying first 1000 rows.

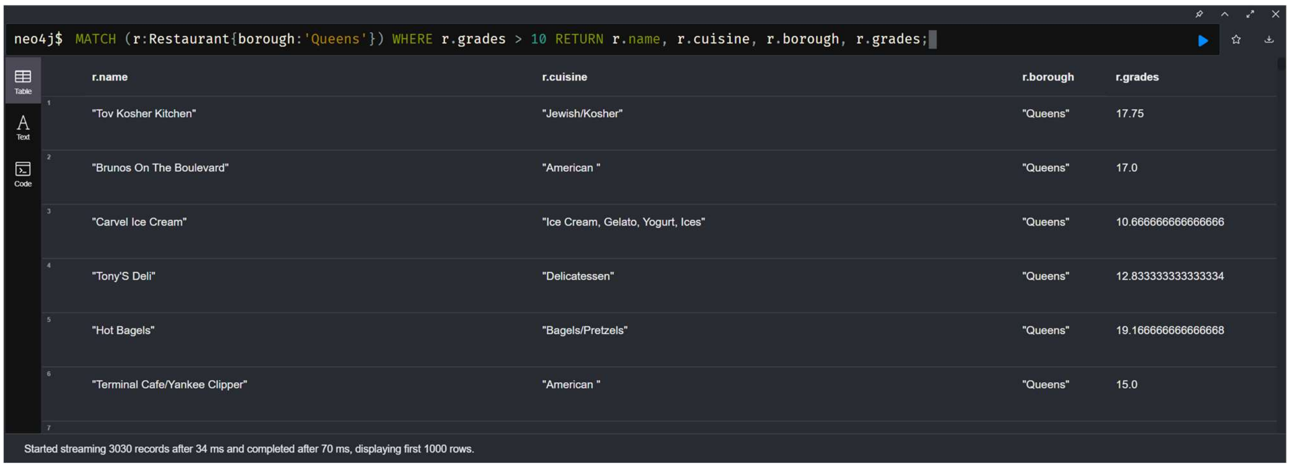
We then observe than the result is composed of restaurants of various cuisine type with their name, all being in the Queens.

Query 4 : Get all of Queens’ restaurants name if their average grade is above 10.

The expected output this time should contain the restaurants’ name and cuisine once again, but only if they are grade on average above 10.

<u>Querry:</u>
MATCH (r:Restaurant{borough:’Queens’})}) WHERE r.grades > 10 RETURN r.name, r.cuisine, r.borough, r.grades;

The querry is very similar to the previous one, with the addition of a where section to filter out restaurants with an average grades under 10 (WHERE r.grades > 10).
This time, we add both the borough property and the grades one to check that the conditions are met.



```
neo4j$ MATCH (r:Restaurant{borough:'Queens'}) WHERE r.grades > 10 RETURN r.name, r.cuisine, r.borough, r.grades;
```

	r.name	r.cuisine	r.borough	r.grades
1	"Tov Kosher Kitchen"	"Jewish/Kosher"	"Queens"	17.75
2	"Brunos On The Boulevard"	"American "	"Queens"	17.0
3	"Carvel Ice Cream"	"Ice Cream, Gelato, Yogurt, Ices"	"Queens"	10.666666666666666
4	"Tony'S Deli"	"Delicatessen"	"Queens"	12.833333333333334
5	"Hot Bagels"	"Bagels/Pretzels"	"Queens"	19.166666666666668
6	"Terminal Cafe/Yankee Clipper"	"American "	"Queens"	15.0
7				

```
Started streaming 3030 records after 34 ms and completed after 70 ms, displaying first 1000 rows.
```

