12/20/2022

Big Data - Fall 2022 - Final Project

Streaming Crime Data To Evaluate Neighborhood Safety

Prepared By: **The Stream Team**

Ryan Kim [rk2546]

Dipak Patel [dp3148]

Aditya Chavan [ac8586]

Bhavish Yalamanchi [by2162]

TABLE OF CONTENTS

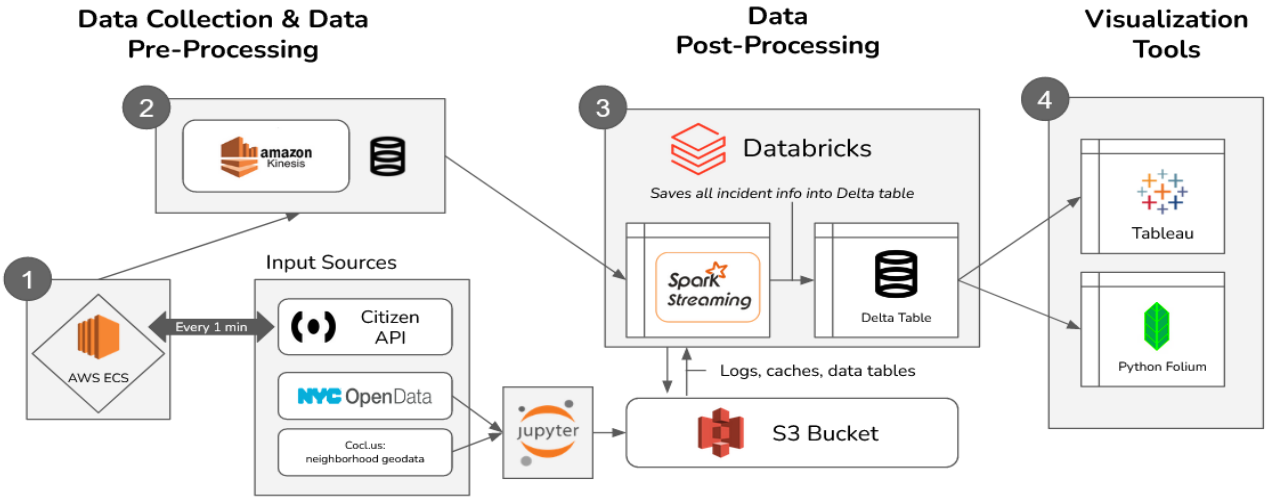
Problem Statement

How can we leverage real time data streaming to enable an end application to visualize public safety in particular NYC neighborhoods?

Input Sources:

Name	Source	Description	Link/URL
NYPD Arrest Data (Year to Date)	NYC OpenData	*"This is a breakdown of every arrest effected in NYC by the NYPD during the current year. This data is manually extracted every quarter and reviewed by the Office of Management Analysis and Planning. Each record represents an arrest effected in NYC by the NYPD and includes information about the type of crime, the location and time of enforcement.In addition, information related to suspect demographics is also included. This data can be used by the public to explore the nature of police enforcement activity. Please refer to the attached data footnotes for additional information about this dataset."* - (official description on NYC OpenData page)	https://data.cityofnewyork.us/Public-Safety/NYPD-Arrest-Data-Year-to-Date-/uip8-fykc
Citizen Trending API	Citizen	Each response contains a JSON list of about 40 events. These events may contain duplicates. This is meant for populating the mobile app and so some work was needed to be done in order to easily use the data for streaming.	https://citizen.com/
NYC Neighborhood Geodata	cocl.us	GeoJSON data that allows for mapping NYC neighborhoods on maps. Contains GeoPoint data for each neighborhood as well as neighborhood specifics such as name and borough.	https://cocl.us/new_york_dataset

Architecture Diagram & Overview



Step 1: Pre-Processing

In this step we gathered all the data and formatted it in a way that can be easily read by **Databricks** and **Spark Structured Streaming**. In order to combine the data we had to also use neighborhood mapping which we talk about in later sections. The brains of the pre processing occurs in the **AWS Kinesis** data stream. Before sending the streaming citizen data to Kinesis we decode the JSON from the **Citizen API** and we send each one 1 by 1 to the data stream. Since we use **ECS** we can make this more robust such as searching for duplicates and batching data together before sending it to Kinesis. This could have cut down on AWS costs as well.

