Analyzing and Mitigating Gun Violence: A NoSQL Database-Driven Approach Using MongoDB and Neo4j on NYPD Shooting Incident Data

Rashmi Shree Veeraiah
Sachin Kumar Srinivasa Murthy
Bhavana Prasad Kote
Maharsh Soni

A. Abstract

Shooting incidents in school and gun violence are considered an American **epidemic**. Gun violence is one of the leading causes of death among children and teenagers in the United States. It is estimated that around 1,17,345 people are shot in America every year. There has been 2032 school shooting incidents in the US since 1970. People of all age groups are being impacted due to gun violence. In the recent post-pandemic era, there has been an upsurge in shootings incidents. Our focus is to analyze the gun violence incidents data provided by the City of New York through OPEN API Access. Using this dataset, we design efficient NoSQL Database driven Analytical solutions using MongoDB (document) and Neo4j (graph) tools. Through our analysis, we aim to provide critical insights that could optimize policing efforts with reduced budgets to minimize shooting incidents and gun violence.

B. Motivation

Gun violence affects families and the country, creating emotional damage and fear among people, which can negatively impact the country's outlook, and the people's lifestyle. We are motivated to explore historical data and provide data-driven solutions and interventions that the NYPD can incorporate to prevent and minimize such events at scale. In addition to the cause, we choose the NYPD Shooting Incident Data by cityofnewyork.us due to the following design challenges and features:

1. The dataset has four entities VICTIM, PERPETRATOR, LOCATION, and INCIDENT. But it only has one unique identifier, which is the INCIDENT_KEY. Therefore, we might have to derive Unique identifiers for other entities without a key if necessary.
2. Each incident can involve one or more victims and perpetrators. While each row of the Data reports a victim involved in an incident uniquely, there is no unique identifier for the perpetrator. Addressing this challenge is necessary as it is impossible to go back in time and collect something that had to be collected.
3. Each location is uniquely identified by the Latitude and Longitude coordinates, enabling us to design Geo-Spatial Analytical Solutions.

C. Dataset details

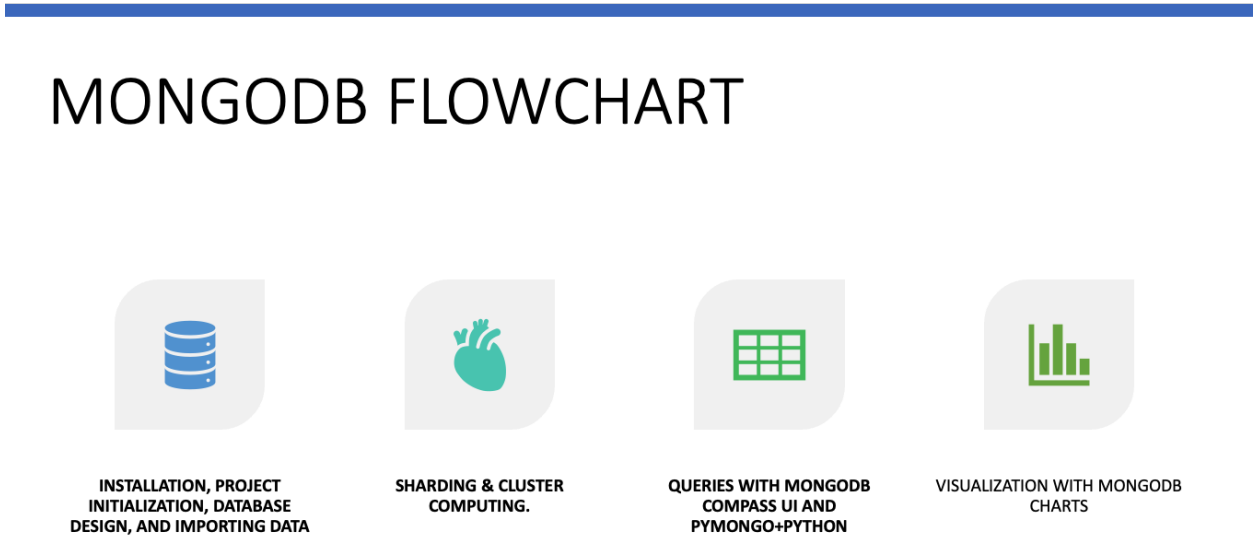
Link: https://catalog.data.gov/dataset/nypd-shooting-incident-data-historic

The dataset contains four entities - Incidents, Locations, Victims, and Perpetrators. The overall dataset includes around 7243 rows and 22 columns, out of which we engineer three features (PERP_ID, VICTIM_ID, MASS_SHOOTING). A snapshot of the dataset information is below:

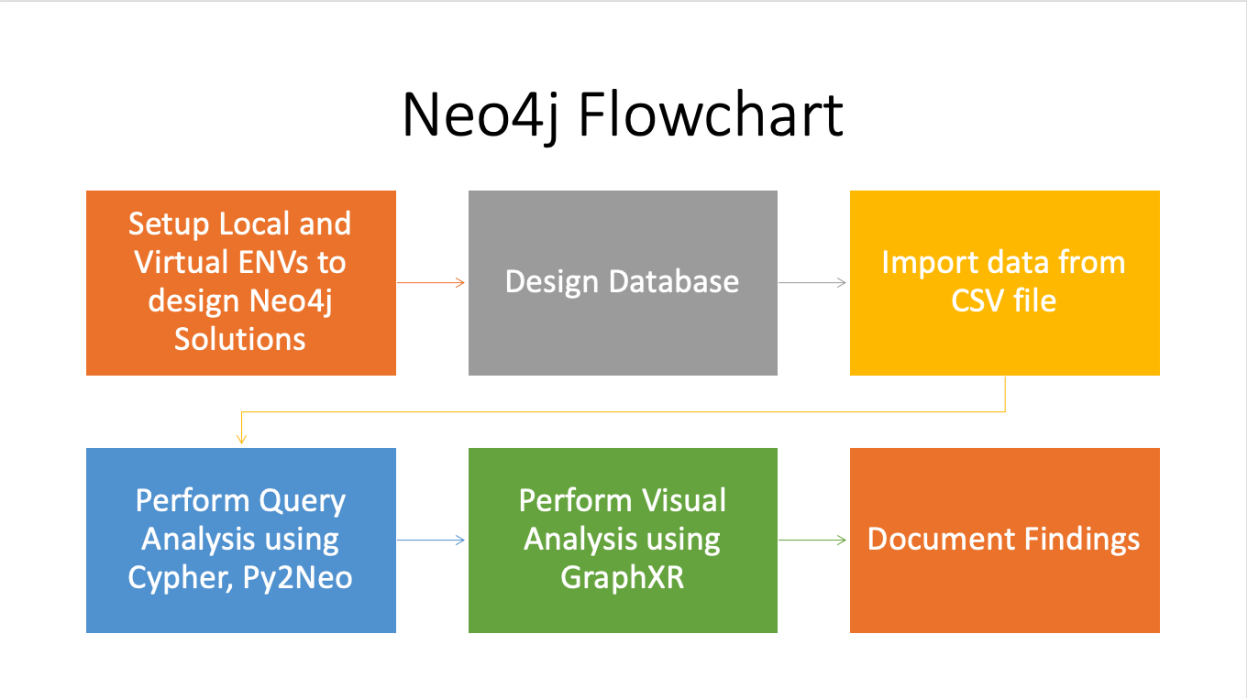
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7243 entries, 0 to 7242
Data columns (total 22 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   PERP_ID                 7243 non-null   int64  
 1   VICTIM_ID               7243 non-null   int64  
 2   INCIDENT_KEY            7243 non-null   int64  
 3   OCCUR_DATE              7243 non-null   object 
 4   OCCUR_TIME              7243 non-null   object 
 5   BORO                    7243 non-null   object 
 6   PRECINCT                7243 non-null   int64  
 7   JURISDICTION_CODE       7243 non-null   int64  
 8   LOCATION_DESC           7243 non-null   object 
 9   STATISTICAL_MURDER_FLAG 7243 non-null   bool   
 10  PERP_AGE_GROUP          7243 non-null   object 
 11  PERP_SEX                7243 non-null   object 
 12  PERP_RACE               7243 non-null   object 
 13  VIC_AGE_GROUP           7243 non-null   object 
 14  VIC_SEX                 7243 non-null   object 
 15  VIC_RACE                7243 non-null   object 
 16  X_COORD_CD              7243 non-null   float64
 17  Y_COORD_CD              7243 non-null   float64
 18  Latitude                7243 non-null   float64
 19  Longitude               7243 non-null   float64
 20  Lon_Lat                 7243 non-null   object 
 21  MASS_SHOOTING           7243 non-null   object 
dtypes: bool(1), float64(4), int64(5), object(12)
memory usage: 1.2+ MB
```

D. MongoDB Workflow/ Flowchart:



E. Neo4J Workflow/ Flowchart:



F. Technical difficulty

1. **Learning Curve**

- We had to learn and deliver solutions with many tools in a mere ten days including MongoDB compass, **MongoDB** Atlas, Neo4j Desktop, **Neo4j Aura** DB, **Cypher** query language, **MQL** querying with **Pymongo**+Python programming language, **GraphXR** connector, and **Py2Neo**.
- Initially, we faced issues related to connectivity to the cluster in MongoDB Atlas from Python, which was resolved successfully after considerable effort.
- While performing visualization using Mongo Charts, the dashboard and the charts were not visible to all the other teammates and in the blog written publicly. This was sorted out by enabling the permissions for group and public(access with URL link) and using the embed option to include in the blog.
- Since we are adjusted to thinking from an SQL perspective (entities and relationships) for writing queries, it was difficult to unlearn and get acquainted with using JSON-data format and query constructs in MongoDB and Cypher query language in Neo4j.

2. **Time Management.**

- Managing multiple deliverables, design of solutions, running experiments, documenting outcomes, drafting a quality blog, and recording the process for a youtube video was quite challenging. We gained significant experience managing time and chasing deliverables, an essential skill to perform well in the industry.

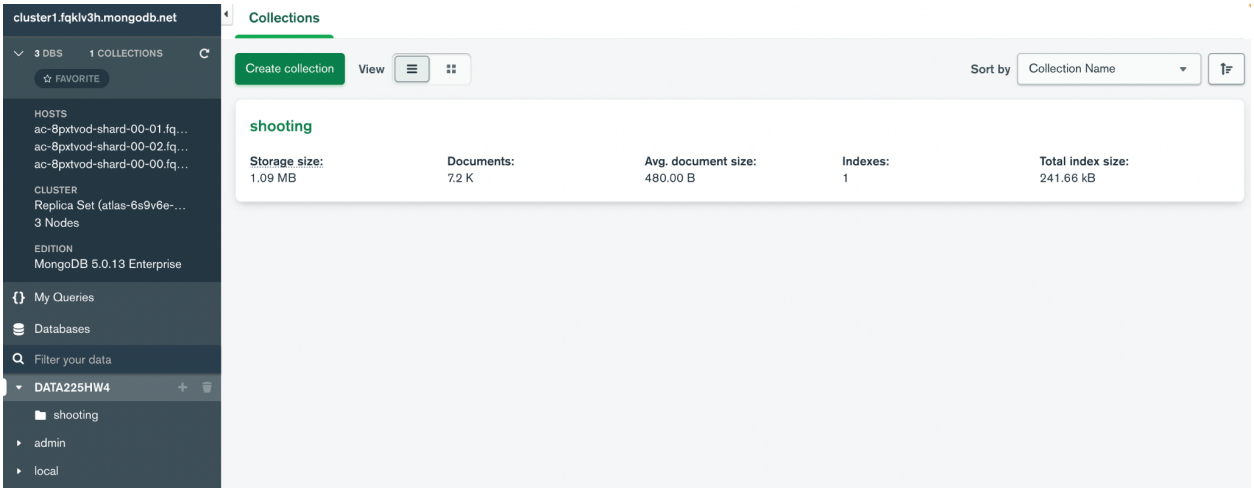
G. MongoDB Solution

Step-1: Installation, Project Initialization, Database Design, and Importing Data

We installed mongodb-community using the brew command, and we installed the pymongo, and dnspython client libraries to interact with the hosted MongoDB database.

```
bhavana@Bhavanas-MacBook-Pro ~ % pip install pymongo
Collecting pymongo
  Downloading pymongo-4.3.2-cp310-cp310-macosx_10_9_universal2.whl (412 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 412.7/412.7 kB 2.0 MB/s eta 0:00:00
Collecting dnspython<3.0.0,>=1.16.0
  Downloading dnspython-2.2.1-py3-none-any.whl (269 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 269.1/269.1 kB 1.4 MB/s eta 0:00:00
Installing collected packages: dnspython, pymongo
Successfully installed dnspython-2.2.1 pymongo-4.3.2
```

We can use the MongoDB Compass UI solution to create a collection. However, we created this using pymongo, and below, we provide a screenshot of the UI solution to validate the success of our pymongo python script.



We connect to the MongoDB Atlas database using Mongosh via CLI.

```
(base) maharshsoni@Maharshs-MacBook-Air ~ % mongosh "mongodb+srv://nyshootingcluster.yq7k5n5.mongodb.net/myFirstDatabase" --apiVersion 1 --user
name g4mongo

Enter password:
Current Mongosh Log ID: 63689b9cb5148f0e1b896bdc
Connecting to:          mongodb+srv://<credentials>@nyshootingcluster.yq7k5n5.mongodb.net/myFirstDatabase?appName=mongosh+1.6.0
MongoInvalidArgumentError: Password cannot be empty
(base) maharshsoni@Maharshs-MacBook-Air ~ % mongosh "mongodb+srv://nyshootingcluster.yq7k5n5.mongodb.net/nycDb" --apiVersion 1 --username g4mon
go
Enter password: *********
Current Mongosh Log ID: 63689cc53ffba2866a76b160
Connecting to:          mongodb+srv://<credentials>@nyshootingcluster.yq7k5n5.mongodb.net/nycDb?appName=mongosh+1.6.0
Using MongoDB:          5.0.13 (API Version 1)
Using Mongosh:          1.6.0

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

Atlas atlas-13ucd7-shard-0 [primary] nycDb>
```

Below is our python script to connect to MongoDB, create a new database collection, and insert our dataset into collections by using the **insert_many** function.

python_mongodb_HW4.py

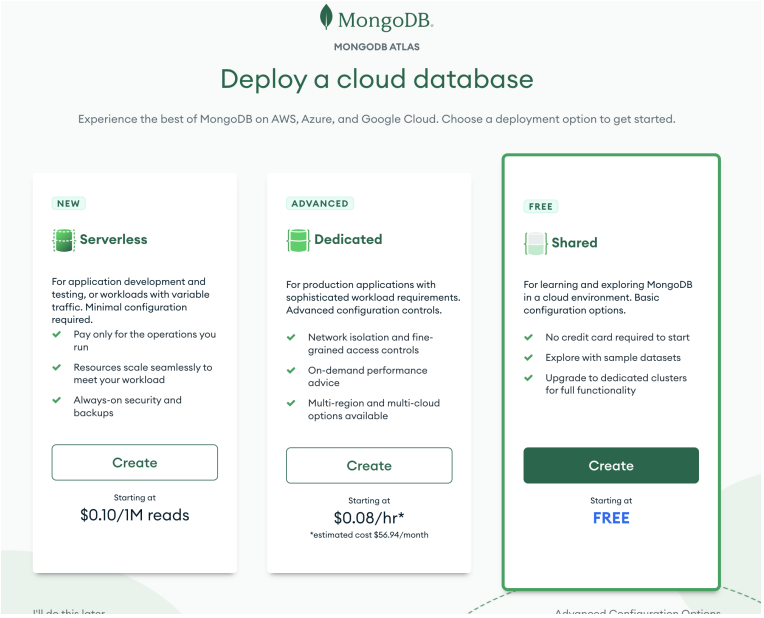
```python
from pymongo import MongoClient
import urllib
import csv
import json
import pandas as pd

def get_database():
    connection_str = 'mongodb+srv://bhavanaprasad07:'+urllib.parse.quote('Bhavana@07')+'@cluster1.fqklv3h.mongodb.net/?retryWrites=true&w=majority'
    client = MongoClient(connection_str)
    print("connection established: ", client)
    print(client.list_database_names())

    return client

client = get_database()

db_name = client['DATA225HW4']
collection = db_name['shooting']
print(collection)


df = pd.read_csv('NYPD_Shooting_Incident.csv')
#data = df.to_dict(orient = "records")
df.to_json('NYPD_Shooting_Incidents.json', orient = 'records', date_format = 'iso')

file = open('NYPD_Shooting_Incidents.json','r')
data = json.load(file)