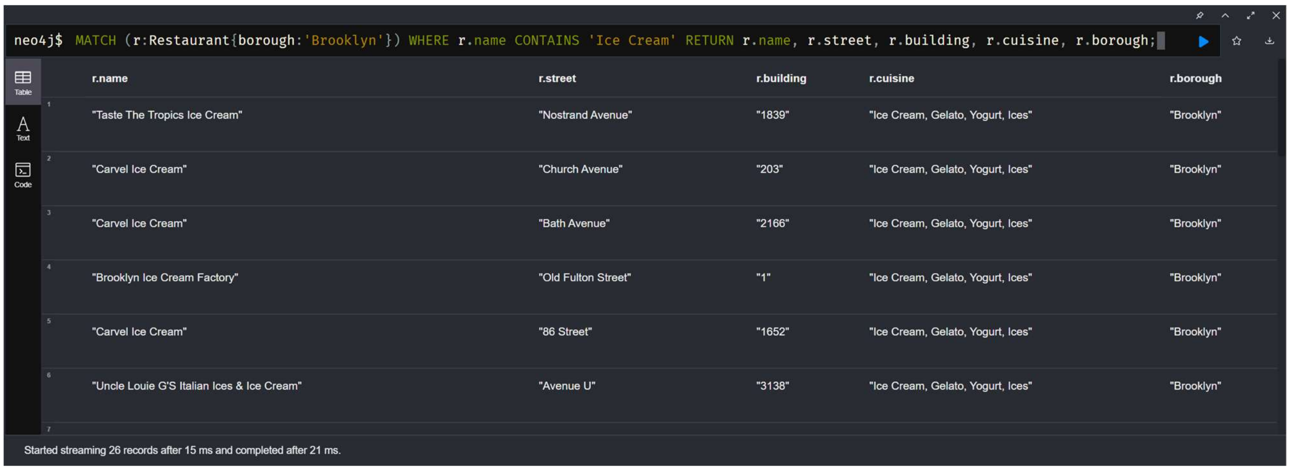
Once again, the result is just as we expected it, with the 4 expected columns, and all restaurants in the Queens with average grades above 10.

Query 5 : Find all restaurants with ‘Ice Cream’ in their name of Brooklyn and get their name, cuisine type and location

The expected output should be a list of restaurants of Brooklyn with Ice Cream in their name, as well as their address (building and street)

<u>Querry:</u>
MATCH (r:Restaurant{borough:’Brooklyn’}) WHERE r.name CONTAINS ‘Ice Cream’ RETURN r.street, r.building, r.cuisine, r.borough;

The result section contains all required fields as well as their borough to ensure they are in Brooklyn.



```
neo4j$ MATCH (r:Restaurant{borough:'Brooklyn'}) WHERE r.name CONTAINS 'Ice Cream' RETURN r.name, r.street, r.building, r.cuisine, r.borough;
```

r.name	r.street	r.building	r.cuisine	r.borough
"Taste The Tropics Ice Cream"	"Nostrand Avenue"	"1839"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"
"Carvel Ice Cream"	"Church Avenue"	"203"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"
"Carvel Ice Cream"	"Bath Avenue"	"2166"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"
"Brooklyn Ice Cream Factory"	"Old Fulton Street"	"1"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"
"Carvel Ice Cream"	"86 Street"	"1652"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"
"Uncle Louie G'S Italian Ices & Ice Cream"	"Avenue U"	"3138"	"Ice Cream, Gelato, Yogurt, Ices"	"Brooklyn"

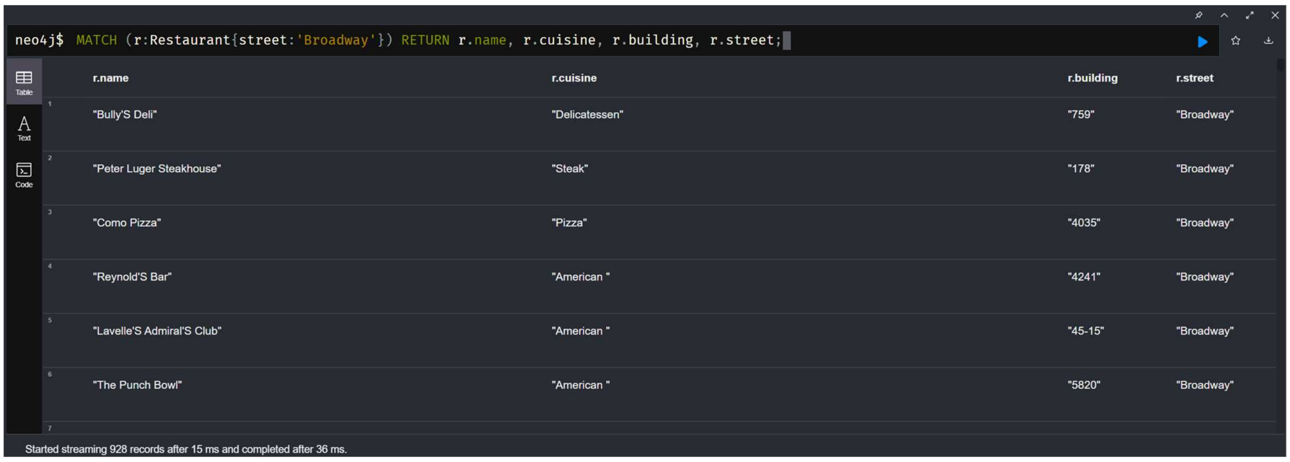
Started streaming 26 records after 15 ms and completed after 21 ms.