Step 2: Data Streaming

In this step, we take data which is sent from the **ECS** instance into the **AWS Kinesis** data stream. This process is easily repeatable for any needs. It takes in data from any source and then creates a data stream that other applications can tap into. The stream has some manageable parameters such as removal of stale data. The default was to remove after 1 day, which in our case was more than enough. This allows us to stay true to our end goal of enabling real time end applications. Kinesis can accept data from a variety of sources. In our case we use the **boto3 Python aws client** to send data over to the stream. As mentioned in the previous section this can be improved and made more robust. We will continue to discuss details of these steps in the following sections.

Step 3: Post-Processing

Post-processing occurs within **Databricks**, which provides an interface similar to JupyterHub Python notebooks as well as data storage options through its *delta tables*, data collections represented as tables accessible to external visualization tools. These delta tables and any cached data from Databricks notebooks are stored in **AWS S3 buckets**.

Data streamed into the **AWS Kinesis** stream are piped into our Databricks notebooks via **Spark Structured Streaming**, an extended framework of Spark that accepts data from a variety of streaming sources such as TCP connections, API gateways, and more importantly AWS Kinesis streams. Through the Spark Structured Streaming framework, data is transformed from Byte data into PySpark dataframes. From here, we perform transformations onto the dataframes so that the latest incident updates are either appended to our aggregated incidents data (if not records of that incident exist) or replace existing incident data with the latest, up-to-date info (if a record about that incident already exists). Incidents are also mapped to neighborhoods in NYC through these transformations. After these transformations, the latest version of our aggregated incidents data are pushed into a delta table named “incidents”. This “incidents” delta table is accessible to external tools and can be read as long as proper access to the data cluster on DataBricks is provided.

In addition to aggregated incidents, the historic NYC crime data mapped to NYC neighborhoods from our pre-processing stage are uploaded into another delta table “nypd_arrests_with_neighborhoods”. This delta table is also accessible to external tools.

Step 4: Visualization

Our data processing system has multiple applications, including real-time visualization of data. It can process a wide range of data sources and has employed both code-based approaches using Python and the Folium library, as well as visualization tools like Tableau. The system's data has potential use in navigation, where it could be integrated with systems like Google Maps to help users make informed decisions about routes based on incidents in the area. We have also developed a Tableau dashboard that is connected to our Databricks database and displays real-time data. The dashboard includes a map that plots incidents, as well as graphs that show counts of incidents by category, neighborhood, and time. It also has filters that allow users to filter the data based on different values. The data and visualization tools can be used in a variety of applications to help users make informed decisions and improve safety and security.

Pre-Processing: Data streaming & Neighborhood Mapping

Data Streaming

Streaming data from Python running in EC2 into Kinesis is a powerful way to process large amounts of real-time data. Kinesis is a fully managed service that allows developers to easily and reliably collect, process, and analyze streaming data at any scale. By streaming data from our Python script (stream.py), which fetches data from the public Citizen API, hosted in EC2 into Kinesis, we can take advantage of the distributed computing capabilities of Kinesis to process and analyze the data in real-time. We can expand our streaming script to be as robust as we want it to be, thanks to the power of having a full linux box under the hood. An alternative would have been to use Lambda functions to employ a serverless method of fetching. In either case the proper IAM roles have to be granted inside AWS.

To the right is the code to parse the JSON returned from the Citizen API and pass records of each incident into the AWS Kinesis stream. The `timer()` method is run on an infinite loop and is called every 60 seconds; this time delay is controllable through a constant variable `_TIME_DELAY`. The scheduling of the loop is controlled by Python's `schedule` package.