collection.insert_many(data)
```

Step-2: Sharding & Cluster Computing.

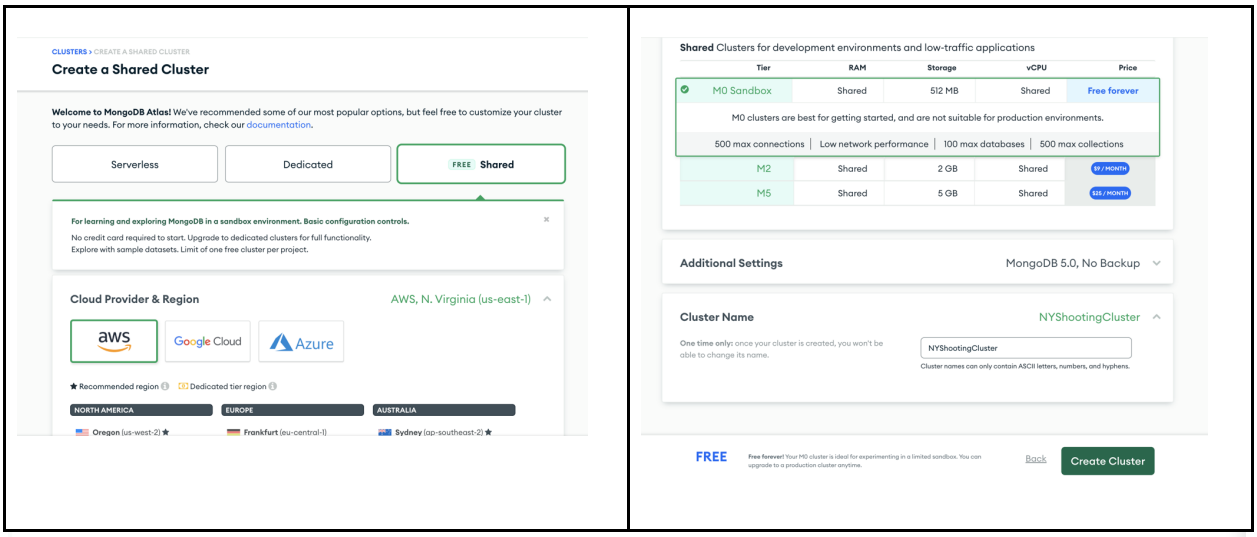
In mongo DB Atlas, we can create three different types of cluster tiers:

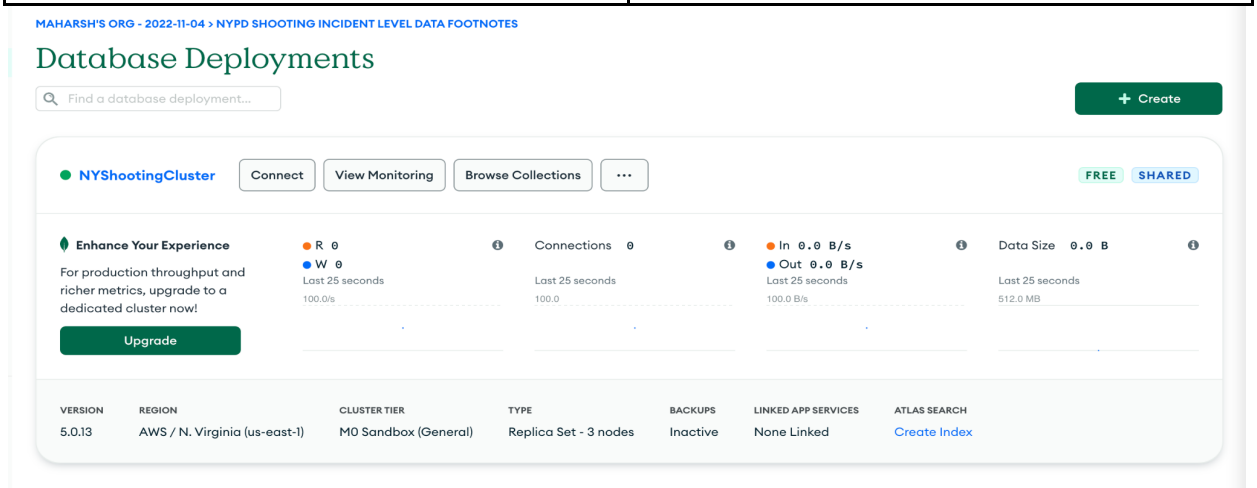
1. **Serverless cluster (Cost-associated)**
2. **Dedicated cluster (Cost-associated)**
3. **Shared cluster (Free)**

We have used a shared cluster due to no cost associated with the solution for this project.

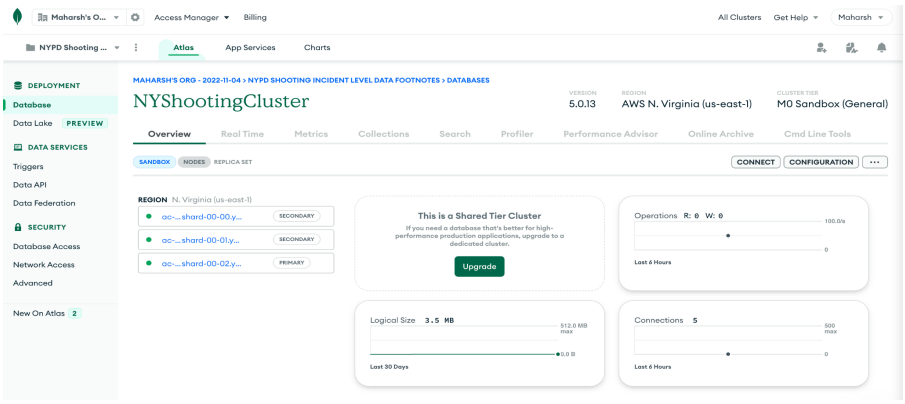


Creation of a shared cluster:

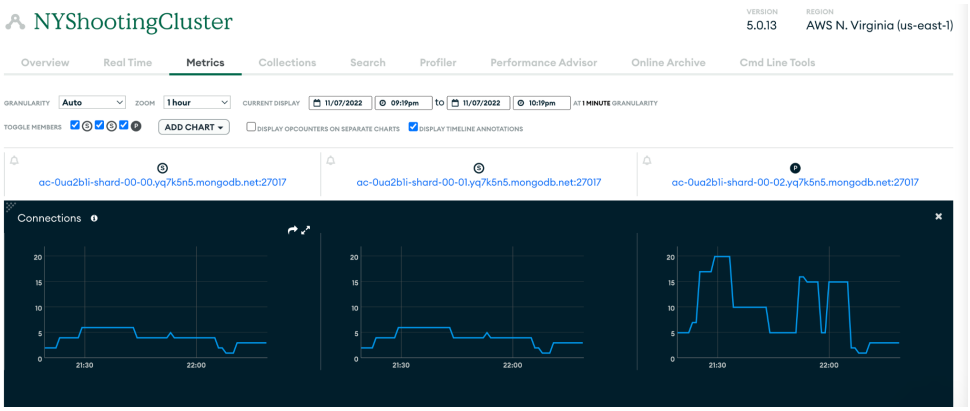




Three different shards are created into MongoDB; One primary and two secondary shards. We are presenting the shards details, and activity/ health log dashboard below:

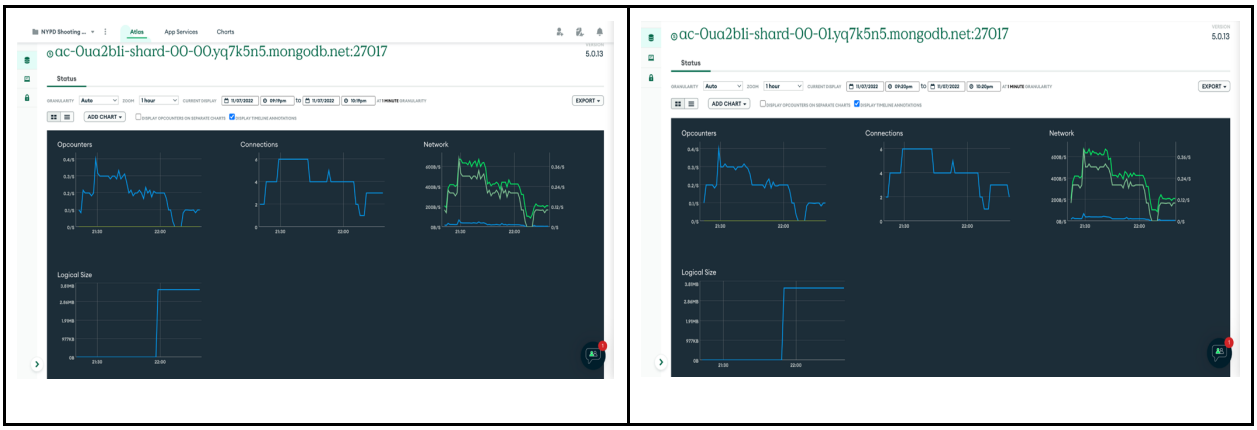


In the above image, we can see one primary and two secondary shards.



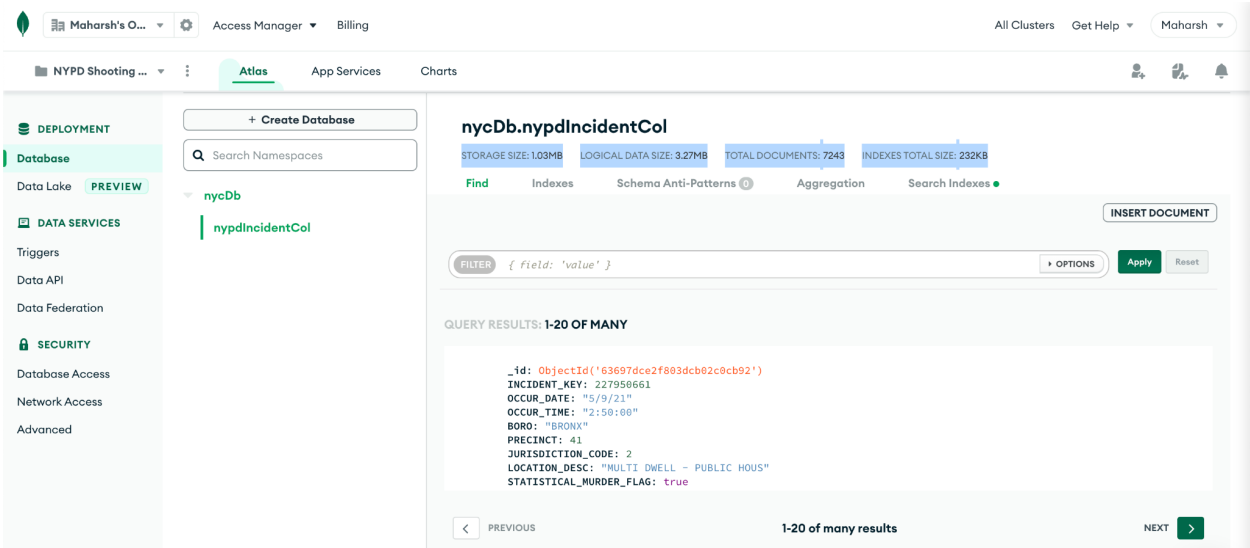
Metrics Dashboard of all shards (S, S, P)

Primary Shard Dashboard



Secondary Shards Dashboard

Below is a sample document data from the created shared database collections cluster.



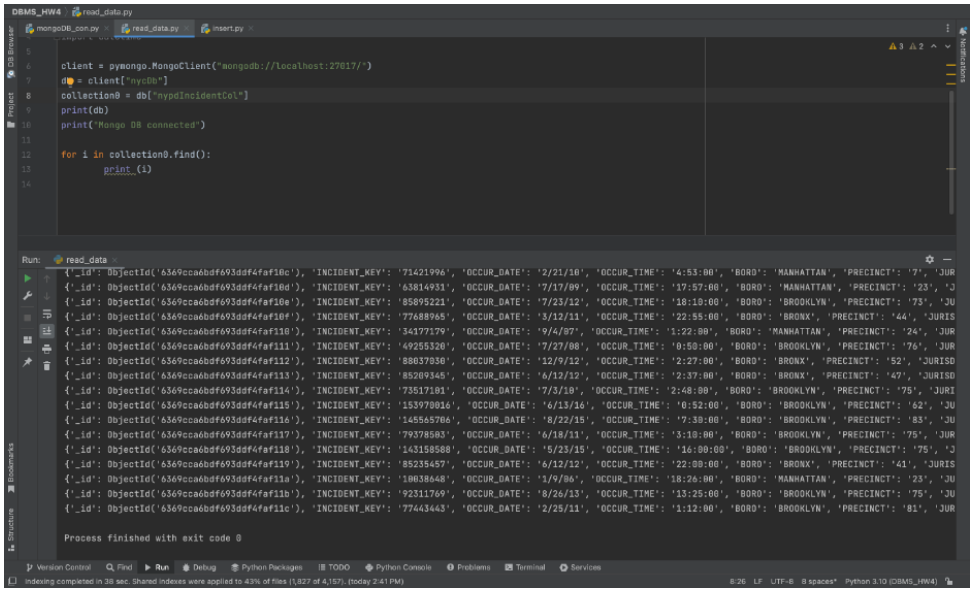
Step-3: Queries with MongoDB Compass UI and PyMongo+Python

In the interest of time, and minimal flexibility offered by the database, we perform fewer aggregation pipelines in comparison to the normal queries.

<u>Query-1: Performing SELECT * FROM DB (Parallel function in PyMongo)</u>

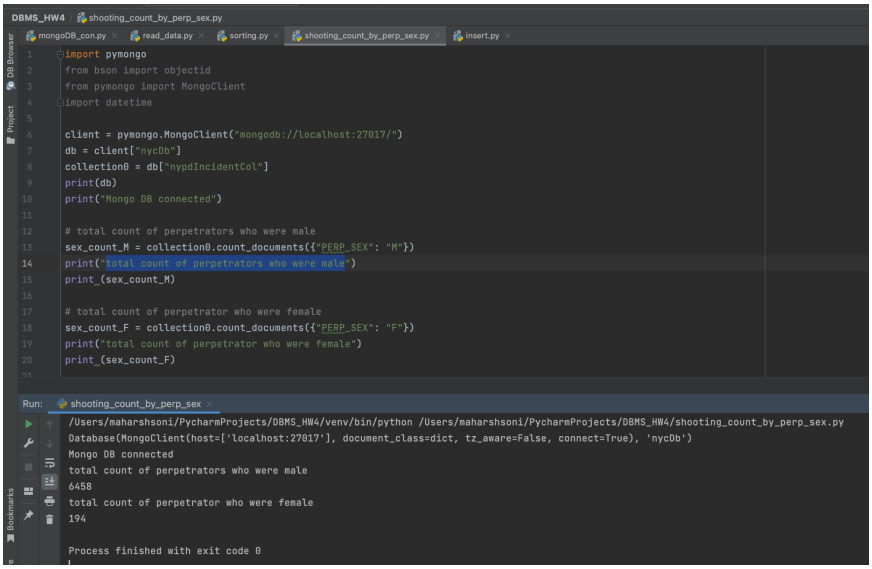
MongoDB has a method, find() to search and retrieve all documents in the collection. Find() returns all data (works like select *)

Result:



Query-2: Display aggregated count of perpetrators by Gender.

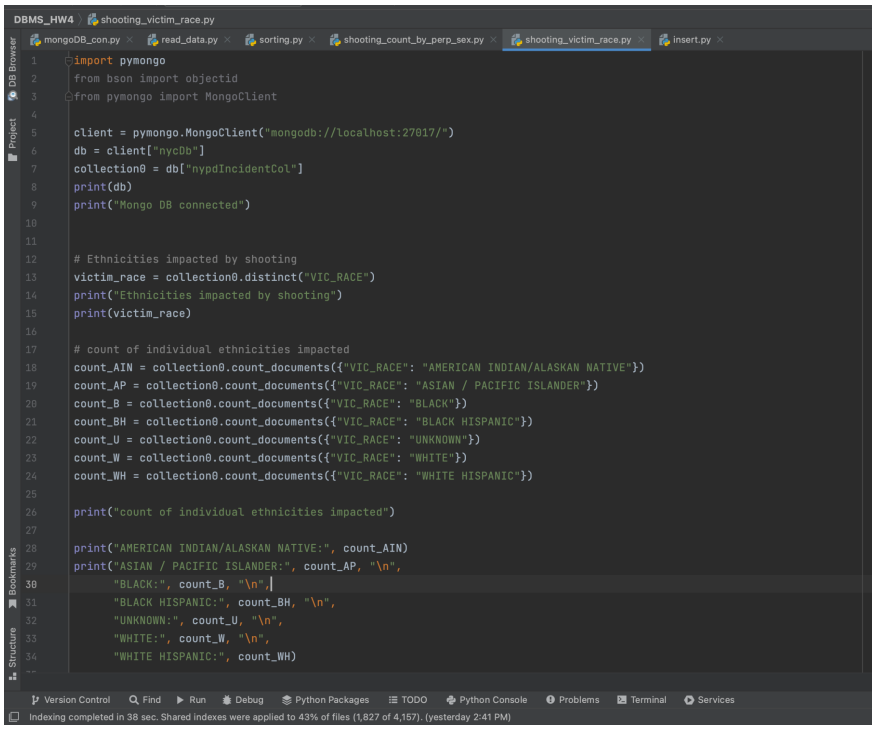
Result- There are more Male Perpetrators than females.