The output matches the expectation, all fields are here and all the restaurants are in Brooklyn and have Ice Cream in their name.

Query 6 : Get all restaurants of Broadway with their cuisine type and their building

The expected output should hold just that as well as the building to help better locate them.

<u>Querry:</u>
MATCH (r:Restaurant{street:’Broadway’}) RETURN r.name, r.cuisine, r.building, r.street;



```
neo4j$ MATCH (r:Restaurant{street:'Broadway'}) RETURN r.name, r.cuisine, r.building, r.street;
```

r.name	r.cuisine	r.building	r.street
"Bully'S Deli"	"Delicatessen"	"759"	"Broadway"
"Peter Luger Steakhouse"	"Steak"	"178"	"Broadway"
"Como Pizza"	"Pizza"	"4035"	"Broadway"
"Reynold'S Bar"	"American "	"4241"	"Broadway"
"Lavelle'S Admiral'S Club"	"American "	"45-15"	"Broadway"
"The Punch Bowl"	"American "	"5820"	"Broadway"

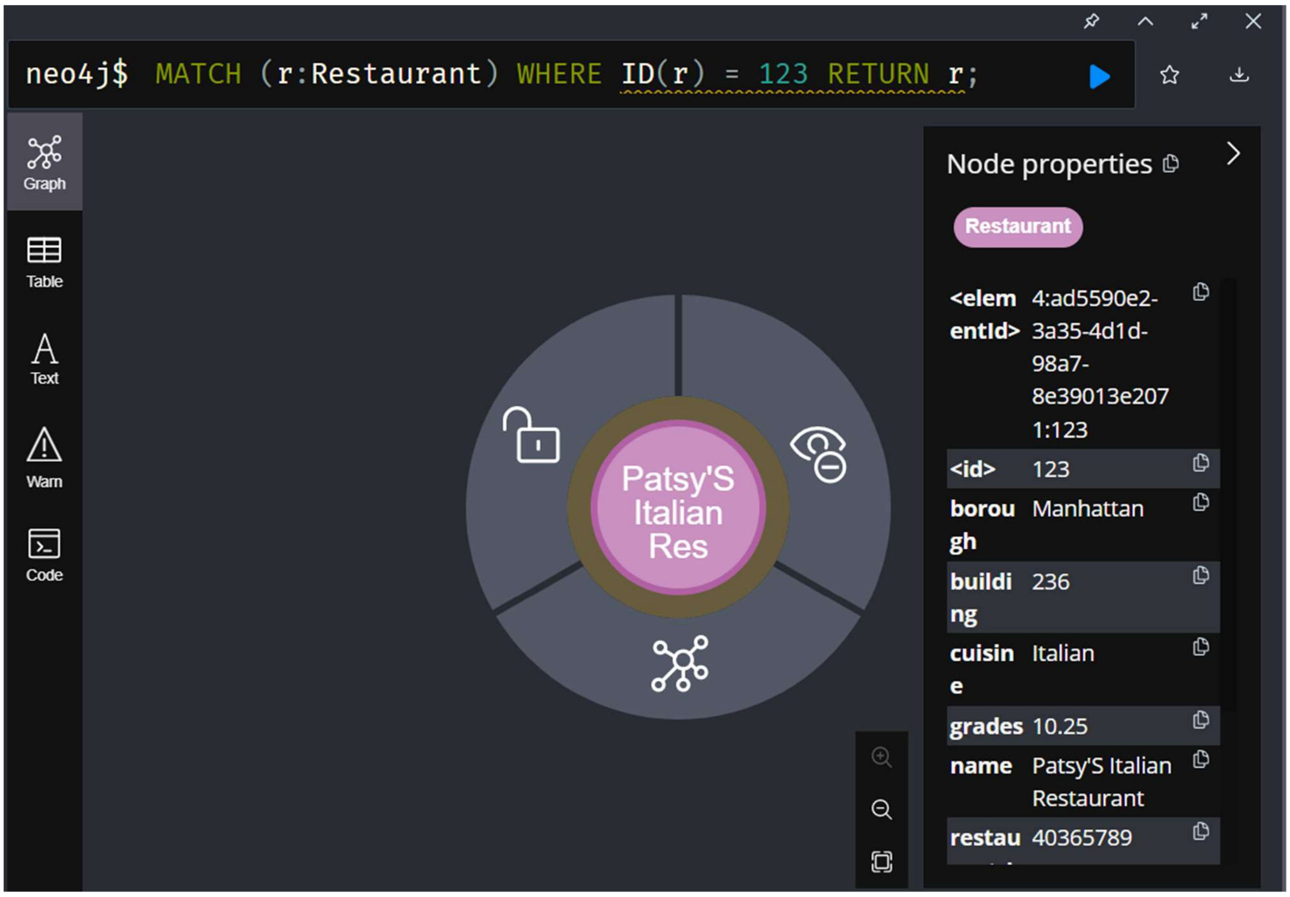
Started streaming 928 records after 15 ms and completed after 36 ms.