Prior to sending data to the AWS Kinesis stream, the data is modified slightly. Specifically, a new field "updates_list" is set as an array. The original JSON data has an "updates" field that is a key-value dictionary. To properly parse this data in Spark Structured Streaming, we convert this dictionary into an array.

```
# The timer function that runs every `_TIME_DELAY` seconds
def timer():
    print('timer entered')

    # Get current time for the filename
    # timestr = time.strftime("%Y-%m-%d_%H-%M-%S")
    # Make the request to Citizen
    r = requests.get(\
        _CITIZEN_REQUEST_URL,\
        _CITIZEN_REQUEST_HEADERS\
    )

    #####Upload to kinesis
    for idx, x in enumerate( r.json()['results'] ):
        x['updates_list'] = [[k,v['text'],v['ts'],v['type']] for k,
v in x['updates'].items()]
        json_object = json.dumps(x, indent=2)
        client = boto3.client('kinesis', region_name='us-east-1')
        partition_key = str(uuid.uuid4())
        try:
            response =
client.put_record(StreamName='bigdatafinalproject',
Data=json_object, PartitionKey=partition_key)
            print("Placed object into kinesis \n
{}\n\n".format(partition_key))
        except ClientError:
            print("Couldn't put record in stream %s.", self.name)
            raise

# This is the schedule that runes `timer()` everu `_TIME_DELAY`
seconds
schedule.every(_TIME_DELAY).seconds.do(timer)

# Infinite loop to run the scheduler
while True:
    schedule.run_pending()
    time.sleep(1)
```

Neighborhood Mapping

In addition to AWS Kinesis streaming, we pre-process the historic NYC crime data by attributing each arrest to neighborhood. Each arrest provided by the original NYPD Arrest Data is attributed to an NYC borough and defined latitude-longitude coordinates. To achieve greater fidelity in the general location of each crime and to define a common field between the historic arrest data and processed Citizen data, each arrest must be attributed to NYC neighborhood. This is performed in a Python Jupyter Notebook (*NeighborhoodCrimes.ipynb*) and takes advantage of PySpark dataframes and transformations to efficiently modify each arrest record.

The first step is to extract geodata for each NYC neighborhood such as its approximate latitude-longitude coordinates and borough. The data is provided through cocl.us's GeoJSON data file.

```
with open('inputs/newyork_data.json') as json_data:
    newyork_data = json.load(json_data)
neighborhoods_data = newyork_data['features']
# define the dataframe columns
column_names = ['Borough', 'Neighborhood', 'Latitude', 'Longitude']
# instantiate the dataframe
neighborhoods = pd.DataFrame(columns=column_names)
for data in neighborhoods_data:
    borough = neighborhood_name = data['properties']['borough']
    neighborhood_name = data['properties']['name']
    neighborhood_latlon = data['geometry']['coordinates']
    neighborhood_lat = neighborhood_latlon[1]
    neighborhood_lon = neighborhood_latlon[0]
    neighborhoods = neighborhoods.append({'Borough': borough,
                                          'Neighborhood':
neighborhood_name,
                                          'Latitude':
neighborhood_lat,
                                          'Longitude':
neighborhood_lon}, ignore_index=True)
```

The next step is to uncover which neighborhood each arrest is located to closest. This can be performed by using the Haversine distance calculation method, which is a mathematical formula used to calculate the great-circle distance between two points on a sphere. The process can generally be described as, for each arrest, calculate the Haversine distance between that arrest's geopoint with each neighborhood's geopoint; the neighborhood that produces the shortest distance is considered that arrest's "neighborhood".

Particular transformations have to be made before the distance calculation, such as converting the “Borough” field inside the “neighborhoods” dataframe into a new “NBorough” field that contains only the first initial of the borough’s name. This is because the NYPD Arrest data codifies borough by their initials and not the full borough name.

```
NBDict = {"Manhattan":"M","Bronx":"B","Staten
Island":"S","Brooklyn":"K","Queens":"Q"}
mapping = create_map([lit(x) for x in
chain(*NBDict.items())])

neighborhoodsDF = spark.createDataFrame(neighborhoods)\
    .withColumn("NBorough", mapping[col("Borough")])
neighborhoodsDF.show()
```

We and then perform a Left join on Borough with the NYPD Arrest data so that every arrest is now linked to all neighborhoods and their geodata. The join type is Left instead of Cross because we we need to only perform distance calculations between an arrest’s location with neighborhoods within the borough assigned to the arrest - this optimizes the time needed to perform the distance calculation as neighborhoods in boroughs not attributed to an arrest don’t need to be considered.

```
neighborCrimeDF = openCrimeDF.join(
    neighborhoodsDF.select([
        "NBorough",
        "Neighborhood",
        col("Latitude").alias("NLatitude"),
        col("Longitude").alias("NLongitude")
    ]),
    openCrimeDF.ARREST_BORO == neighborhoodsDF.NBorough,
    "left")
neighborCrimeDF.show()
```