```
import pymongo
from bson import objectid
from pymongo import MongoClient
import datetime

client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["nycDb"]
collection0 = db["nypdIncidentCol"]
print(db)
print("Mongo DB connected")

# total count of perpetrators who were male
sex_count_M = collection0.count_documents({"PERP_SEX": "M"})
print("total count of perpetrators who were male")
print_(sex_count_M)

# total count of perpetrator who were female
sex_count_F = collection0.count_documents({"PERP_SEX": "F"})
print("total count of perpetrator who were female")
print_(sex_count_F)
```

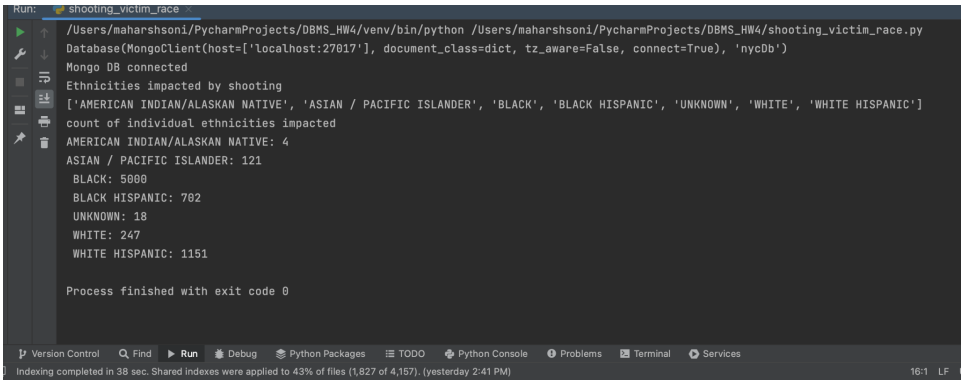
```
/Users/maharshsoni/PycharmProjects/DBMS_HW4/venv/bin/python /Users/maharshsoni/PycharmProjects/DBMS_HW4/shooting_count_by_perp_sex.py
Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'nycDb')
Mongo DB connected
total count of perpetrators who were male
6458
total count of perpetrator who were female
194

Process finished with exit code 0
```

Query3: GroupBy count of Victims by Ethnicity.



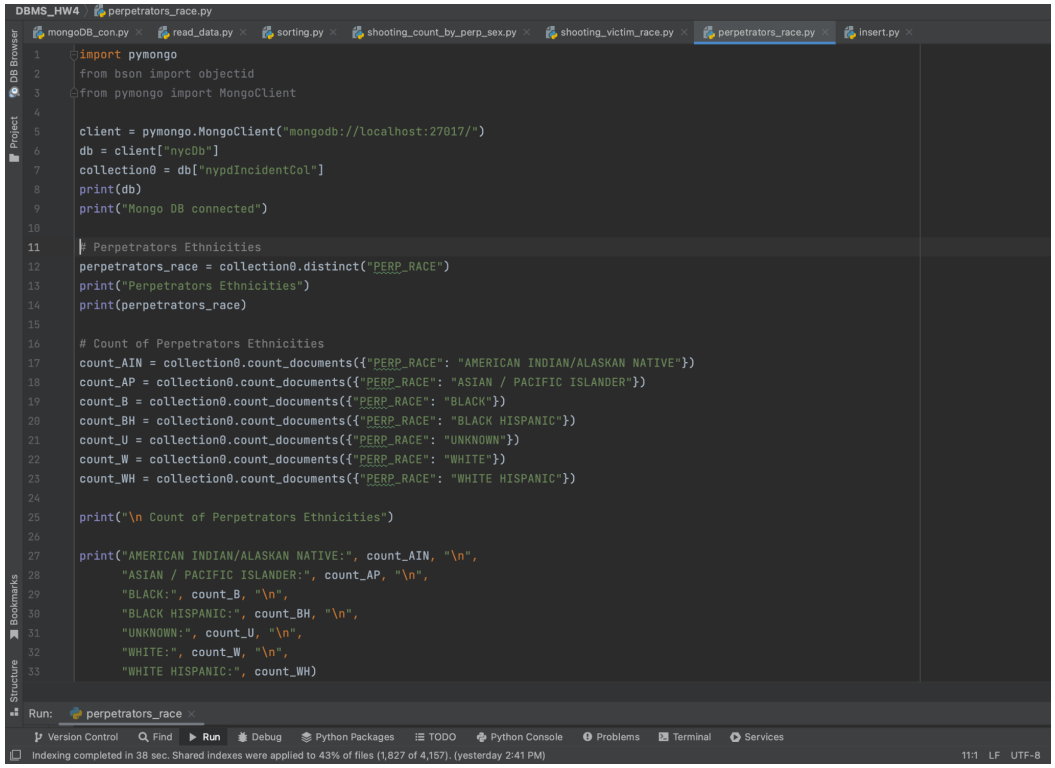
```
import pymongo
from bson import objectid
from pymongo import MongoClient

client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["nycDb"]
collection0 = db["nypdIncidentCol"]
print(db)
print("Mongo DB connected")


# Ethnicities impacted by shooting
victim_race = collection0.distinct("VIC_RACE")
print("Ethnicities impacted by shooting")
print(victim_race)

# count of individual ethnicities impacted
count_AIN = collection0.count_documents({"VIC_RACE": "AMERICAN INDIAN/ALASKAN NATIVE"})
count_AP = collection0.count_documents({"VIC_RACE": "ASIAN / PACIFIC ISLANDER"})
count_B = collection0.count_documents({"VIC_RACE": "BLACK"})
count_BH = collection0.count_documents({"VIC_RACE": "BLACK HISPANIC"})
count_U = collection0.count_documents({"VIC_RACE": "UNKNOWN"})
count_W = collection0.count_documents({"VIC_RACE": "WHITE"})
count_WH = collection0.count_documents({"VIC_RACE": "WHITE HISPANIC"})

print("count of individual ethnicities impacted")

print("AMERICAN INDIAN/ALASKAN NATIVE:", count_AIN)
print("ASIAN / PACIFIC ISLANDER:", count_AP, "\n",
      "BLACK:", count_B, "\n",
      "BLACK HISPANIC:", count_BH, "\n",
      "UNKNOWN:", count_U, "\n",
      "WHITE:", count_W, "\n",
      "WHITE HISPANIC:", count_WH)
```

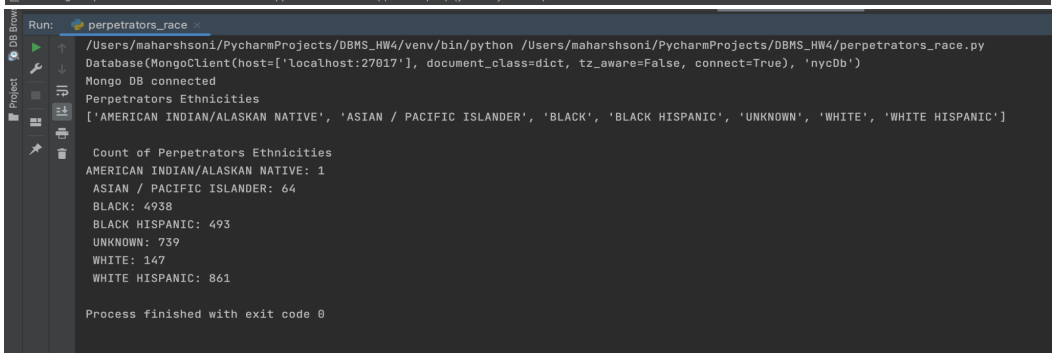
Result: A significant proportion of victims includes BLACK followed by WHITE HISPANIC



Query4: GroupBy count of Perpetrators by Ethnicity.

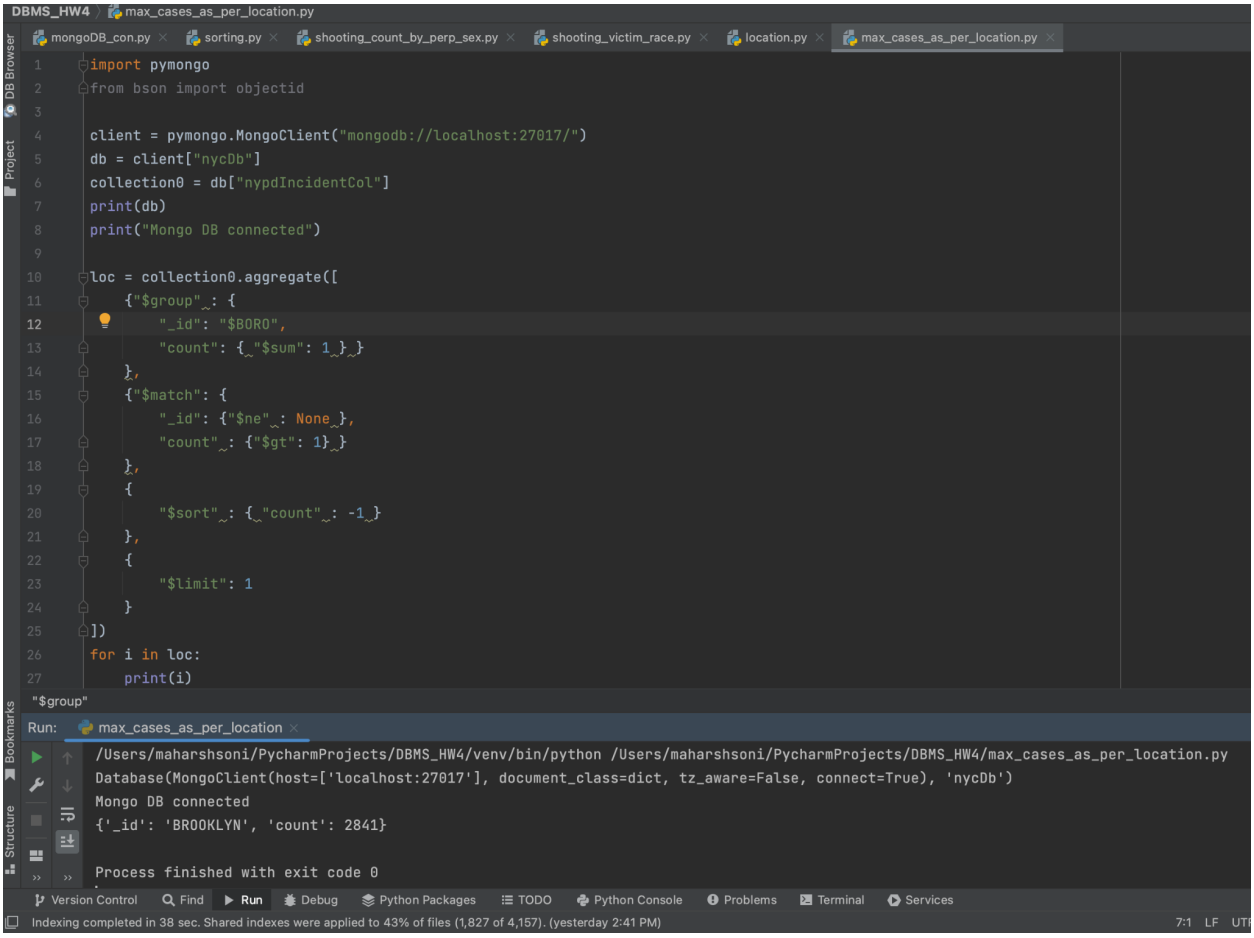
Result- A major proportion of perpetrators are BLACKs followed by WHITE Hispanics





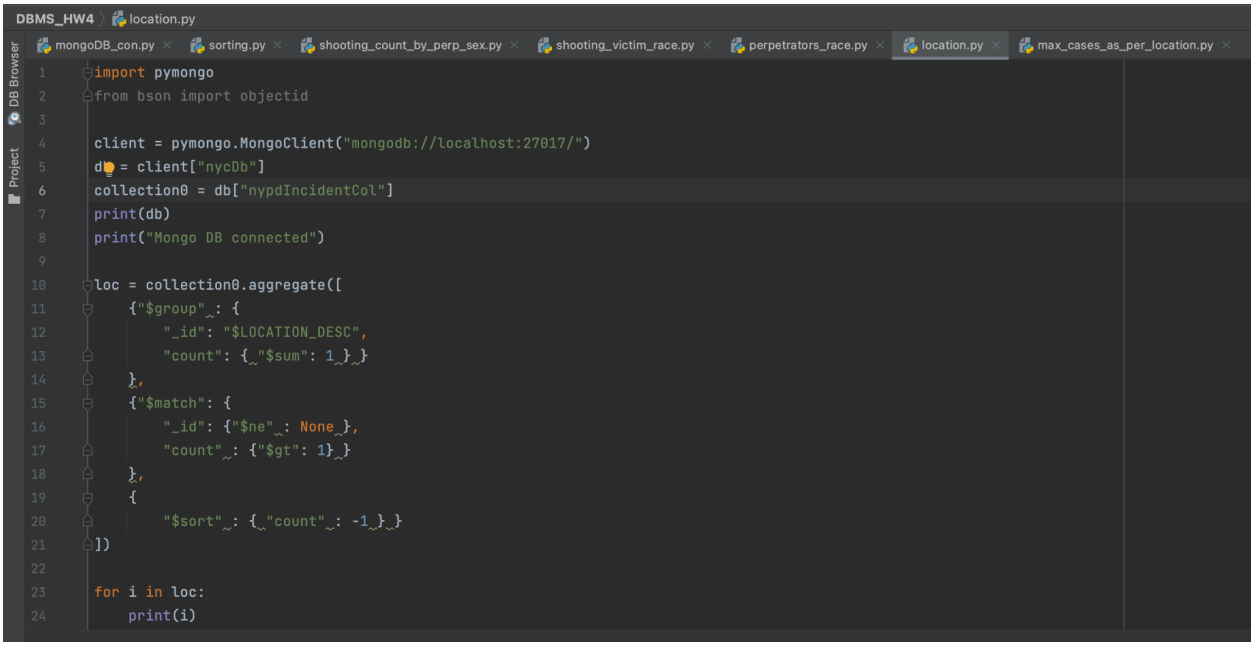
Query-5: Return the Borough Name with the highest number of shooting incidents recorded

Result- Brooklyn has the highest number of incidents in NYC from 2006 to 2021.

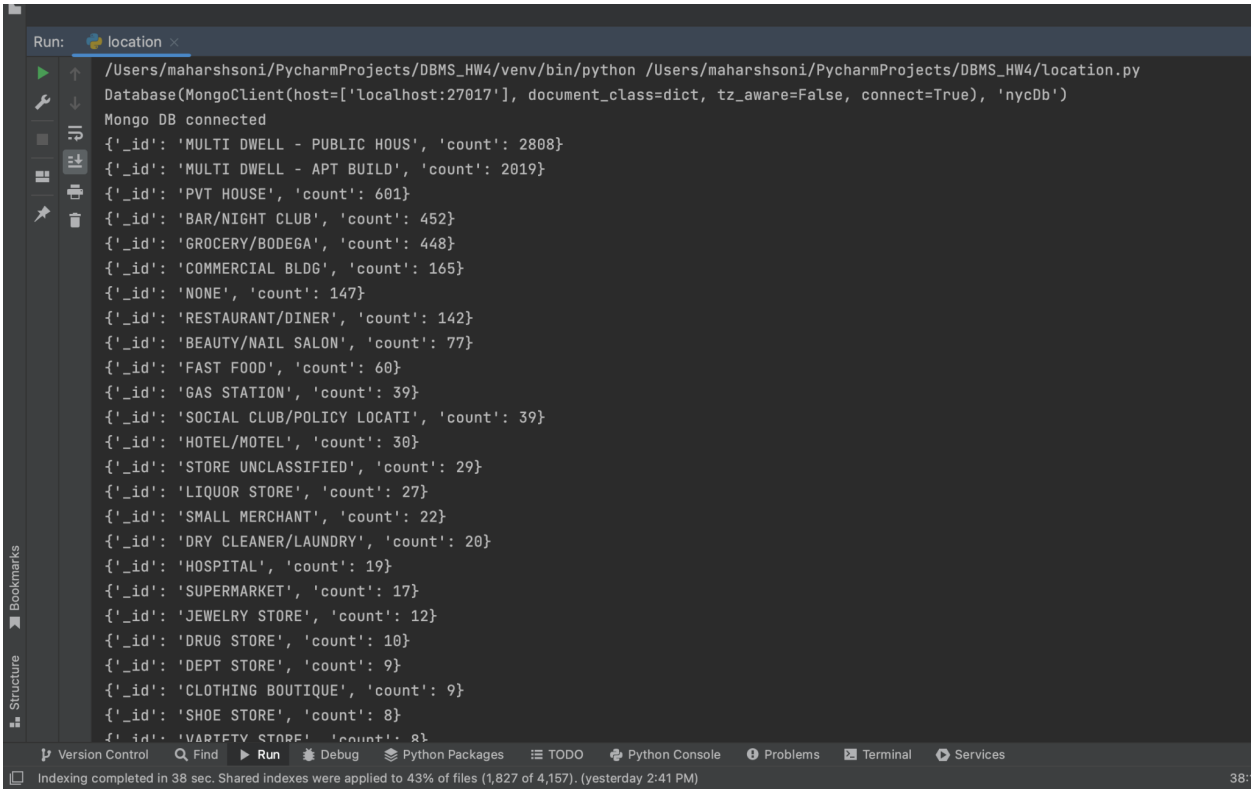


Query-6: GroupBy count of shooting incidents by Location Description

Result- Multi-Dwelling Locations (includes sub-classes) account for more than 50% of the shooting incidents recorded.



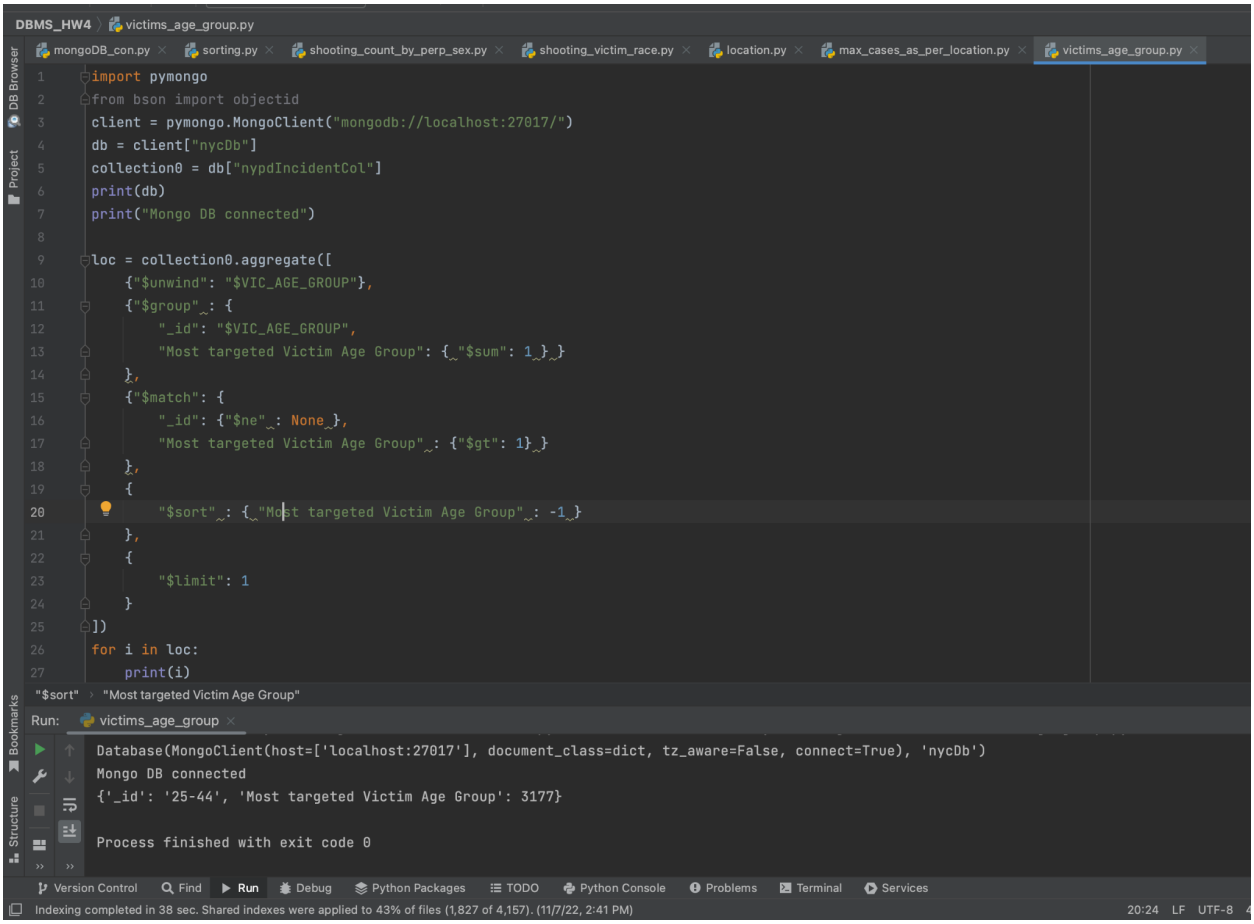
```
import pymongo
from bson import objectid