The ouput matches the expectation as for the columns contained. Moreover all restaurants are on Broadway.

Query 7 : The restaurant whose id is 123

The expected output is the restaurant of id 123 with all related information.
It is to be noted that as we return a node and not properties of a node, the output will take the form of a graph (with a singular dot in this case).

Querry:
MATCH (r:Restaurant) WHERE ID(r) = 123 Return r;



```
neo4j$ MATCH (r:Restaurant) WHERE ID(r) = 123 RETURN r;
```

Node properties

Restaurant

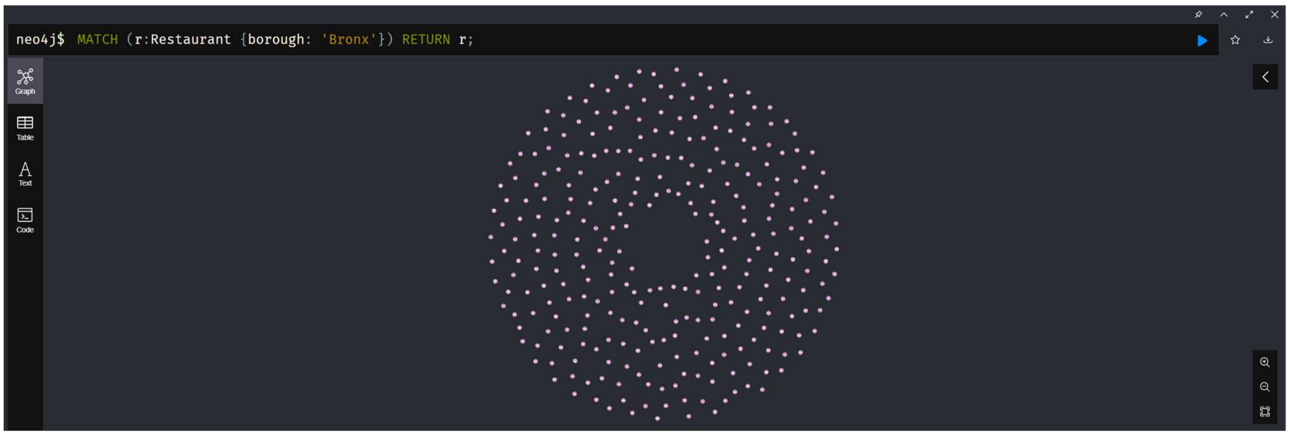
<elementId>	4:ad5590e2-3a35-4d1d-98a7-8e39013e2071:123
<id>	123
borough	Manhattan
building	236
cuisine	Italian
grades	10.25
name	Patsy'S Italian Restaurant
restau	40365789

The output is made of a single restaurant with all its information in a graph. Its id is 123, as expected.

Query 8 : Get all the information of all restaurants in the Bronx

Here the output should take the same form as for the previous querry, except it should hold more nodes.

<u>Querry:</u>
MATCH (r:Restaurant {borough: ‘Bronx’}) RETURN r;



```
neo4j$ MATCH (r:Restaurant {borough: 'Bronx'}) RETURN r;
```

The screenshot is only a sample of the output and still matches the expectations.