To perform the Haversine distance calculation, we create a user-defined function that is executed on each row of our joined dataframe. Then we perform a Window function where, for each arrest, we rank each row based on the newly-calculated “Distance” field such that the closest neighborhood can be identified as rank #1. We then filter the dataframe so that only rows ranked as #1 remain - in other words, each arrest from the original NYPD Arrests data still remains and now is attributed to the closest neighborhood.

```
# the haversine function expects an iterable of the form
(lat, lon)
!pip install haversine
from haversine import haversine
from pyspark.sql.types import ArrayType, DoubleType
from pyspark.sql.window import Window

def string2array(lat,long):
    if lat is None or long is None:
        return None
    return [float(lat), float(long)]

def haversine_miles(x, y):
    return haversine(x, y, unit='mi')

udf_haversine = udf(haversine_miles, DoubleType())
udf_string2array = udf(string2array,
ArrayType(DoubleType()))

window =
Window.partitionBy("ARREST_KEY").orderBy(col("Distance").asc
())

neighborCrimeDF2 = neighborCrimeDF\

.withColumn("LatLong",udf_string2array(col("Latitude"),col("
Longitude")))\

.withColumn("NLatLong",udf_string2array(col("NLatitude"),col
("NLongitude")))\

.withColumn("Distance",udf_haversine(col("LatLong"),col("NLa
tLong")))

NCDF =
neighborCrimeDF2.withColumn("DistanceRank",rank().over(windo
w)).filter(col("DistanceRank")==1)
```

This new dataframe is then exported and uploaded as a delta table to Databricks, for potential use inside our Databricks notebooks or externally via publicly available tools that can interface with our Databricks’ delta tables.

Recap: *We wish to attribute crimes to neighborhoods because we wanted a better understanding of how crimes related to neighborhoods as opposed to boroughs. Boroughs are too large on the geographic scale - associating crimes to boroughs prevents a detailed inquiry into how crime is distributed within each borough. By performing this pre-processing step,* ***we can now see the distribution and patterns of crime in the city at the neighborhood level, and to identify areas that may require further attention or resources.*** *This data is passed onto our Databricks cluster for public use within and outside of Databricks.*

Post-Processing: Databricks, Spark Structured Streaming

Kinesis Streaming & Spark Structured Streaming

Data streamed from AWS Kinesis is accessed from within a Databricks notebook through Spark Structured Streaming (from now on referred to as “SSS”). SSS comes with the capability to access any AWS Kinesis stream, assuming that the proper access keys and secrets are provided. Furthermore, we’re hosting our post-processing infrastructure on Databricks, which simplifies the process of accessing AWS services. This project concerns the use of PySpark’s implementation of Structured Streaming.

Similar to Spark/PySpark, SSS operates on DataFrames and will import the streaming data into a dataframe. These dataframes are unique in how they listen in on the Kinesis stream, updating themselves every time new data is pushed to the Kinesis stream.

```
awsAccessKeyId = "<AWS ACCESS KEY>" # update the access key
awsSecretKey = "<AWS SECRET KEY>" # update the secret key

kinesisStreamName = "bigdatafinalproject" # update the kinesis
stream name
kinesisRegion = "us-east-1"

kinesisDF = spark\
  .readStream\
  .format ("kinesis") \
  .option ("streamName", kinesisStreamName) \
  .option ("region", kinesisRegion) \
  .option ("initialPosition", "LATEST") \
  .option ("format", "json") \
  .option ("awsAccessKey", awsAccessKeyId) \
  .option ("awsSecretKey", awsSecretKey) \
  .option ("inferSchema", "true") \
  .load()
```

Parsing Kinesis Streaming Data into Incidents

The first difficulty is to parse the streaming data, which is provided in the ByteType format, into the StringType through

```
result = kinesisDF.selectExpr(' CAST(data AS STRING) as
decoded').select("decoded")
display(result)
```

typecasting.

```
schema = StructType([
        StructField("key", StringType(), True),
        StructField("title", StringType(), True),
        StructField("address", StringType(), True),
        StructField("cs", FloatType(),True),
        StructField("ts", FloatType(),True),
        StructField("cityCode", StringType(), True),
        StructField("neighborhood", StringType(), True),
        StructField("categories", ArrayType(StringType()), True),
        StructField("latitude", FloatType(), True),
        StructField("longitude", FloatType(), True),
        StructField("closed", BooleanType(), True),
        StructField("updates_list",
ArrayType(ArrayType(StringType())), True)
    ])

jsonEdits = result.select (
    from_json ("decoded", schema).alias("json")
) # Parse the column "value" and name it "json"
display(jsonEdits)
```

The second difficulty is actually choosing which data to parse from the JSON. Each row, now cast into strings, can be parsed further into JSON. However, many irrelevant fields unique to Citizen's needs are present within each row of data.