client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["nycDb"]
collection0 = db["nypdIncidentCol"]
print(db)
print("Mongo DB connected")

loc = collection0.aggregate([
    {"$group": {
        "_id": "$LOCATION_DESC",
        "count": {"$sum": 1}}
    },
    {"$match": {
        "_id": {"$ne": None},
        "count": {"$gt": 1}}
    },
    {
        "$sort": {"count": -1}}
])

for i in loc:
    print(i)
```



```
/Users/maharshsoni/PycharmProjects/DBMS_HW4/venv/bin/python /Users/maharshsoni/PycharmProjects/DBMS_HW4/location.py
Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'nycDb')
Mongo DB connected
{'_id': 'MULTI DWELL - PUBLIC HOUS', 'count': 2808}
{'_id': 'MULTI DWELL - APT BUILD', 'count': 2019}
{'_id': 'PVT HOUSE', 'count': 601}
{'_id': 'BAR/NIGHT CLUB', 'count': 452}
{'_id': 'GROCERY/BODEGA', 'count': 448}
{'_id': 'COMMERCIAL BLDG', 'count': 165}
{'_id': 'NONE', 'count': 147}
{'_id': 'RESTAURANT/DINER', 'count': 142}
{'_id': 'BEAUTY/NAIL SALON', 'count': 77}
{'_id': 'FAST FOOD', 'count': 60}
{'_id': 'GAS STATION', 'count': 39}
{'_id': 'SOCIAL CLUB/POLICY LOCATI', 'count': 39}
{'_id': 'HOTEL/MOTEL', 'count': 30}
{'_id': 'STORE UNCLASSIFIED', 'count': 29}
{'_id': 'LIQUOR STORE', 'count': 27}
{'_id': 'SMALL MERCHANT', 'count': 22}
{'_id': 'DRY CLEANER/LAUNDRY', 'count': 20}
{'_id': 'HOSPITAL', 'count': 19}
{'_id': 'SUPERMARKET', 'count': 17}
{'_id': 'JEWELRY STORE', 'count': 12}
{'_id': 'DRUG STORE', 'count': 10}
{'_id': 'DEPT STORE', 'count': 9}
{'_id': 'CLOTHING BOUTIQUE', 'count': 9}
{'_id': 'SHOE STORE', 'count': 8}
```

Query-7: Victim's most-affected by Age-Group

Result- It appears that individuals aged 25-44 suffered the most casualties of shooting incidents in new york.



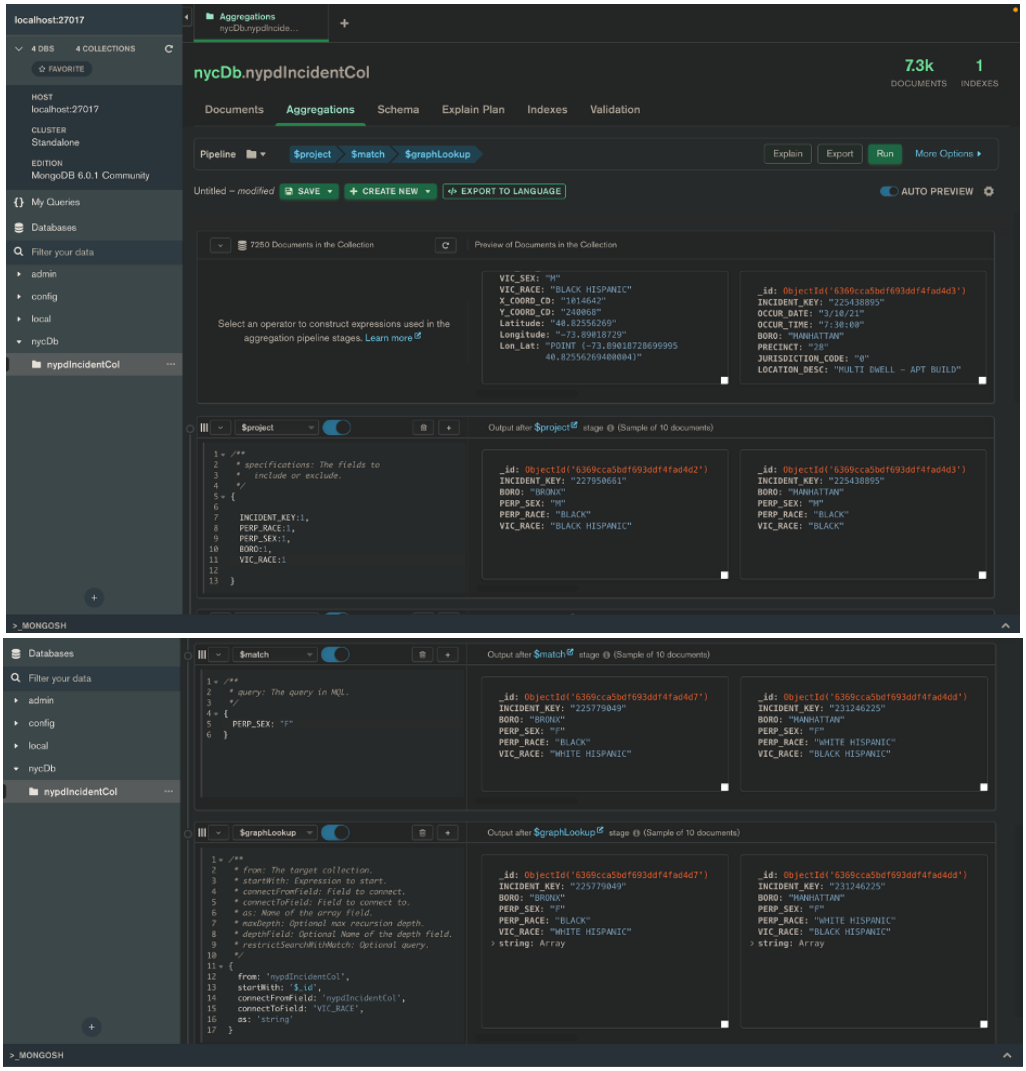
Aggregation_Pipelines:

We have used various aggregations in the pipeline on MongoDB Compass such as

1. **$project,**
2. **$match**,
3. **$graphLookup**

$project: With the $project function, only specific fields are passed onto the next stage whether they are from input documents or newly computed ones.
$match: Documents that match the specified conditions are only passed to the next pipeline stage by the $match operator.
$graphLookup: This aggregate searches a collection recursively, and gives users the option to specify the depth of recursion and the query filter.

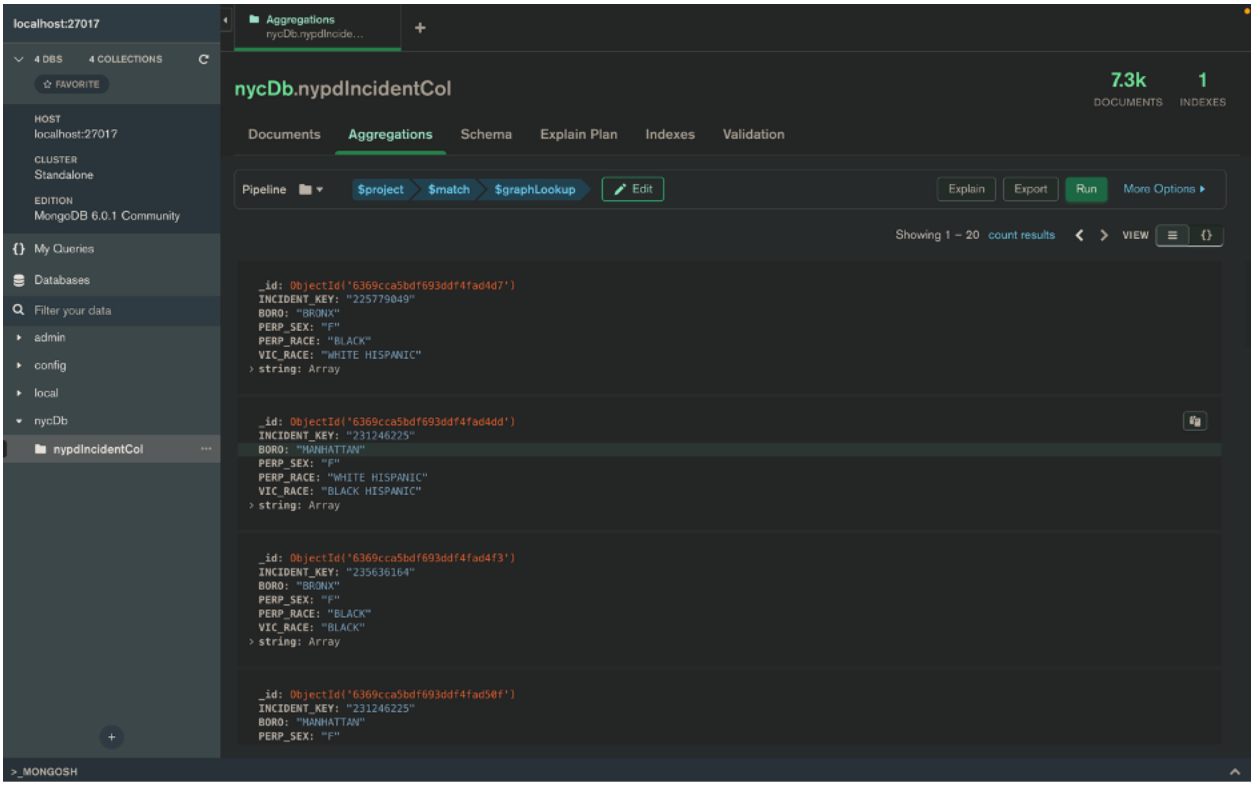
Query Screenshot:



Result Screenshot:

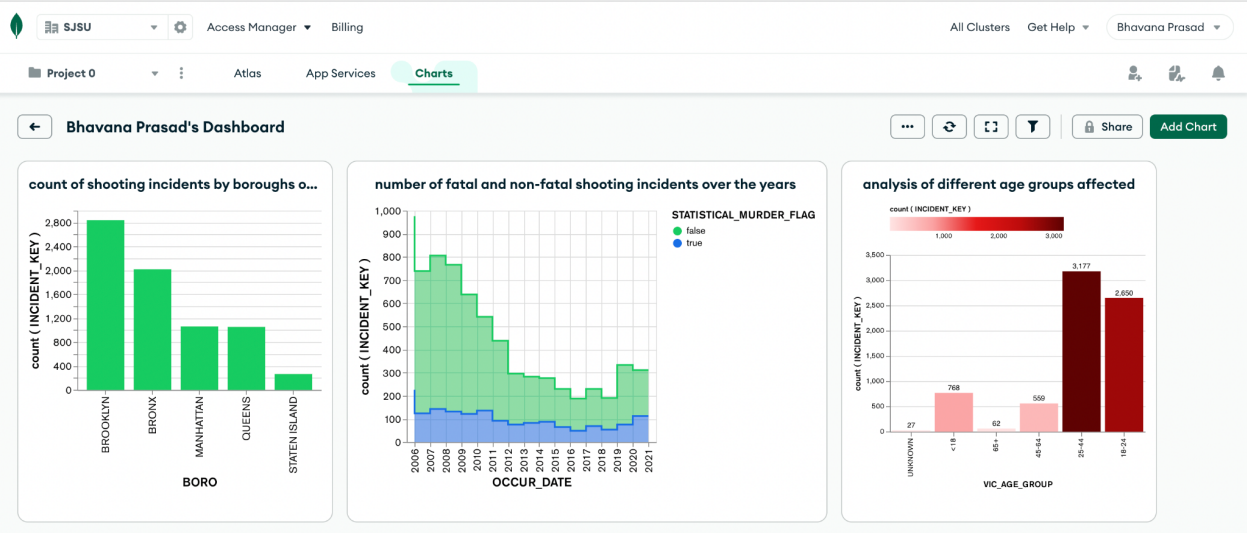
After clicking on the Run button, the aggregation pipeline gets executed and shows results like this. Users can export this result from MongoDB to local storage in CSV or JSON format.

Our aggregation pipeline gets all Perpetrators linked to an incident and location where their Gender is Female, along with their Racial Attributes.



Step-4: Data Visualization using MongoDB charts

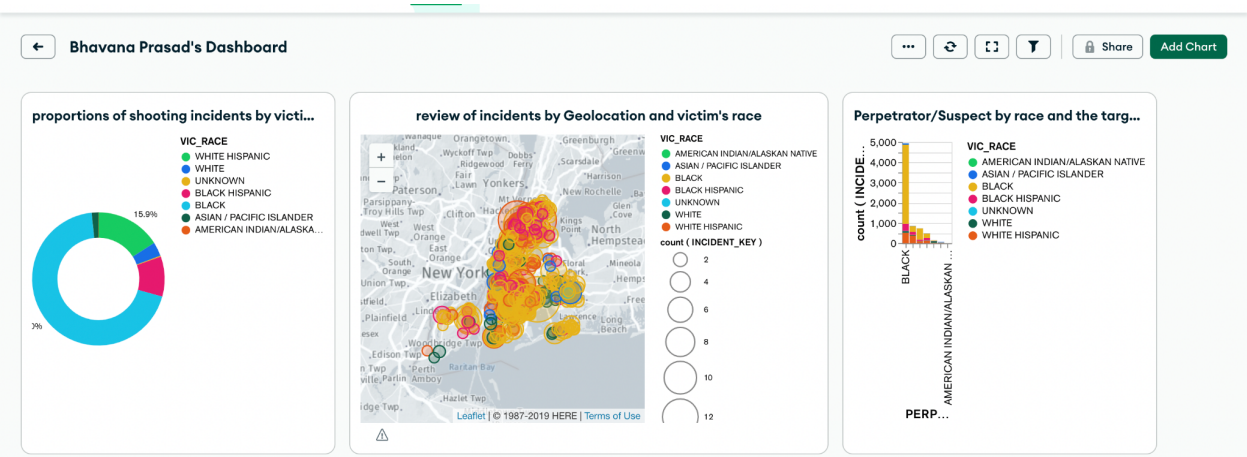
In order to gain further insights and identify an overall pattern, trends, and clusters, Visualization charts in MongoDB are used. Below are the charts and dashboards created by the visualization results.



- The first chart provides the count of shooting incidents per borough in New York, sorted in descending order. It's visible that the maximum number of incidents(approx 2800) occurred in the Brooklyn location. Staten Island has the least number of incidents that have taken place.

- The second chart is a trend analysis of shooting incidents over the years from 2006 to 2021, which is then categorized by the no of incidents that resulted in murders and just injuries. It is clearly visible that shooting incidents occur in large proportions during economic downturns such as the 2008-Big Recession and the 2020-COVID19 Pandemic.

- The third chart explains the statistics of different age groups of people affected. It is evident that the 25-44 and 18-24 age groups are more affected.



- To understand the nature of shooting crimes, the above visualization charts are used. The left graph shows the proportions of the race of people affected(victims) by the shooting. Proportions are higher in the Black race, about 69%, followed by white Hispanics, about 15.9%.

- Middle graph depicts the intensity of incidents by geolocation categorized by different races and ethnicities. The intensity of incidents among the black race is higher towards the south and spread across all regions. However, black Hispanics are more affected in the northern part of NY.

- Chart towards the right is a stacked bar graph that summarizes the proportion of people of different ethnicities targeted by perpetrators of each race. It shows that the perpetrators of the black race targeted more black people. This leaves us thinking of the possibilities of hate crimes within their own ethnicity.

- The above dashboard contains different visualization charts on the occurrence of incidents on time of the day, by month, and the ratio of murders and injuries in different location types(ex: public places/private/hospitals/grocery stores, etc.)

- Looking at the chart, Incidents occurred mostly during evenings and midnight between 7 PM and 4 AM.

- Seasonality impact: The number of shooting incidents is higher during the summer (May to October) and January.

H. NEO4J Solution

Step-1: Setup Local and Virtual ENVs to design Neo4j Solutions

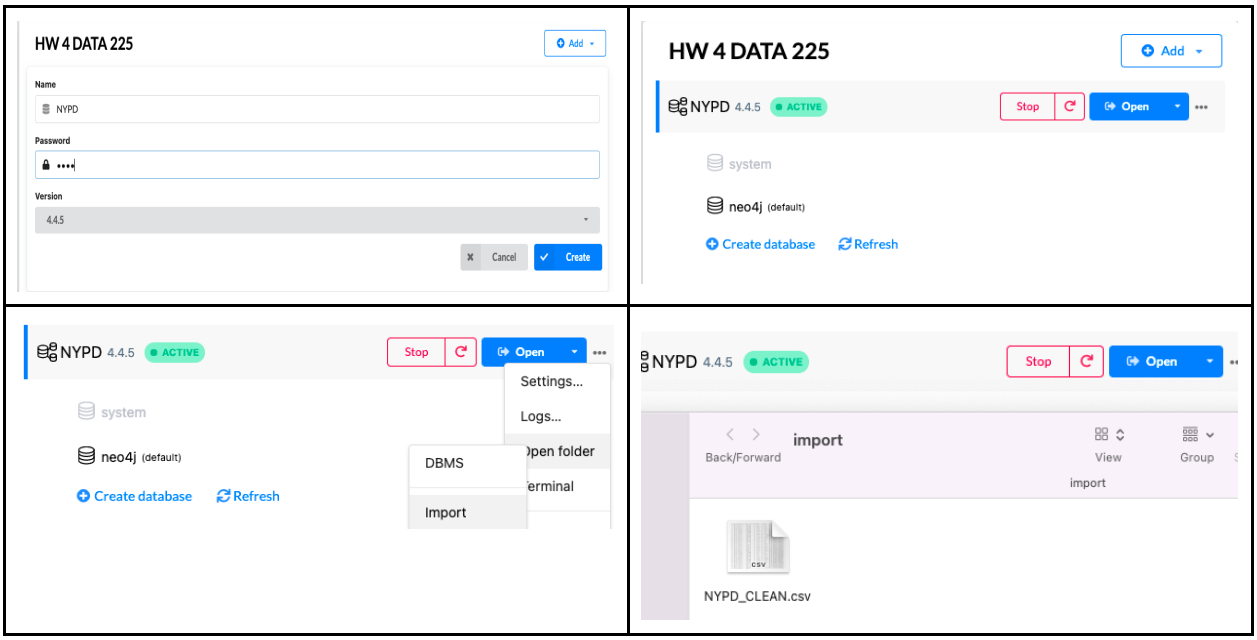
There are two ways to work with Neo4j.

1. Neo4J desktop (local environment)
2. Neo4J Aura DB (cloud environment)

Further, we can write Cypher Code or use any neo4j-connectors, such as py2neo, neo4j-spark-connector, etc., to create and query the database.

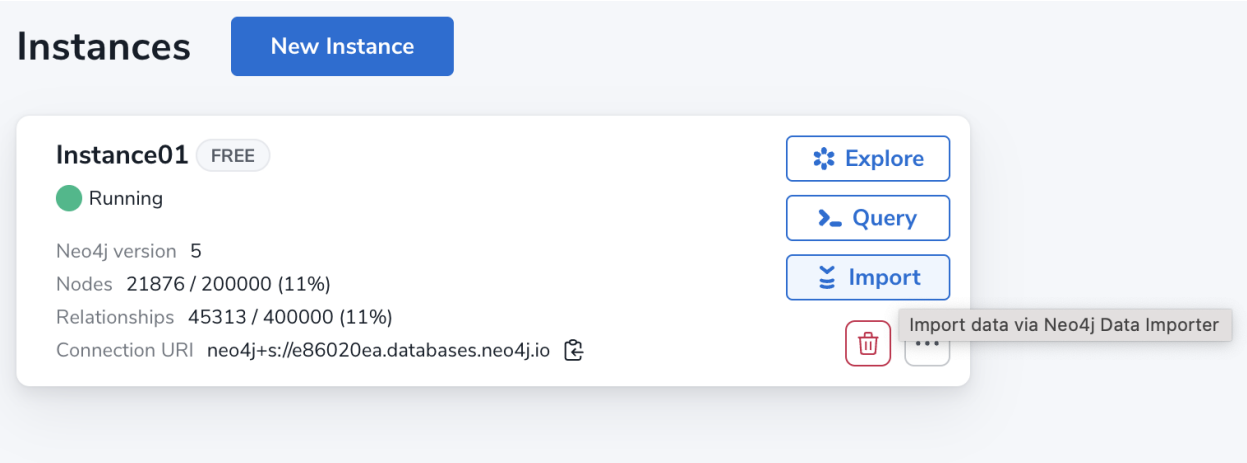
The Neo4J Aura DB solution makes it easy to import code with drag & drop functionalities. We take advantage of this solution but also use python's py2neo library and demonstrate the construction of nodes and relationships.

Local Environment Setup:



Cloud Environment Setup:

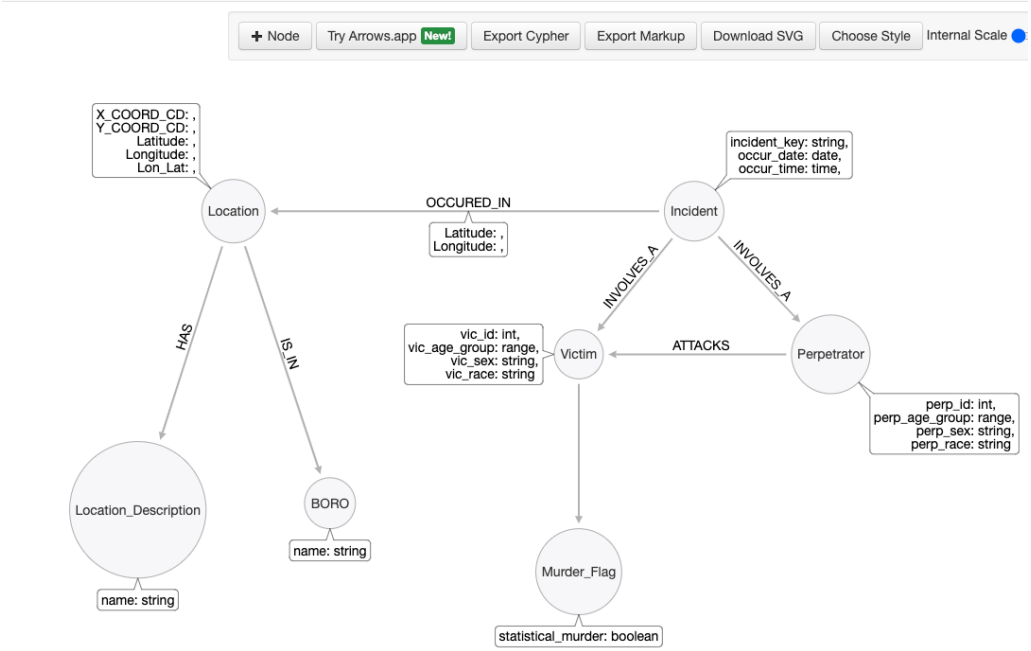
- Visit https://console.neo4j.io/?product=aura-db#databases
- Click on Create New Instance.
- Follow all default settings to proceed with the free tier
- Once the instance is Successfully setup, You will see a green dot
- Choose Import to import the data from the CSV file, and model the database
- Choose Query or Explore Options to Analyse the Database



Step-2: Database Design

We choose the Apcjones Arrow tool to model our data first and then we used all three alternatives Cypher Code, Py2Neo, and Import UI tool on Aura DB to demonstrate how to create the Nodes and Relationships, and design the Neo4J Graph Database solution. The screenshots in the sections below give you a walkthrough of how we took advantage of each approach to design, and populate the graph Database.

Data Model - Used Arrow Tool [http://www.apcjones.com/arrows/#]



a. Cypher Code Approach:

Create Nodes

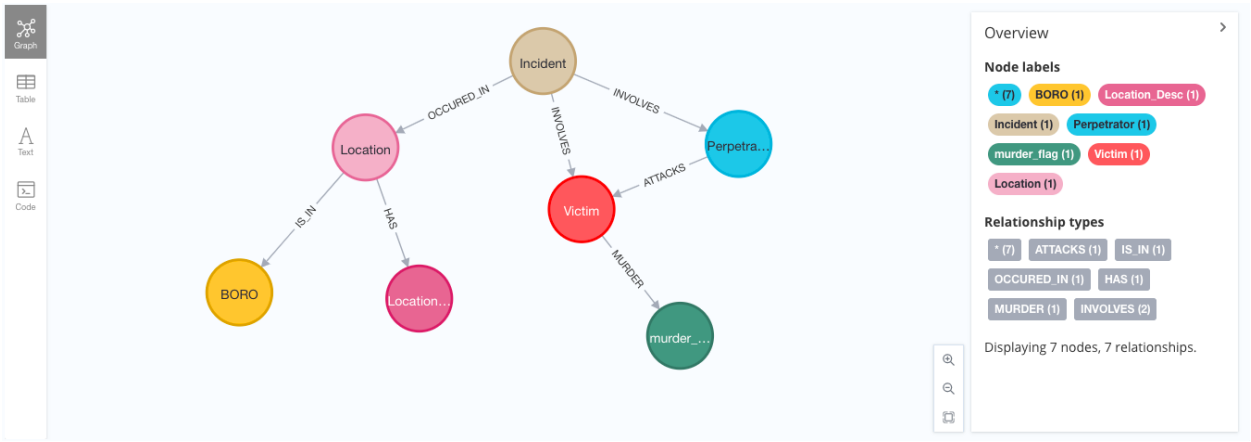
```
LOAD CSV WITH HEADERS FROM "file:///NYPD_CLEAN.csv" AS csvLine
CREATE (v:Victim {vic_id: csvLine.VICTIM_ID, vic_age_group: toString(csvLine.VIC_AGE_GROUP), vic_race: csvLine.VIC_RACE, vic_sex: csvLine.VIC_SEX})
CREATE (p:Perpetrator {perp_id: csvLine.PERP_ID, perp_age_group: toString(csvLine.PERP_AGE_GROUP), perp_race: csvLine.PERP_RACE, perp_sex: csvLine.PERP_SEX})
CREATE (i:Incident {incident_key: toString(csvLine.INCIDENT_KEY), occur_date: csvLine.OCCUR_DATE, occur_time: csvLine.OCCUR_TIME, mass_shooting: csvLine.MASS_SHOOTING})
CREATE (l:Location {x_coord: csvLine.X_COORD_CD, y_coord: csvLine.Y_COORD_CD, latitude: csvLine.Latitude, longitude: csvLine.Longitude})
CREATE (ld:Location_Desc {loc_desc: csvLine.LOCATION_DESC})
CREATE (boro:BORO {name: csvLine.BORO})
CREATE (mf:murder_flag {stat_murder_flag: csvLine.STATISTICAL_MURDER_FLAG})
```

Create Relations