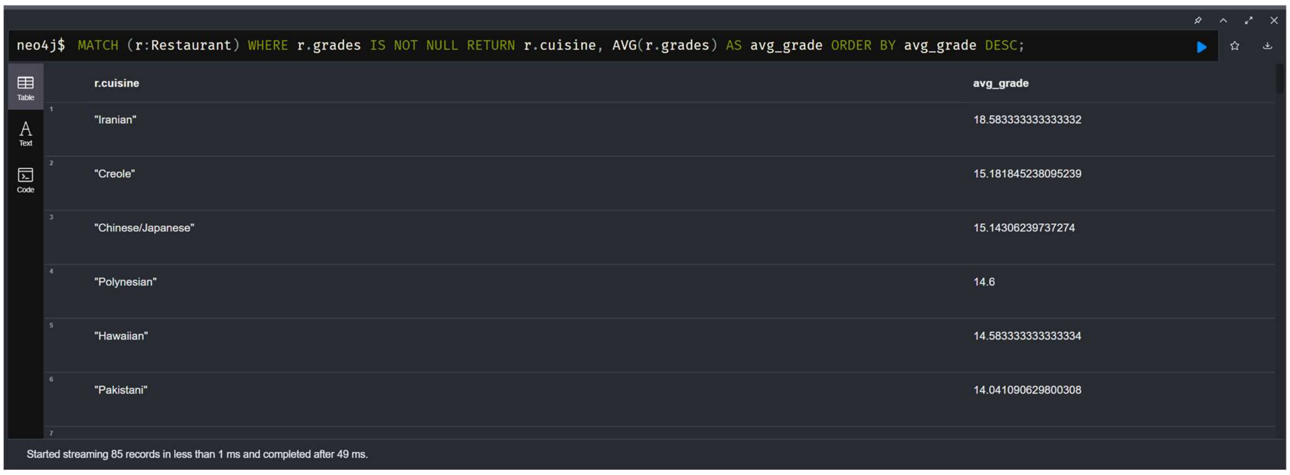
Complex Query

Query : Get the average grade of all cuisine types.

The expected output should hold the cuisine type with their corresponding average grade across all restaurants cooking said cuisine.
It is to be noted that the output will hold the average of the average of grades of all restaurants. It is due to our treatment of the dataset and does not take into account the number of grades each restaurant has or their population.

Querry:
MATCH (r:Restaurant) WHERE r.grades IS NOT NULL RETURN r.cuisine, AVG(r.grades) AS avg_grade ORDER BY avg_grade DESC;

Here We check that all restaurants are indeed graded. The AVG(r.grades) computes the average of all grades, the whole being defaultly grouped by the other field, in that case the cuisine type. The alias (AS avg_grade) is used to later order the results according to the average grade of each cuisine type, here in descending order(DESC).



```
neo4j$ MATCH (r:Restaurant) WHERE r.grades IS NOT NULL RETURN r.cuisine, AVG(r.grades) AS avg_grade ORDER BY avg_grade DESC;
```

	r.cuisine	avg_grade
1	"Iranian"	18.583333333333332
2	"Creole"	15.181845238095239
3	"Chinese/Japanese"	15.14306239737274
4	"Polynesian"	14.6
5	"Hawaiian"	14.583333333333334
6	"Pakistani"	14.041090629800308
7		

Started streaming 85 records in less than 1 ms and completed after 49 ms.