Therefore, we've chosen to parse out these fields from each incident received from Citizen:

- **key** : string
- **address** : string
- **cs** : timestamp
- **ts** : timestamp
- **cityCode** : string
- **neighborhood** : string
- **categories** : Array<string>
- **latitude** : float
- **longitude** : float
- **closed** : boolean
- **updates_list** : Array<Array<string>>

Updating the Latest Updates into our Existing Dataframe of Incidents

One important caveat about AWS Kinesis and our pre-processing pipeline is that the data streamed from AWS Kinesis is a raw count of current incidents. This means that AWS Kinesis can (and will) send data of incidents that it sent already in prior data updates. In other words, ***duplicate data will constantly be streamed into our dataframes****. We need ensure that duplicate data doesn't exist in our dataframes.*

We need the following process:

1. Extract the latest update for each incident received from Kinesis
2. Either a) insert a new incident into our records if it doesn't exist in our records yet, or b) update an incident with the latest update if the incident already exists in our records.

To retrieve the latest update for each incident retrieved from Kinesis, we need to explode the "updates_list" array for each row.

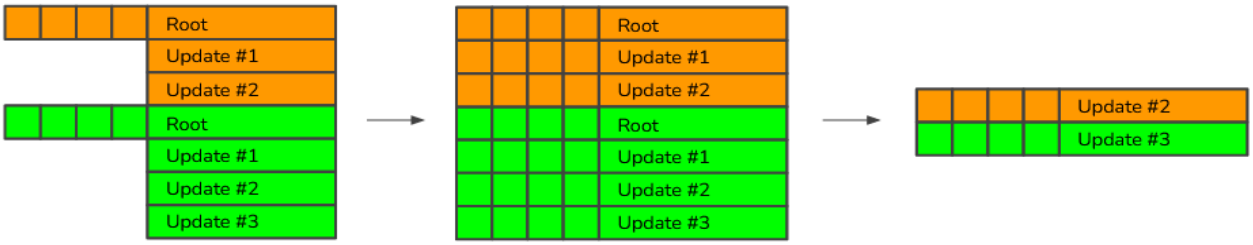
```
updateStreamDF = jsonEdits\
    .select(
        col("json.key").alias("key"),
        col("json.address").alias("address"),
        to_utc_timestamp(from_unixtime(col("json.cs")/1000.0,
'yyyy-MM-dd HH:mm:ss'), 'EST').alias("cs"),
        to_utc_timestamp(from_unixtime(col("json.ts")/1000.0,
'yyyy-MM-dd HH:mm:ss'), 'EST').alias("ts"),
        col("json.cityCode").alias("cityCode"),
        col("json.neighborhood").alias("neighborhood"),
        col("json.categories").alias("categories"),
        col("json.latitude").alias("latitude"),
        col("json.longitude").alias("longitude"),
        col("json.closed").alias("closed"),
        col("json.updates_list").alias("updates")
    )\
    .withColumn("status",explode("updates").alias("status"))\
    .drop("updates")\

.select("key","address","cs","ts","cityCode","neighborhood","catego
ries","latitude","longitude","closed",
            col("status")[0].alias("status_key"),
            col("status")[1].alias("status_text"),
            to_utc_timestamp(from_unixtime(col("status")[2]/1000,
'yyyy-MM-dd HH:mm:ss'), 'EST').alias("status_ts"),
            col("status")[3].alias("status_type")
    )
```

We then aggregate by incident “key”, and then extract the latest info (aka the last row for each incident).