```
MERGE (i)-[ip:INVOLVES]→(p)
MERGE (i)-[iv:INVOLVES]→(v)
MERGE(p)-[pv:ATTACKS]→(v)
MERGE (v)-[m:MURDER]→(mf)
MERGE (i)-[o:OCCURED_IN {latitude: csvLine.Latitude, longitude: csvLine.Longitude}]→(l)
MERGE (l)-[in:IS_IN]→(boro)
MERGE (l)-[h:HAS]→(ld)
```

Added 50701 labels, created 50701 nodes, set 152103 properties, created 50701 relationships, completed after 6065 ms.

db.schema()



b. Py2Neo Python Connector Approach

Add Nodes

```
[ ] graph = Graph("neo4j+s://e86020ea.databases.neo4j.io", auth=("neo4j", "dZ1mOkN_w8AcF12WfUMyC_tHWB8lRSwyac5B5TTqsLo"))
    graph
```

```
Graph('neo4j+s://e86020ea.databases.neo4j.io:7687')
```

```
[ ]
    # Create nodes
    create_nodes(graph.auto(), dict_i, labels={"Incident"})
    print(graph.nodes.match("Incident").count())
```

```
5238
```

```
[ ] create_nodes(graph.auto(), dict_ms, labels={"MassShooting"})
    print(graph.nodes.match("MassShooting").count())
```

```
2
```

```
create_nodes(graph.auto(), dict_l, labels={"location"})
print(graph.nodes.match("location").count())

create_nodes(graph.auto(), dict_j, labels={"Jurisdiction"})
print(graph.nodes.match("Jurisdiction").count())

create_nodes(graph.auto(), dict_b, labels={"BORO"})
print(graph.nodes.match("BORO").count())

create_nodes(graph.auto(), dict_ld, labels={"LOCATION_DESC"})
print(graph.nodes.match("LOCATION_DESC").count())
```

```
3428
2
5
37
```

prepare relationships

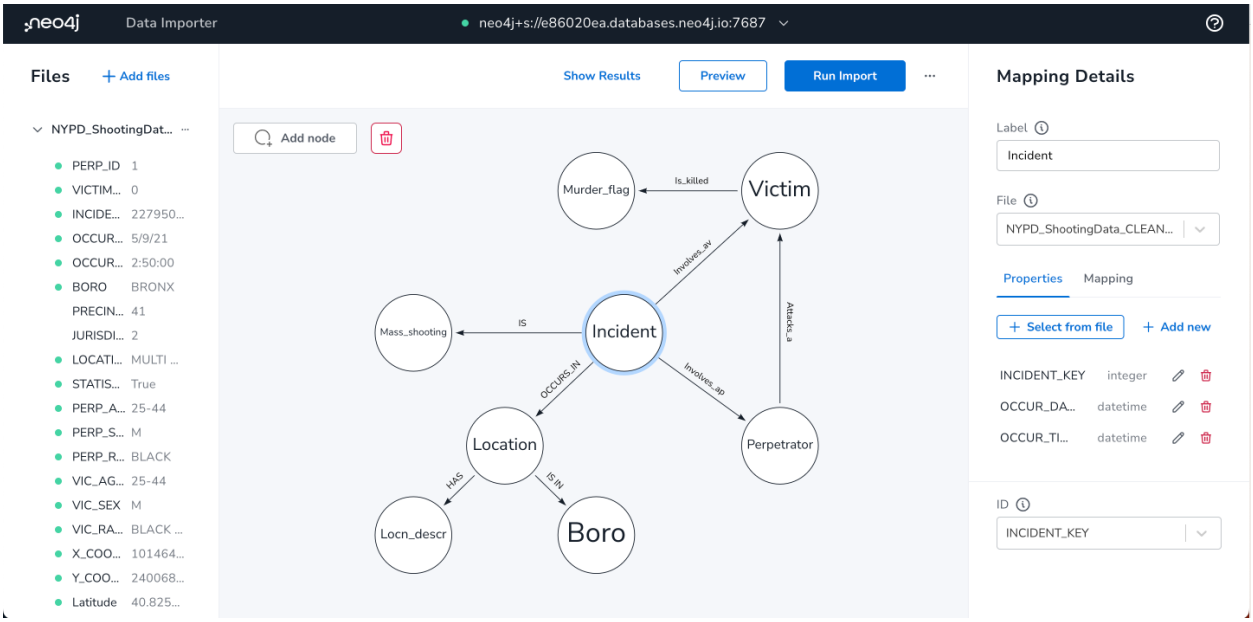
```
df_rel = df.filter(["INCIDENT_KEY", "VICTIM_ID"])
json_rel = df_rel.to_json(orient="records")
dict_rel_v = json.loads(json_rel)

#create relationships incidents involving victims
ex_victim = []

for p in dict_rel_v:
    incident= p["INCIDENT_KEY"]
    victim = p["VICTIM_ID"]
    ex_victim.append((incident,{'involves_v':1},victim))

print(ex_victim)
```

```
[(227950661, {'involves_v': 1}, 0), (225438895, {'involves_v': 1}, 1), (232193235, {'in
```