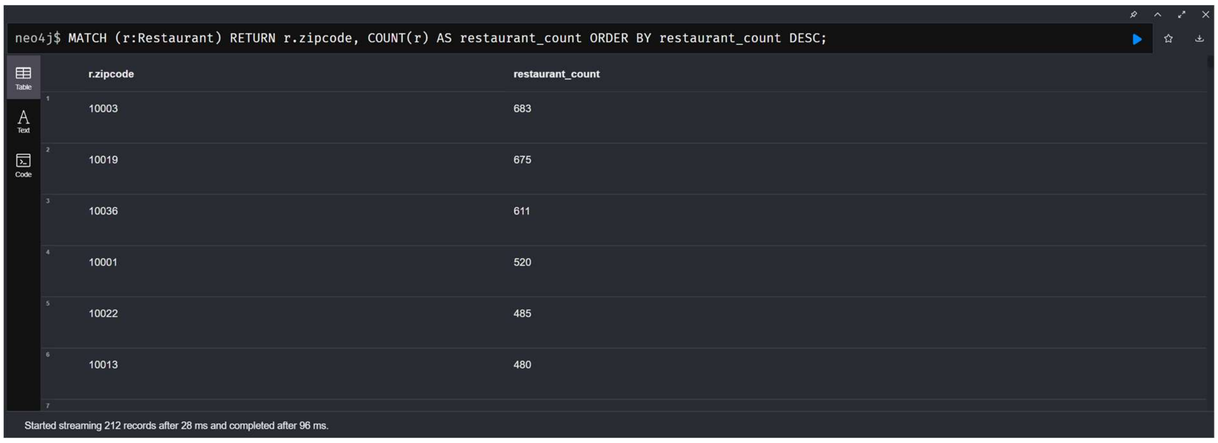
Here the output matches has the 2 expected Fields, with a grade by cuisine type and the grades in descending order.

Query : Count the number of restaurants per zipcode.

The goal of this querry is to compute the number of restaurants each zipcode holds.
The grouping will once again be automatically handled, and as the 2 return fields are the zip code and the number of restaurants, it should work as intended.

Querry:
MATCH (r:Restaurant) RETURN r.borough, COUNT(r) AS restaurant_count ORDER_BY restaurant_count DESC

The main difference with the previous querry is the replacement of AVG by COUNT to get a number of occurrence rather than an average



```
neo4j$ MATCH (r:Restaurant) RETURN r.zipcode, COUNT(r) AS restaurant_count ORDER BY restaurant_count DESC;
```

r.zipcode	restaurant_count
10003	683
10019	675
10036	611
10001	520
10022	485
10013	480

```
Started streaming 212 records after 28 ms and completed after 96 ms.
```

The result does hold both required fields, and restaurant counts are put in descending order.

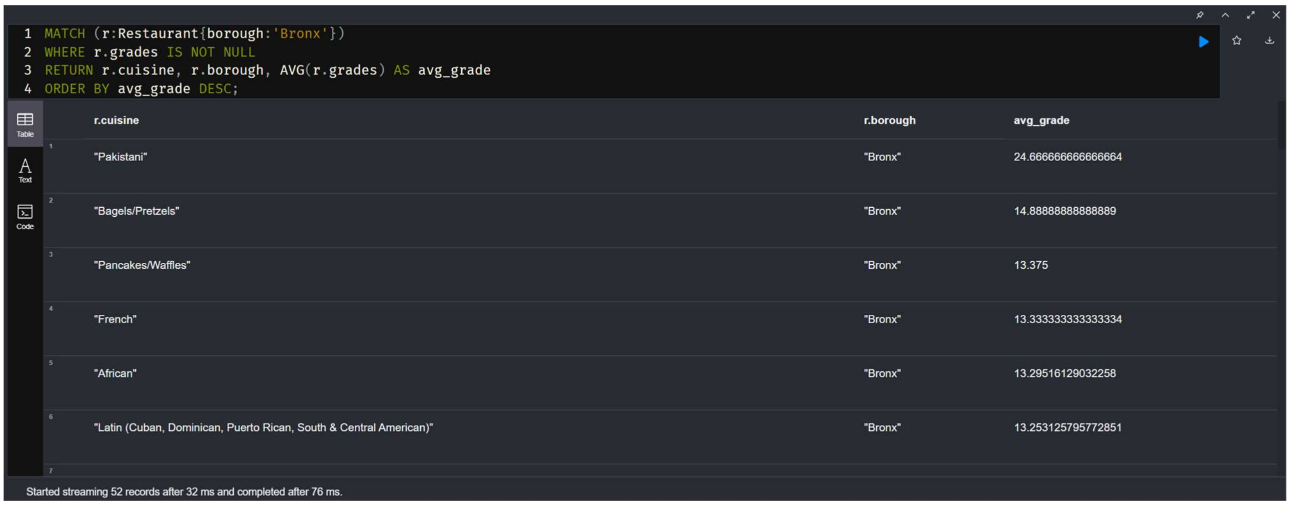
Hard Query

Query : Get the average grade of all cuisine types in the Bronx.

The result should be the same as for the first complex querry, except that it should only take into account the restaurants in the Bronx. The borough columns is used to check that the constraints are respected.