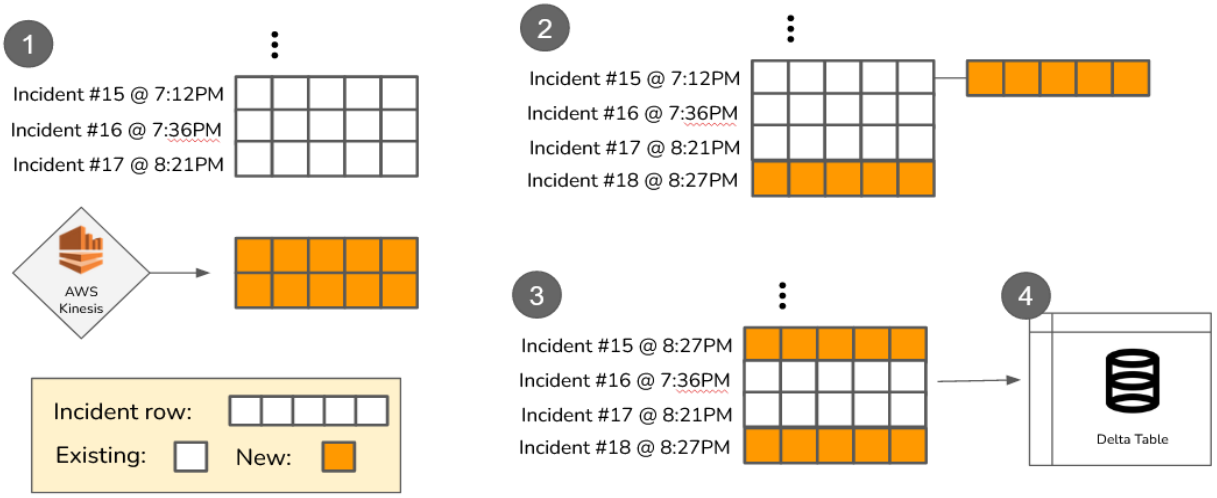
```
def GetImpliedNeighborhood(x):
    try:
        return x.split(",")[0]
    except:
        return x
udf_GetImpliedNeighborhood = udf(GetImpliedNeighborhood,
StringType())

incidentsStreamDF = updateStreamDF\
    .withColumn("impliedNeighborhood",
udf_GetImpliedNeighborhood(col("neighborhood")))\
    .groupBy("key")\
    .agg(
        min('status_ts').alias('firstTimeStamp'),
        max('status_ts').alias('lastTimeStamp'),
        last("address").alias("address"),
        last("cityCode").alias("cityCode"),
        last("cs").alias("cs"),
        last("ts").alias("ts"),
        last("neighborhood").alias("neighborhood"),
        last("impliedNeighborhood").alias("impliedNeighborhood"),
        last("categories").alias("categories"),
        last("latitude").alias("latitude"),
        last("longitude").alias("longitude"),
        last("closed").alias("closed"),
        last("status_type").alias("last_status_type"),
        last("status_text").alias("last_status_text")
    )
```

The results of the code above are visualized below in two separate images. Note that the images do not exactly match the transformations performed in the code - they are merely visual descriptions of how the code performs on a broad scale.

1. Explode the “updates_list” field for each incident, then extract the latest update:



2. Update our records with either new incidents or updates to existing incidents:



In the end, the `incidentsDF` dataframe contains all incidents retrieved from AWS Kinesis.

The last step is to cache this data into a delta table so that external tools can interact with the incident data.

We do this by overwriting our "incidents" delta table with the recoreds from the `incidentsDF` dataframe.

```
incidentsStreamToTable = (

incidentsStreamDF.writeStream.queryName("Incidents_Stream_To_Tab
le")\
        .format("delta")\
        .outputMode("complete")\
        .option("checkpointLocation",
"/tmp/delta/incidents/_checkpoints/")\
        .toTable("incidents")
)
```

The `incidents` delta table now contains the most up-to-date records of incidents happening in New York City.

We perform this method of post-processing for each incident because, as far as we understand, the most important data about each incident is the latest update. While it is better from a historic standpoint to keep a record for every development in an incident, our concern is mainly to update visualization tools so that they access the last known information about an incident.

Why Databricks and Spark Structured Streaming?

Databricks is an online cloud and collaborative solution to data streaming. Databricks allows for notebooks to communicate with AWS services such as AWS Kinesis. Furthermore, Databricks notebooks operate off of Python and provide a user experience very similar to that of Jupyter notebooks. Since many of us have operated with Jupyter notebooks for the duration of this course, we felt it was most efficient to operate with a familiar user interface that offered solutions to our problem of data streaming.