```
rel_iv = create_relationships(graph.auto(),
                              ex_victim,
                              "INVOLVES_V",
                              start_node_key=("Incident", "INCIDENT_KEY"),
                              end_node_key=("victim", "VICTIM_ID"))
print(rel_iv)
```

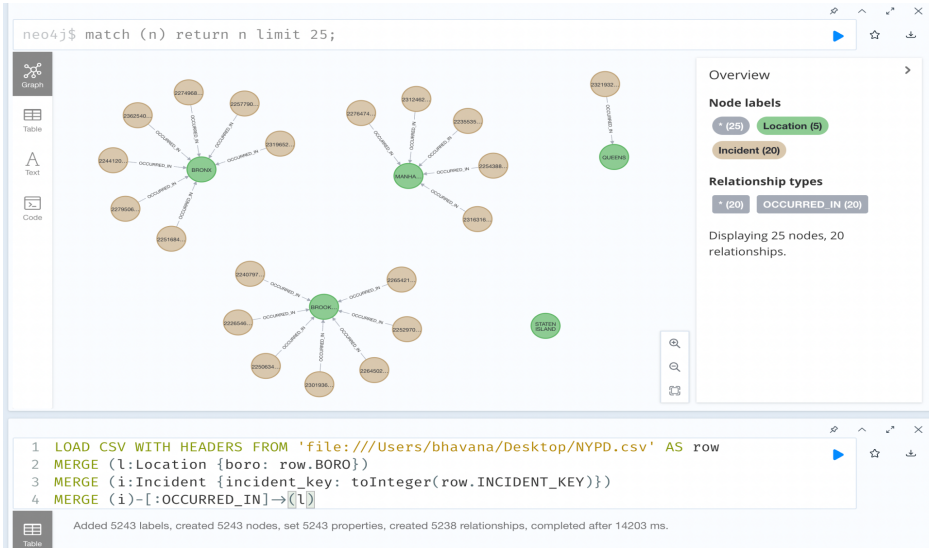
```
None
```

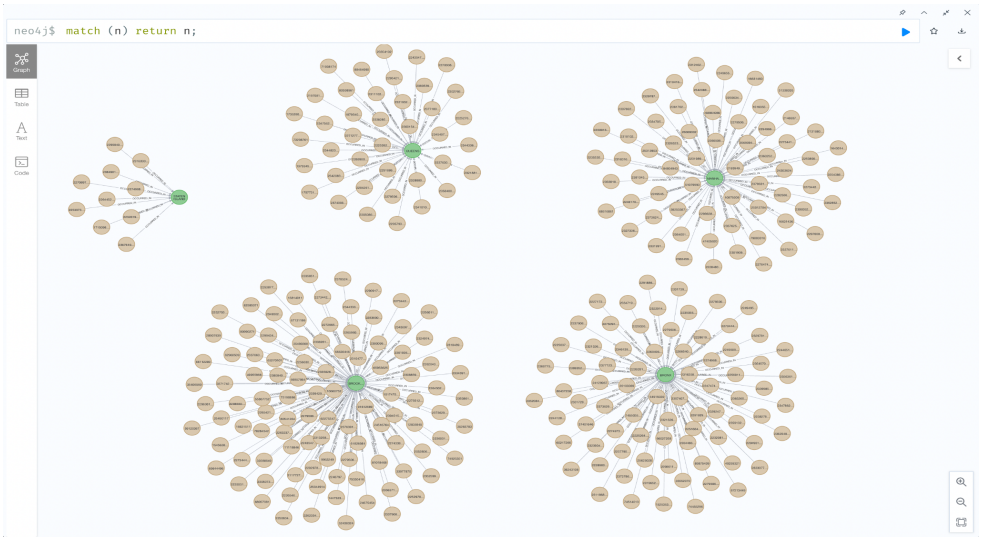
c. AURA DB UI approach



Step-3: Perform Query Analysis using Cypher, Py2Neo

Query-1: To identify the location in which maximum shooting incidents occurred in New York, matching the incidents and borough in which incidents took place.





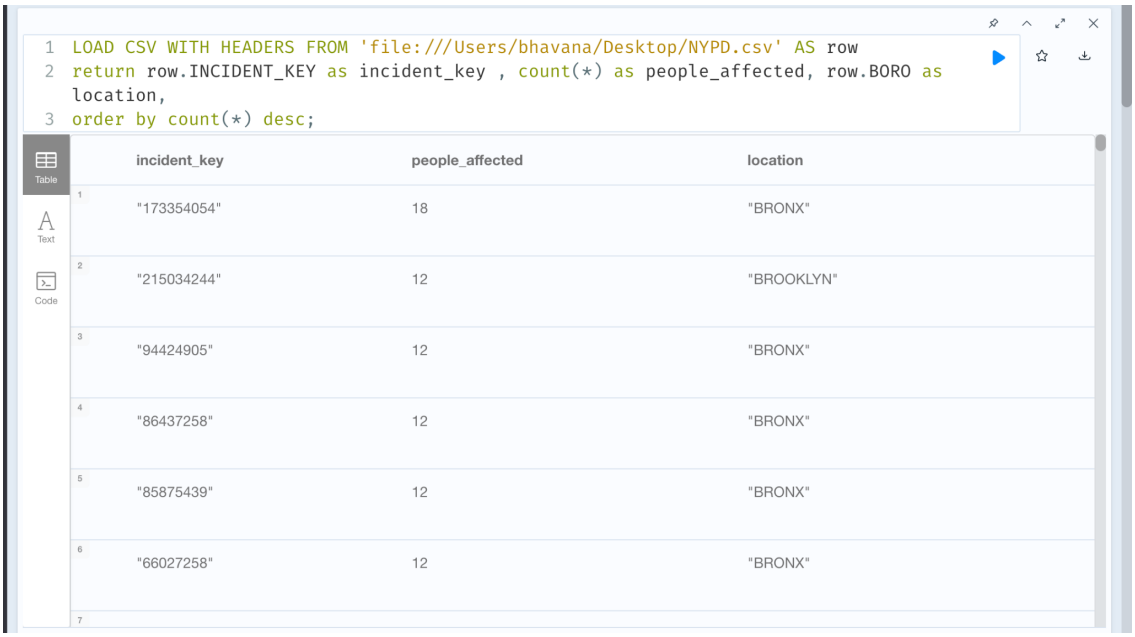
Result- We can see that most shooting incidents occurred in the **BROOKLYN** location in New York followed by **BRONX**

Query-2: Finding (count) how many incidents occurred in each location (BORO) which is sorted in Descending order.

Result- In Query-1 we were able to visually interpret. In this query, we can associate them with a quantitative value and show the result in tabular form.

Query-3: Retrieving Mass Shooting Incidents and counting the number of people affected by mass shootings.

Result- If there is a mass shooting, we retrieved the incident_key, and location to show the number of people affected in that location by that mass shooting incident, and then we sort the result in descending order of people affected. We see that in BRONX 18 people were affected in an incident in the BRONX.

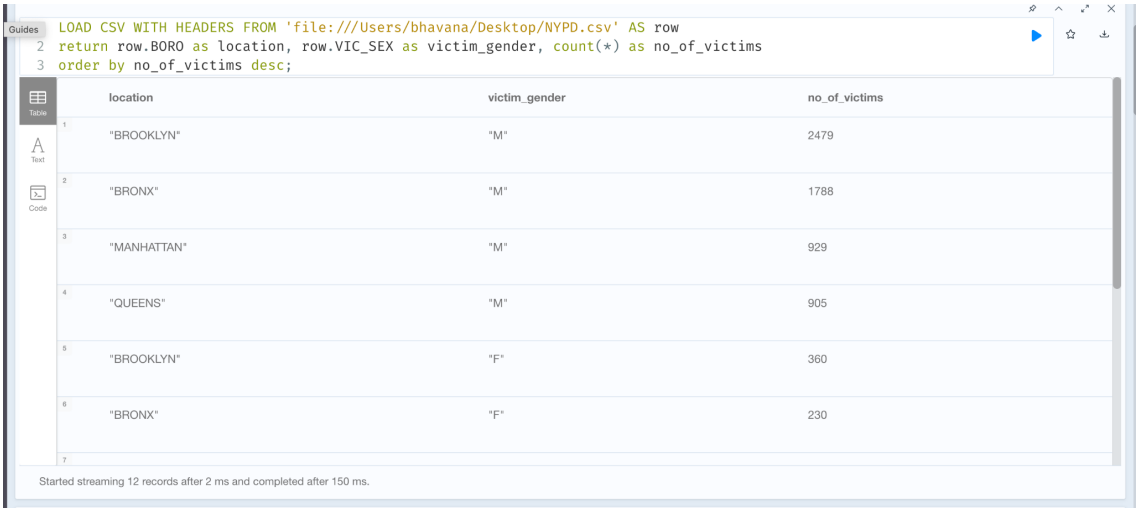


```
1 LOAD CSV WITH HEADERS FROM 'file:///Users/bhavana/Desktop/NYPD.csv' AS row
2 return row.INCIDENT_KEY as incident_key , count(*) as people_affected, row.BORO as
  location,
3 order by count(*) desc;
```

Table | Text | Code

	incident_key	people_affected	location
1	"173354054"	18	"BRONX"
2	"215034244"	12	"BROOKLYN"
3	"94424905"	12	"BRONX"
4	"86437258"	12	"BRONX"
5	"85875439"	12	"BRONX"
6	"66027258"	12	"BRONX"
7			

Query-4: Groupby Location, Gender, and count number of victims
Result- Brookly had the highest number of Female Victims as compared to the other 4 Boroughs.



Guides

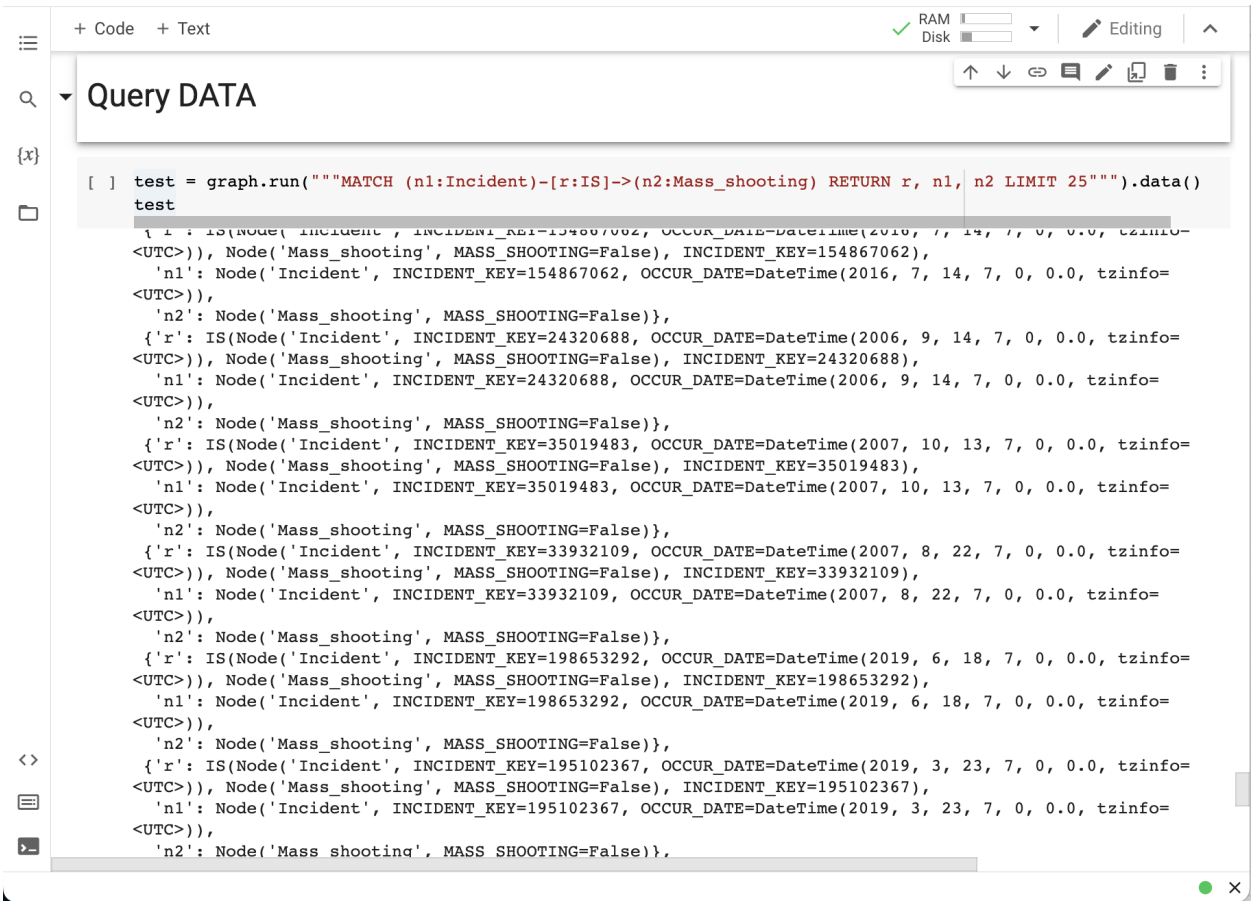
```
LOAD CSV WITH HEADERS FROM 'file:///Users/bhavana/Desktop/NYPD.csv' AS row
2 return row.BORO as location, row.VIC_SEX as victim_gender, count(*) as no_of_victims
3 order by no_of_victims desc;
```

Table | Text | Code

	location	victim_gender	no_of_victims
1	"BROOKLYN"	"M"	2479
2	"BRONX"	"M"	1788
3	"MANHATTAN"	"M"	929
4	"QUEENS"	"M"	905
5	"BROOKLYN"	"F"	360
6	"BRONX"	"F"	230
7			

Started streaming 12 records after 2 ms and completed after 150 ms.

Query-5: Sample Query to understand how to Query Neo4J with Py2Neo Python Module



Query-6: Different age groups affected along with the count

Result - Age groups 25 to 44 were the most attacked in the victim population.

```
neo4j@neo4j> LOAD CSV WITH HEADERS FROM "file:///NYPD_CLEAN.csv" AS row
             return row.VIC_AGE_GROUP, count(*) AS AGE_GROUP
             order by count(*) desc;

+-------------------------------+
| row.VIC_AGE_GROUP | AGE_GROUP |
+-------------------------------+
| "25-44"           | 3177      |
| "18-24"           | 2650      |
| "<18"             | 768       |
| "45-64"           | 559       |
| "65+"             | 62        |
| "UNKNOWN"         | 27        |
+-------------------------------+
```

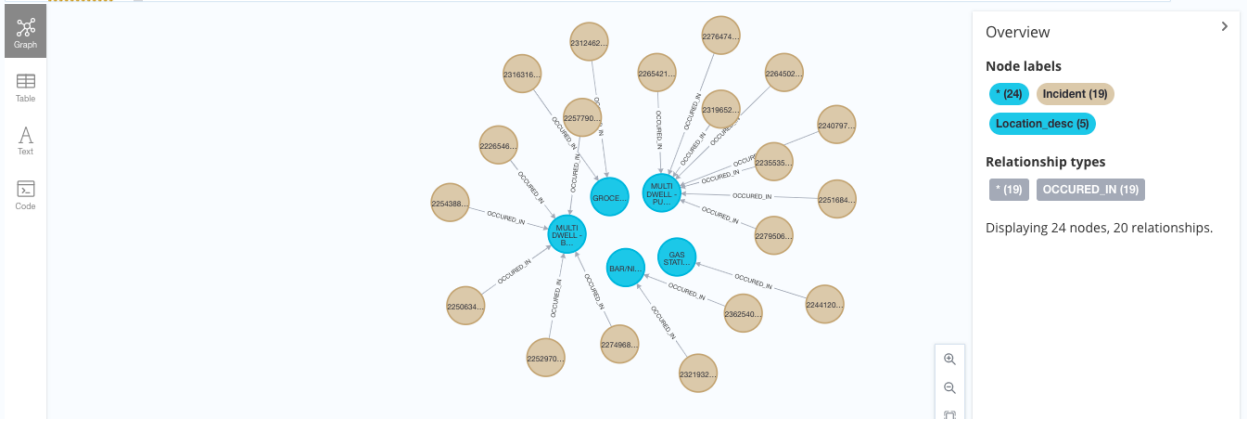
Query-7: People affected by different location types in mass shootings.

```
LOAD CSV WITH HEADERS FROM "file:///NYPD_CLEAN.csv" AS row
return row.INCIDENT_KEY as incident_key , count(*) as people_affected, row.LOCATION_DESC as Place
order by count(*) desc;
```

	incident_key	people_affected	Place
1	"173354054"	18	"MULTI DWELL - APT BUILD"
2	"215034244"	12	"MULTI DWELL - PUBLIC HOUS"
3	"94424905"	12	"MULTI DWELL - APT BUILD"
4	"86437258"	12	"MULTI DWELL - PUBLIC HOUS"
5	"85875439"	12	"MULTI DWELL - PUBLIC HOUS"
6	"66027258"	12	"MULTI DWELL - PUBLIC HOUS"
7			

Started streaming 5238 records after 10 ms and completed after 11 ms, displaying first 1000 rows.

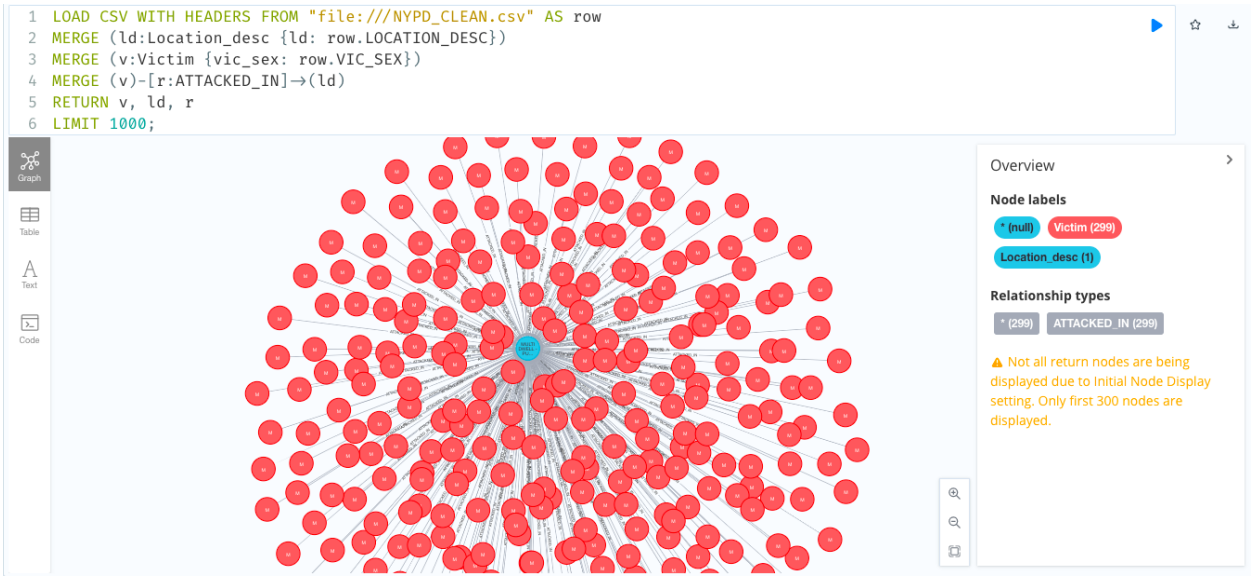
```
LOAD CSV WITH HEADERS FROM "file:///NYPD_CLEAN.csv" AS row
MERGE (ld:Location_desc {ld: row.LOCATION_DESC})
MERGE (i:Incident {incident_key: toInteger(row.INCIDENT_KEY)})
MERGE (i)-[r:OCCURED_IN]->(ld)
RETURN i, ld, r
LIMIT 20;
```



Query-8: People affected categorized by location type (place) and gender