<u>Query:</u>
MATCH (r:Restaurant{borough:'Bronx'}) WHERE EXISTS(r.grades) // Check if restaurant has grades RETURN r.cuisine, r.borough, AVG(r.grades) AS avg_grade GROUP BY r.cuisine ORDER BY avg_grade DESC;

The key difference between this query and the first complex one is the extra filter on the restaurants using its properties, just as was done in the simple querries(r:Restaurant{borough:'Bronx'}).
The Grouping is still handled automatically and properly according to the cuisine type as the borough is identical for all nodes.



```
MATCH (r:Restaurant{borough:'Bronx'})
WHERE r.grades IS NOT NULL
RETURN r.cuisine, r.borough, AVG(r.grades) AS avg_grade
ORDER BY avg_grade DESC;
```

r.cuisine	r.borough	avg_grade
"Pakistani"	"Bronx"	24.666666666666664
"Bagels/Pretzels"	"Bronx"	14.88888888888889
"Pancakes/Waffles"	"Bronx"	13.375
"French"	"Bronx"	13.333333333333334
"African"	"Bronx"	13.29516129032258
"Latin (Cuban, Dominican, Puerto Rican, South & Central American)"	"Bronx"	13.253125795772851

```
Started streaming 52 records after 32 ms and completed after 76 ms.
```

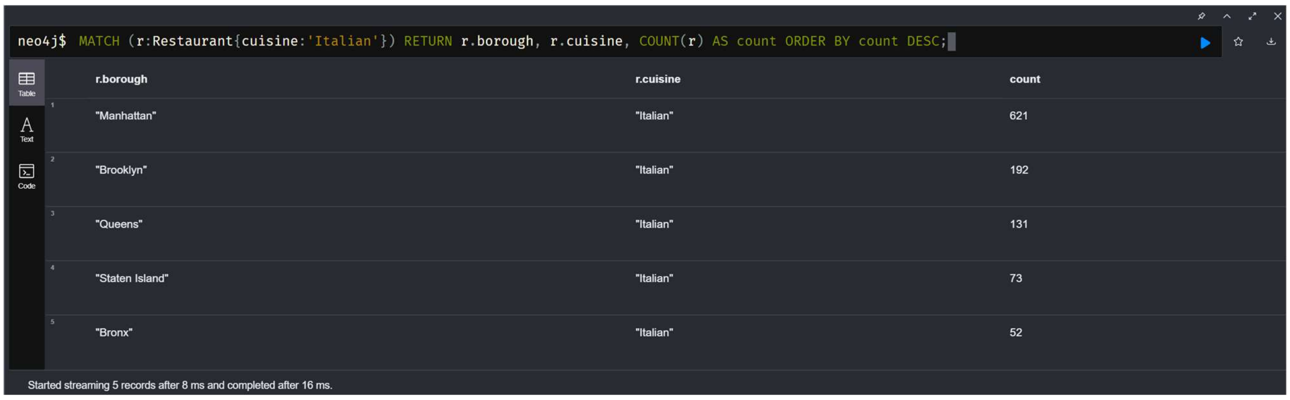
The output holds the 3 expected columns, all restaurants type are correctly filtered to be in the Bronx and all grades are put in descending order.

Query : Count the number of Italian restaurants per borough.

The goal of this querry is to obtain 3 fields: the borough, the number of Italian restaurants and the cuisine type to make sure the constraints are respected.

<u>Query:</u>
MATCH (r:Restaurant{cuisine:'Italian}) RETURN r.borough, r.cuisine, Count(r) AS count ORDER BY count DESC;

The query is very similar to the second complex one, with only the restriction on cuisine type being added.
As for the previous query, the grouping will be handled automatically following the borough.



```
neo4j$ MATCH (r:Restaurant{cuisine:'Italian'}) RETURN r.borough, r.cuisine, COUNT(r) AS count ORDER BY count DESC;
```

	r.borough	r.cuisine	count
1	"Manhattan"	"Italian"	621
2	"Brooklyn"	"Italian"	192
3	"Queens"	"Italian"	131
4	"Staten Island"	"Italian"	73
5	"Bronx"	"Italian"	52

Started streaming 5 records after 8 ms and completed after 16 ms.

As expected, we do get all 3 requested fields, with the count of restaurants in descending order and all of them being Italian.