Spark Structured Streaming is an extended framework of Spark that allow us not only the access to many of Spark's transformation functions but also provides the schema to accept the data streamed from the multiple resources. In this case, data streamed through

AWS Kinesis needed various transformations such as typecasting and aggregations in order to be processed properly. SSS was a lightweight solution that was readily available to us in Databricks and therefore fit all of our requirements.

Visualizations:

Our data processing system has several applications, including real-time visualization of data to gain insights. This system is designed to be flexible and can be used to process a wide range of data sources, such as the Citizen's API that it is currently using. To visualize the data, we have employed both code-based approaches using Python and the Folium library, as well as visualization tools like Tableau.

Python Folium

Using code we tried doing this using Python and the Folium library. Folium is a Python library that allows you to create interactive leaflet maps. It is built on the popular leaflet.js library and can be used to visualize data on top of maps. One of the key features of Folium is that it can be used to visualize data on interactive maps using a variety of different map styles. It also supports the ability to add markers, polylines, and other overlays to the map, as well as the ability to customize the appearance of the map and the data being displayed.

```
NC_CountDF =
NCDF.groupBy(["Neighborhood","NBorough"]).count().toDF("N","B","
Count").drop("B")

NeighborhoodCrimesDF =
neighborhoodsDF.join(NC_CountDF,neighborhoodsDF.Neighborhood ==
NC_CountDF.N, "left").drop("N")

maxCrimes =
NeighborhoodCrimesDF.orderBy(desc("Count")).take(1)[0]["Count"]
minCrimes =
NeighborhoodCrimesDF.orderBy(asc("Count")).take(1)[0]["Count"]

import branca
import branca.colormap as cm

colormap = cm.LinearColormap(colors=['lightblue','red'],
index=[minCrimes,maxCrimes],vmin=minCrimes,vmax=maxCrimes)

# create map of New York using latitude and longitude values
NeighborhoodCrimesMap = folium.Map(location=[latitude,
longitude], zoom_start=10)
NeighborhoodCrimesPD = NeighborhoodCrimesDF.toPandas()

# Read from "incidents" delta table
incidentsDeltaDF = spark\
    .read\
    .format("delta")\
    .options(header=True,inferSchema=True)\
    .load("dbfs:/user/hive/warehouse/incidents")
incidentsPD = incidentsDeltaDF.toPandas()

# add markers to map
AggregateCrimeMap = folium.Map(location=[latitude, longitude],
zoom_start=10)
for lat, lng, borough, neighborhood, count in zip(
    neighborhoodCrimesPD['Latitude'],
    neighborhoodCrimesPD['Longitude'],
    neighborhoodCrimesPD['Borough'],
    neighborhoodCrimesPD['Neighborhood'],
    neighborhoodCrimesPD["Count"]
):
    label = '{}, {}'.format(neighborhood, borough)
    label = folium.Popup(label, parse_html=True)
```

```
    folium.CircleMarker(
            [lat, lng],
            radius=6,
            popup=label,
            color='Black',
            fill=True,
            fill_color=colormap(count),
            fill_opacity=1.0,
            parse_html=False).add_to(AggregateCrimeMap)

for lat, lng, neighborhood in zip(
    incidentsPD['latitude'],
    incidentsPD['longitude'],
    incidentsPD['impliedNeighborhood'],
):
    label = '{}'.format(neighborhood)
    label = folium.Popup(label, parse_html=True)
    folium.CircleMarker(
                [lat, lng],
                radius=3,
                popup=label,
                color='Blue',
                fill=True,
                fill_color="Blue",
                fill_opacity=0.75,
                parse_html=False).add_to(AggregateCrimeMap)

AggregateCrimeMap
```

One potential application of our data is in the field of navigation, where it could be used in conjunction with systems like Google Maps to help users make informed decisions about which routes to take based on the incidents in the area. This type of visualization can be useful for a wide range of applications, including improving safety and security. By providing other applications with access to our data through an API, we can enable a range of possibilities for using this information. For example, a navigation app could incorporate our data into their routing algorithms to avoid areas with high incident rates.