```
neo4j@neo4j> LOAD CSV WITH HEADERS FROM "file:///NYPD_CLEAN.csv" AS row
             RETURN row.LOCATION_DESC AS Place, row.VIC_SEX, count(*) AS NUMBER_OF_PE
OPLE
             ORDER BY row.LOCATION_DESC desc
             LIMIT 20;
+-------------------------------------------------------------+
| Place                        | row.VIC_SEX | NUMBER_OF_PEOPLE |
+-------------------------------------------------------------+
| "VIDEO STORE"                | "M"         | 1                |
| "VARIETY STORE"              | "M"         | 8                |
| "TELECOMM. STORE"            | "M"         | 5                |
| "SUPERMARKET"                | "F"         | 3                |
| "SUPERMARKET"                | "M"         | 14               |
| "STORE UNCLASSIFIED"         | "F"         | 2                |
| "STORE UNCLASSIFIED"         | "M"         | 27               |
| "STORAGE FACILITY"           | "M"         | 1                |
| "SOCIAL CLUB/POLICY LOCATI"  | "F"         | 3                |
| "SOCIAL CLUB/POLICY LOCATI"  | "M"         | 36               |
| "SMALL MERCHANT"             | "M"         | 22               |
| "SHOE STORE"                 | "M"         | 8                |
| "RESTAURANT/DINER"           | "M"         | 125              |
| "RESTAURANT/DINER"           | "F"         | 17               |
| "PVT HOUSE"                  | "M"         | 477              |
| "PVT HOUSE"                  | "F"         | 124              |
| "PHOTO/COPY STORE"           | "M"         | 1                |
| "NONE"                       | "M"         | 143              |
| "NONE"                       | "F"         | 4                |
| "MULTI DWELL - PUBLIC HOUS"  | "M"         | 2495             |
+-------------------------------------------------------------+
```

Query-9: Majority gender affected in MULTI-DWELL PUBLIC HOUSES which is MALE

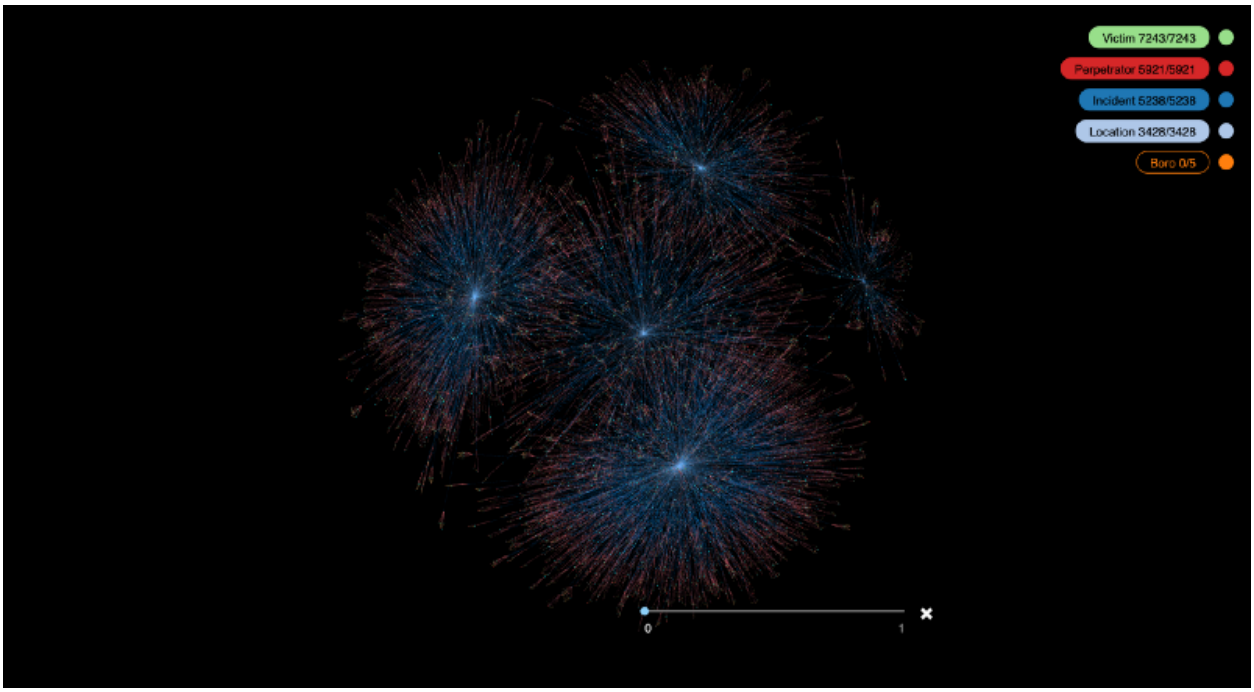


Step-4: Perform Visual Analysis using GraphXR

We will connect the Neo4J database with GraphXR, a powerful Graph Visualization tool that connects to your hosted Aura DB graph database to generate excellent visual insights. We had the limitation of 1000 nodes associated with the Neo4J Aura DB's Bloom Visual tool as a compelling reason to use GraphXR.

Visual Information-1: How many Boroughs are there in New York, and how spread are they?

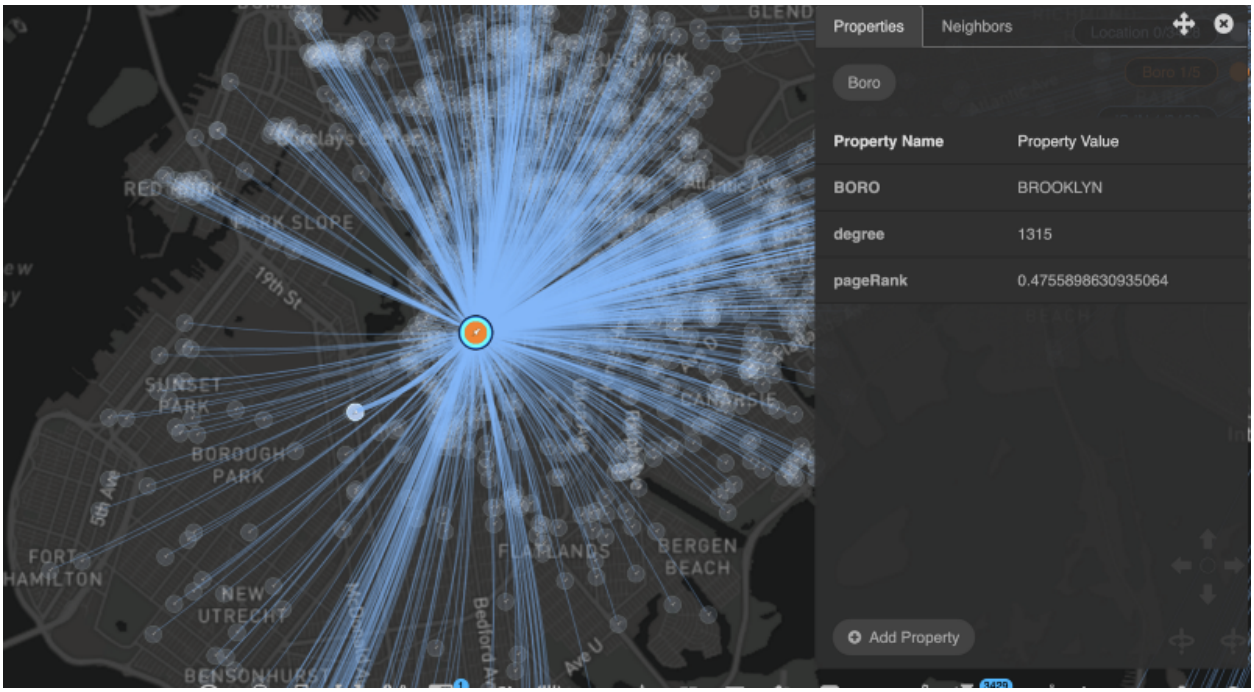
BOROs connected to Location Co-ordinates on a Map Background

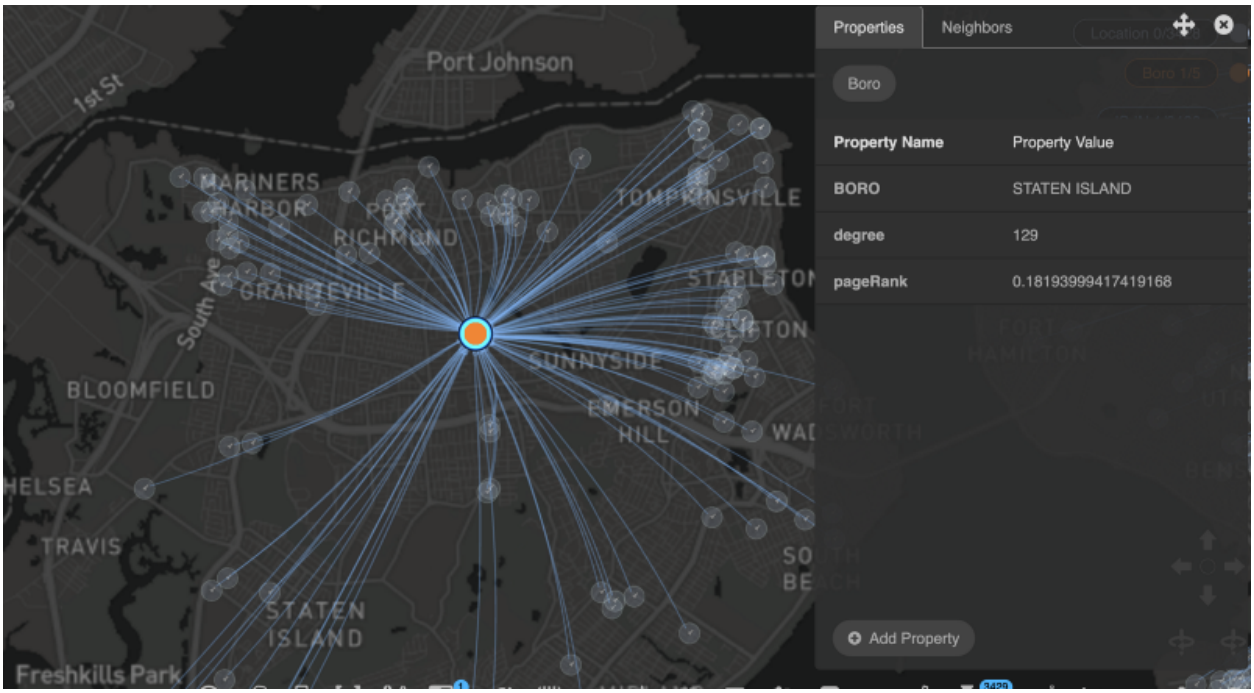


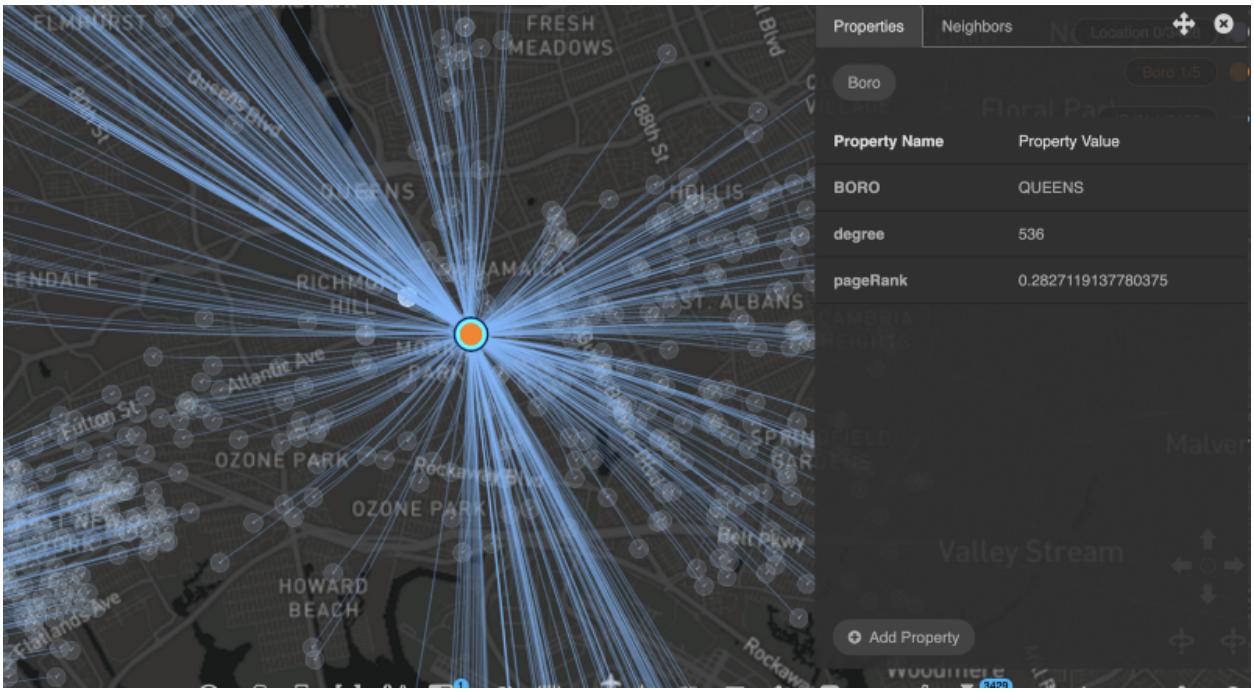
BOROs connected to all other entities w/o Map background

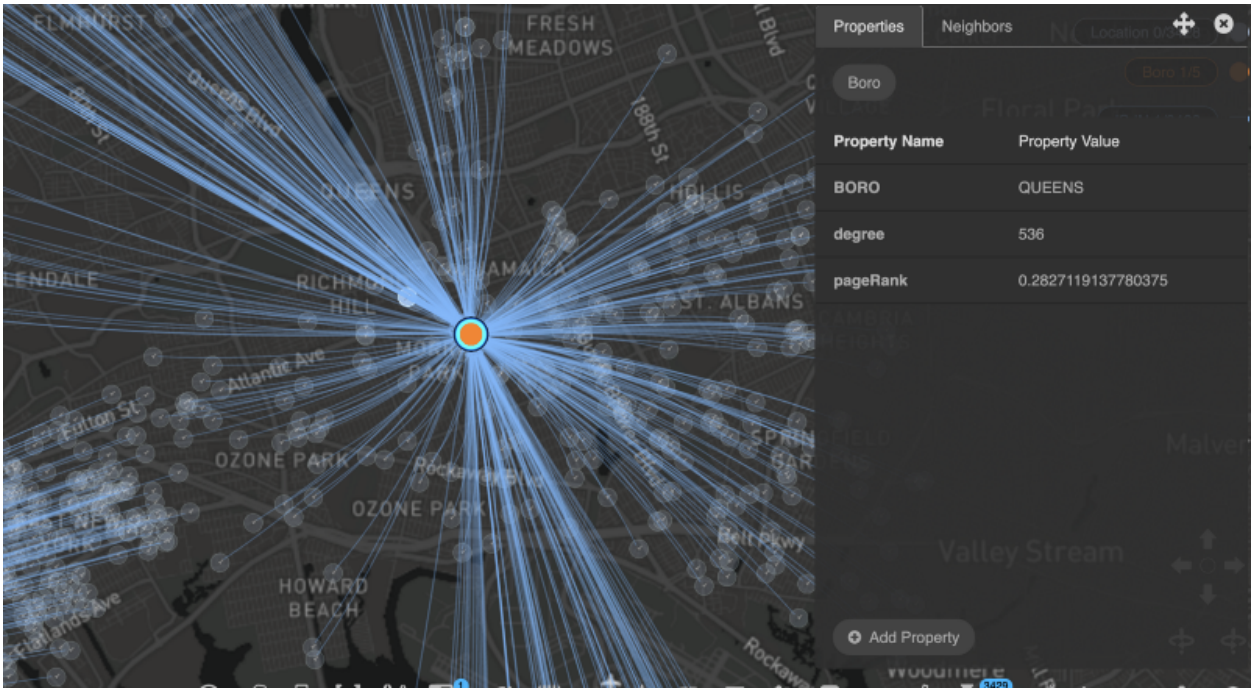
Now we can see that there are 5 Boroughs and how widely spread these Boroughs are. While getting the unique boroughs with pd['col'].value_counts() is more straightforward, the wealth of information this graph offers, including how widespread these boroughs are, can help us define more efficient data-driven interventions/solutions for NYPD.

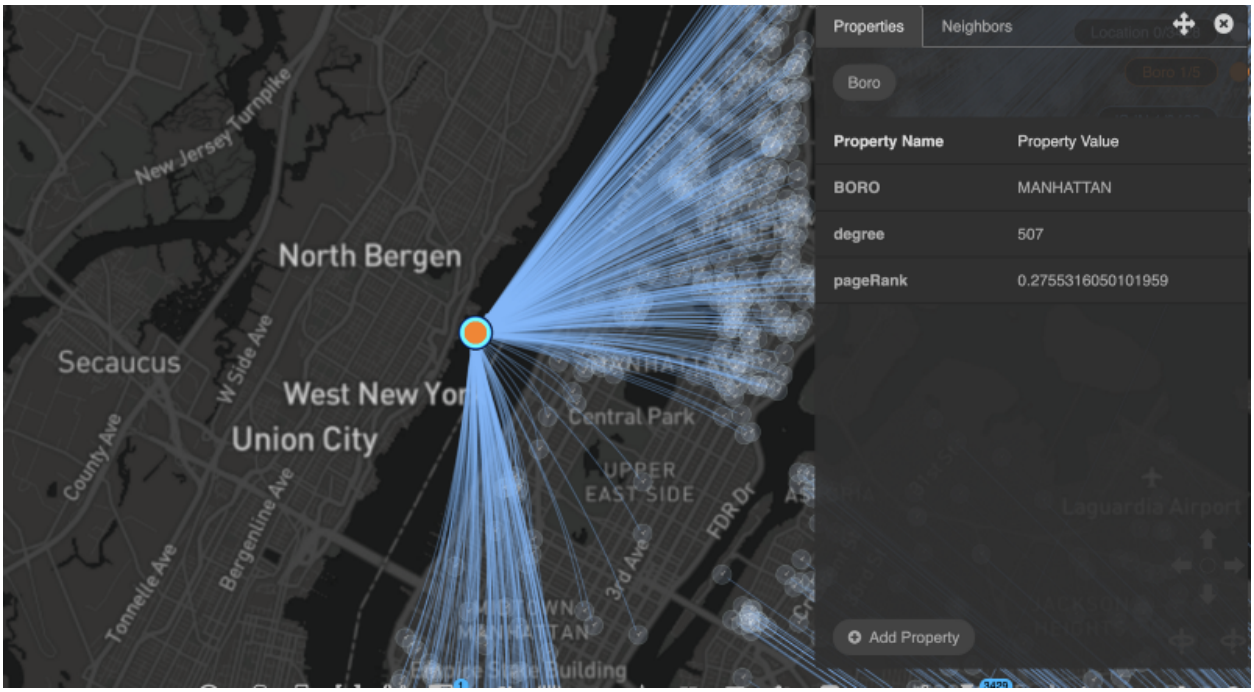
We observe that Queens is more widespread than Staten Island. Further, we can apply the **Centrality and community detection algorithms** to find the actual values. Take a look at the screenshots below. We find that **Brooklyn has the highest degree and page_rank as compared to all other boroughs**. Hence, it is safe to say that **Brooklyn requires ten times more policing than Staten Island based on the degree of relationship.**


Properties
Neighbors
Boro
Property Name
Property Value
BORO
BROOKLYN
degree
1315
pageRank
0.4755898630935064
Add Property
RED HOOK
PARK SLOPE
SUNSET PARK
BOROUGH PARK
FORT HAMILTON
NEW UTRECHT
BENSONHURST
FLATLANDS
BERGEN BEACH
CANARSIE
19th St
5th Ave
Ave U

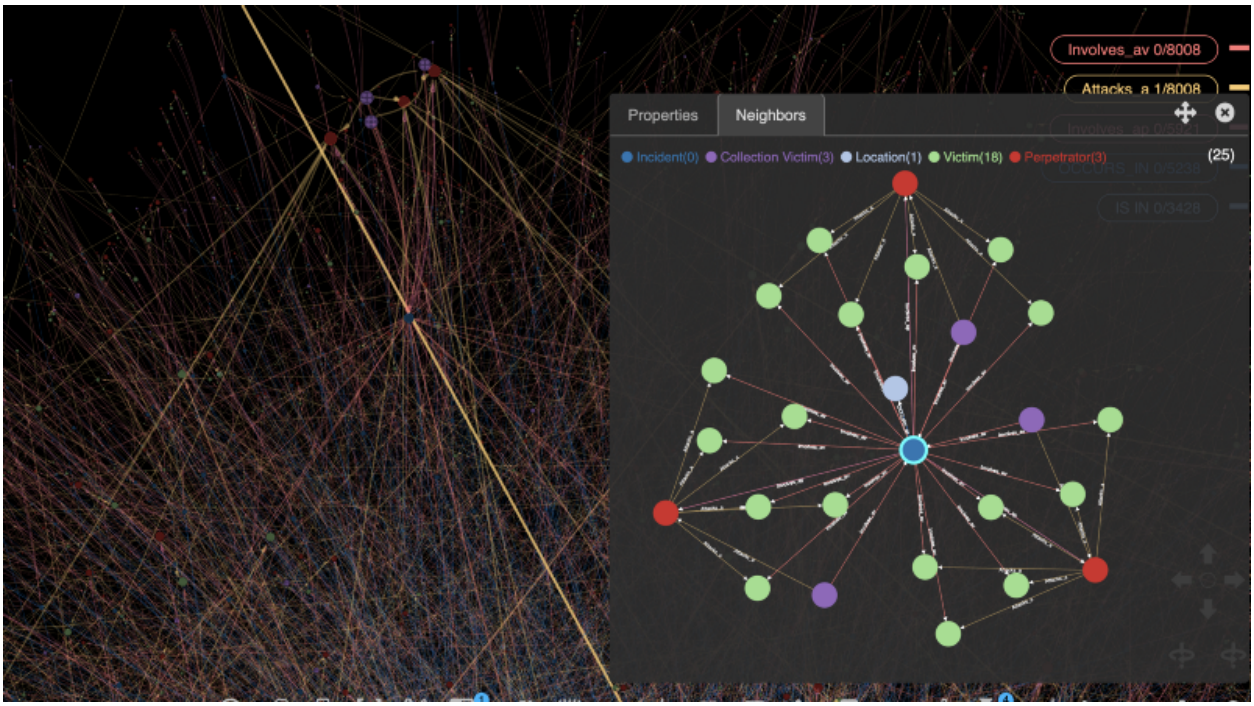

Properties
Neighbors
Boro
Property Name
Property Value
BORO
STATEN ISLAND
degree
129
pageRank
0.18193999417419168
Add Property
Port Johnson
1st St
MARINERS HARBOR
PORT RICHMOND
GRANDVILLE
TOMPKINSVILLE
STAPLETON
SUNNYSIDE
EMERSON HILL
BLOOMFIELD
HELSEA
TRAVIS
STATEN ISLAND
Freshkills Park
South Ave

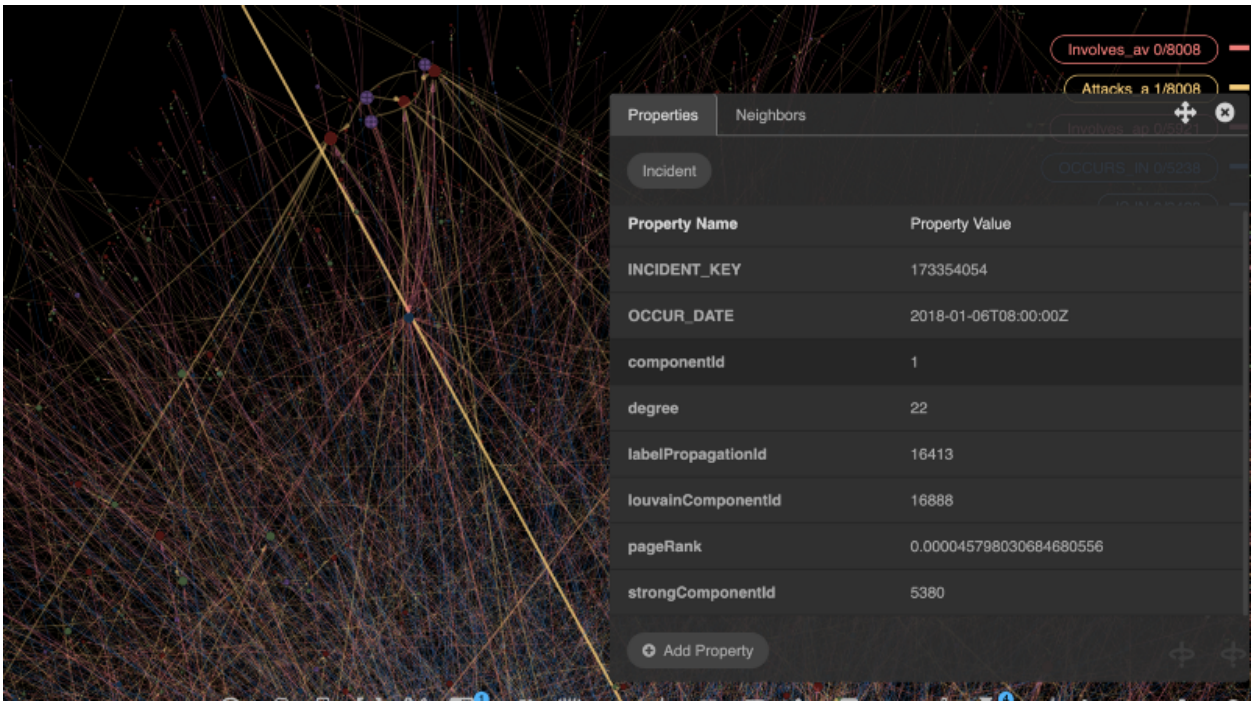

Properties
Neighbors
Boro
Property Name
Property Value
BORO
QUEENS
degree
536
pageRank
0.2827119137780375
Add Property
FRESH MEADOWS
QUEENS
OZONE PARK
HOWARD BEACH
Valley Stream


Properties
Neighbors
Boro
Property Name
Property Value
BORO
QUEENS
degree
536
pageRank
0.2827119137780375
Add Property
FRESH MEADOWS
QUEENS
OZONE PARK
HOWARD BEACH
Valley Stream



VISUAL INSIGHTS 2: Tracing a mass shooting to understand it in depth.





Here, we can see that in one incident in Brooklyn, there were 18 victims shot at by three perpetrators. We can also see that three victims, one shot at by each perpetrator, are dead while others have survived with injuries.

I. Use of CASSANDRA and DOCKER CONTAINERIZATION

For 2 points related to using an alternate NoSQL Database solution, we choose CASSANDRA.
For 5 points associated with Docker Containerization, we build the Cassandra image.

Apache Cassandra is a fully distributed NoSQL database that provides peak availability without a single point of failure by handling large amounts of data over many commodity servers. Cassandra handles all system activities in a proportional manner. There are no special cases like HDFS name nodes, MongoDB mongos, or MySQL Fabric processes that require special attention.

Few features of Cassandra:

- scalable, fault-tolerant, and consistent-column-oriented database.
- Cassandra is used by big companies such as Twitter, Facebook, Discord, Netflix, etc.

Installed Cassandra using docker

```
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker run -d -p 80:80 docker/getting-started
Unable to find image 'docker/getting-started:latest' locally
latest: Pulling from docker/getting-started
9981e73032c8: Pull complete
e5f90f35b4bc: Pull complete
ab1af07f990a: Pull complete
bd5777bb8f79: Pull complete
a47abff02990: Pull complete
d4b8ebd00804: Pull complete
6bec3724f233: Pull complete
b95ca5a62dfb: Pull complete
Digest: sha256:b558be874169471bd4e65bd6eac8c303b271a7ee8553ba47481b73b2bf597aae
Status: Downloaded newer image for docker/getting-started:latest
9d4ca8ef7c78b0dc73236d073bd6ce10bf4c6e9627571961bc5c9a99cc944cd8
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker pull cassandra:latest

latest: Pulling from library/cassandra
4e7e0215f4ad: Pull complete
47b9bffe95bb: Pull complete
479785166a86: Pull complete
6dd5e491c743: Pull complete
2ca38039150c: Pull complete
d7080e857bbe: Pull complete
c9a5c9c1900b: Pull complete
5a3685d8e027: Pull complete
56aba833300c: Pull complete
Digest: sha256:8d3187f77bfa34340e72735d642df18bc6db6ac7e6545ab471f3bce3c10b5dad
```

```
docker run --rm -d --name cassandra --hostname cassandra --network cassandra cassandra
8d33f9ca7bb36d03eae8582972d097f369f3315c87845386330ae8bcf32be762
df2c54efa83fd0279b25ab362a935599e6fae616bcb923f3b8ebc657d38bde36
```

Pulling the Cassandra Docker Image

```
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker pull cassandra
Using default tag: latest
latest: Pulling from library/cassandra
Digest: sha256:8d3187f77bfa34340e72735d642df18bc6db6ac7e6545ab471f3bce3c10b5dad
Status: Image is up to date for cassandra:latest
docker.io/library/cassandra:latest
```

Checking the Available Cassandra Images

```
Run 'docker image COMMAND --help' for more information on a command.
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker images
REPOSITORY              TAG       IMAGE ID       CREATED        SIZE
cassandra               latest    c2aa6d64412f   5 days ago     341MB
docker/getting-started  latest    157095baba98   7 months ago   27.4MB
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker run c2aa6d64412f
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a future r
elease.
CompileCommand: dontinline org/apache/cassandra/db/Columns$Serializer.deserializeLargeSubset(Lorg/apache/cassandra/io/util/DataInp
utPlus;Lorg/apache/cassandra/db/Columns;I)Lorg/apache/cassandra/db/Columns;
CompileCommand: dontinline org/apache/cassandra/db/Columns$Serializer.serializeLargeSubset(Ljava/util/Collection;ILorg/apache/cass
andra/db/Columns;ILorg/apache/cassandra/io/util/DataOutputPlus;)V
CompileCommand: dontinline org/apache/cassandra/db/Columns$Serializer.serializeLargeSubsetSize(Ljava/util/Collection;ILorg/apache/
cassandra/db/Columns;I)I
CompileCommand: dontinline org/apache/cassandra/db/commitlog/AbstractCommitLogSegmentManager.advanceAllocatingFrom(Lorg/apache/cas
```

Starting a Container

```
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker run -d --name cassandra-docker -p 9842:9842 cassandra

edb321c05e96e5c3492e674674659072d0313bf7cd22c5b136ebfcfc7fe9eeda
(base) maharshsoni@Maharshs-MacBook-Air ~ % docker exec -it cassandra-docker bash
```

Using a Keyspace

In Cassandra, a keyspace is a data container similar to a database in RDBMS. **Creating a “**posts” table that has three fields: **id**, **title,** and **content**.

```
root@edb321c05e96:/# cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.0.0 | Cassandra 4.0.7 | CQL spec 3.4.5 | Native protocol v5]
Use HELP for help.
cqlsh> CREATE KEYSPACE tutorial WITH replication = {'class': 'NetworkTopologyStrategy', 'datacenter1': '3'}  AND durable_writes =
true;

Warnings :
Your replication factor 3 for keyspace tutorial is higher than the number of nodes 1 for datacenter datacenter1

cqlsh> USE tutorial;
cqlsh:tutorial>
cqlsh:tutorial> CREATE TABLE posts (
           ...     id int PRIMARY KEY,
           ...     title text,
           ...     content text
           ... )
           ... ;
```

- SELECT all from table
- Inserting data into table
- Updating records

```
cqlsh:tutorial> SELECT * FROM posts;

 id | content | title
----+---------+-------

(0 rows)
cqlsh:tutorial> INSERT INTO posts (id, title, content) VALUES(1,'First post!', 'This is my first post!');
cqlsh:tutorial> SELECT * FROM posts;

 id | content               | title
----+-----------------------+-------------
  1 | This is my first post! | First post!

(1 rows)
cqlsh:tutorial> UPDATE posts SET content='Updated!' WHERE id=1;
cqlsh:tutorial> SELECT * FROM posts;

 id | content  | title
----+----------+-------------
  1 | Updated! | First post!

(1 rows)
```

- Reading specific rows
- Deleting records

```
cqlsh:tutorial> SELECT * FROM posts where id=1;

 id | content  | title
----+----------+-------------
  1 | Updated! | First post!

(1 rows)
cqlsh:tutorial> SELECT title FROM posts;

 title
-------------
 First post!

(1 rows)
cqlsh:tutorial> DELETE FROM posts WHERE id=1;
cqlsh:tutorial> SELECT title FROM posts;

 title
-------

(0 rows)
```

J. Comparing MongoDB, Neo4j (and RDBMS where applicable)

RDBMS	MongoDB	Neo4j
Data is stored in tables as rows and columns. Each row and columns are unique and atomic in nature. A table is an entity; each row is an entity instance, and a column is an attribute.	Data is stored as collections of Documents. Supports JSON document format. Each document represents a row in a relational DBMS. It supports nested documents.	Data are represented as nodes and edges in a graph database. Nodes represent entities, whereas Edges represent relationships between the entities.
Vertical scaling (scaling up) is supported by RDBMS by increasing RAM.	MongoDB supports horizontal scaling with scale-out options such as sharding and cluster computing. Sharding is possible using MongoDB atlas.	Neo4j and other graph databases do not support sharding or partitioning.
Tables are linked via primary key-foreign key relationships.	There is no primary key-foreign key relationship in MongoDB	Relationships between the entities are represented as edges. Hence it has foreign keys.
Complex queries require joins by joining multiple tables, which becomes an intensively expensive operation.	Aggregations are easier and faster using aggregation pipelines provided by MongoDB, which helps perform aggregations in stages.	Neo4j doesn't support an aggregation framework.
Data is modeled using entity-relationship diagrams.	MongoDB is schema-less and each document is independent	Neo4j is schema-free and schema optional. Data modeling can be done using graphs (nodes, edges, and properties)
RDBMS is case insensitive,	MongoDB is case sensitive	Neo4j is also case sensitive

Keywords and aggregate functions in MongoDB equivalent to RDBMS

RDBMS	**MongoDB**
Group by	$group
Count	$count and db.collection.count()
Select	$find, $project
Limit	$limit
Order by	$sort
Where	$match
	Other aggregate functions used: $unwind, $gt, $ne, $sum, $graphLookup

K. HW blog and video posted publicly, and other bells and whistles

HW blog	https://medium.com/@sachinsm2022/accelerating-data-analytics-with-nosql-database-solutions-mongodb-and-neo4j-aa7f358d540
Video posted publicly	https://vimeo.com/770357113

L. Key Learnings, and Next Steps in our learning Journey:

MongoDB:

- In MongoDB, every document will be assigned a unique id called '_id' which is assigned automatically to uniquely identify a document.
- There is no primary key-foreign key relationship in MongoDB, unlike RDBMS, which makes it easier to perform complex queries and aggregations without using cumbersome joins between the tables.
- Aggregation pipelines help execute complex queries. Some of the aggregation keywords we came across are $match, $group, $project, $sort, $lookup, etc.

- Sharding can be used in MongoDB to achieve horizontal scaling by distributing different parts of the dataset. Each of them is called a shard, which enhances the performance and increases each shard's capacity to serve the requests.
- There are three nodes in the MongoDB cluster. One primary shard and two secondary shards.
- We also got hands-on experience converting CSV datasets into JSON files using the CSV Reader and data frames in pandas.
- Other than Tableau and other traditional visualization tools, we got to learn to do data visualization using MongoDB charts which also provide us various options to filter data using queries, embed, export, and enable accessibility of charts created to others(public).

Neo4j:

- In Neo4j, APOC (Awesome Procedures on Cypher) is an add-on library that offers numerous procedures and functions that greatly increase its capabilities.
- We can create and manage as many local databases with Neo4j Desktop in a given database management system (DBMS) instance.
- Additionally, the desktop offers and hosts a variety of graph apps, which are designed to operate with graph datasets. Two of these graph applications are Neo4j Browser and Bloom. Cypher queries can be executed against Neo4j graph data via the browser, and the results can be viewed. Graph data is visualized using Bloom and search inputs.
- Explored the options for importing data into Neo4j:
    1. JSON file into Neo4j: Installed APOC plugins and enabled permissions to import the data.
    2. Using CSV file
- Each DBMS in Neo4j has a separate neo4j.conf file, which contains all the configurations and settings specific to that DBMS and that can be changed according to our requirements.
- Got exposure to various built-in commands, such as MERGE, MATCH, etc, while performing analysis using queries.

M. Conclusions and our Data-Backed Insights to NYPD:

1. Increase policing efforts during economic downturns and allocate funds to mitigate poverty, and joblessness.
2. Increase policing hours during evenings and midnight as most shooting incidents are prone to occur in the evenings and at midnight.
3. Provide aid to ensure access to economic opportunities during the holiday season.
4. Increase fund allocation towards policing efforts in Brooklyn. According to the data, Brooklyn needs ten times more policing efforts than Staten Island.
5. Victims and perpetrators are primarily 25-44. The reasons could be multifarious. A gut check informs that this could be due to a downturn in economic opportunities and susceptibility to getting involved in illicit activities such as the consumption of drugs and gangs.
6. The incidents follow a large significant pattern of victims and perpetrators of the same race. This informs that crimes are most prone within a community rather than across communities. As per

the case study of Singapore's economic research, diversity leads to fewer crimes. Hence based on the visual insights and this case report, efforts must be made to increase the diversity of the population across boroughs to reduce shootings.

References:

1. https://www.datastax.com/blog/apache-cassandra-architecture-vs-top-nosql-competitors
2. https://cassandra.apache.org/doc/latest/cassandra/getting_started/installing.html
3. https://www.mongodb.com/docs/drivers/python/
4. https://www.mongodb.com/docs/develop-applications/
5. https://templatelab.com/flow-chart-template/#google_vignette
6. https://www.youtube.com/watch?v=_IgEHRfi5FY
7. https://graphxr.kineviz.com/
8. https://data-importer.neo4j.io/
9. https://py2neo.org/2021.1/
10. https://medium.com/analytics-vidhya/import-csv-file-into-mongodb-9b9b86582f34
11. https://rod4423.medium.com/import-csv-by-neo4j-cypher-and-scenario-example-fb8c8c7e4dd1
12. https://micropyramid.medium.com/mongodb-crud-operations-with-python-pymongo-a26883af4d09
13. https://medium.com/li-ting-liao-tiffany/visualize-open-data-using-mongodb-in-real-time-2cca4bcca26e
14. https://hevodata.com/learn/mongodb-sharding/
15. https://github.com/evagian/California-road-network-NEO4J-CYPHER-graph-and-queries
16. https://neo4j.com/developer/guide-import-json-rest-api/
17. https://www.youtube.com/watch?v=oXziS-PPIUA