Tableau

We developed a Tableau dashboard that is connected to our Databricks database and displays data in real-time. Tableau is a popular data visualization tool known for its user-friendly interface and versatility. It allows users to easily create a wide range of visualizations, including bar charts, line graphs, maps, and scatter plots, using drag-and-drop functionality. One of the key advantages of Tableau is its high level of customization, allowing users to alter the appearance of their visualizations and add various types of

interactivities, such as filtering and highlighting. Tableau also has a large library of pre-built visualizations and templates, which can make it easier for users to get started with creating their own visualizations.

Our Tableau dashboard is connected to a Databricks delta tables as its data source and displays real-time data. The map on the dashboard plots all the entries from the “incidents” delta table, which are trending incidents added to the table. The map provides details about the incidents, such as the type of incident, description, location, neighborhood, time, and categories. The dashboard also includes filters that allow users to filter the data based on these values. Other graphs on the dashboard include counts of incidents by category, neighborhood, and time. These graphs can provide useful insights about the incidents, such as the frequency of incidents in different neighborhoods and at different times.

In addition to its usefulness for navigation, our Tableau dashboard with its various filters and graphs can also help users to delve deeper into the data and understand trends and patterns. By identifying areas that are particularly prone to certain types of incidents, users can take steps to address these issues and potentially improve safety in those areas. Overall, our data and visualization tools offer a range of benefits and can be used in a variety of applications to help users make informed decisions and improve safety and security. Our Tableau dashboard with archived data is available online on the following link:

https://public.tableau.com/app/profile/aditya3655/viz/shared/RJCMMJQTW

Screenshots:

Folium:

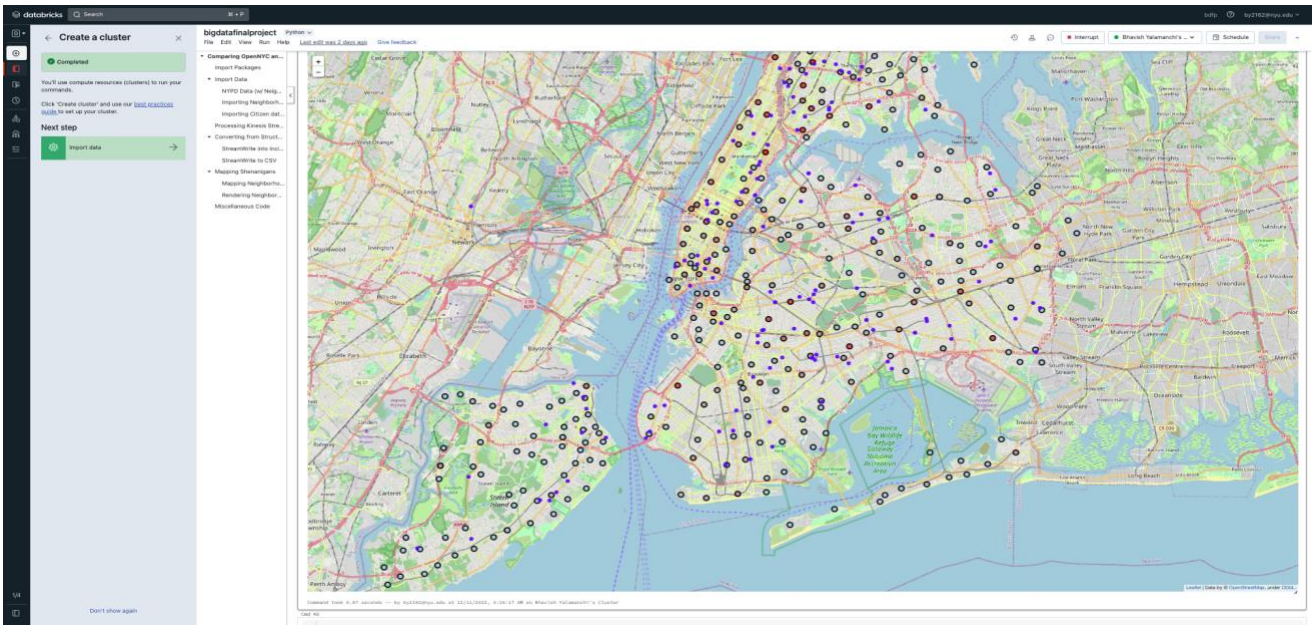
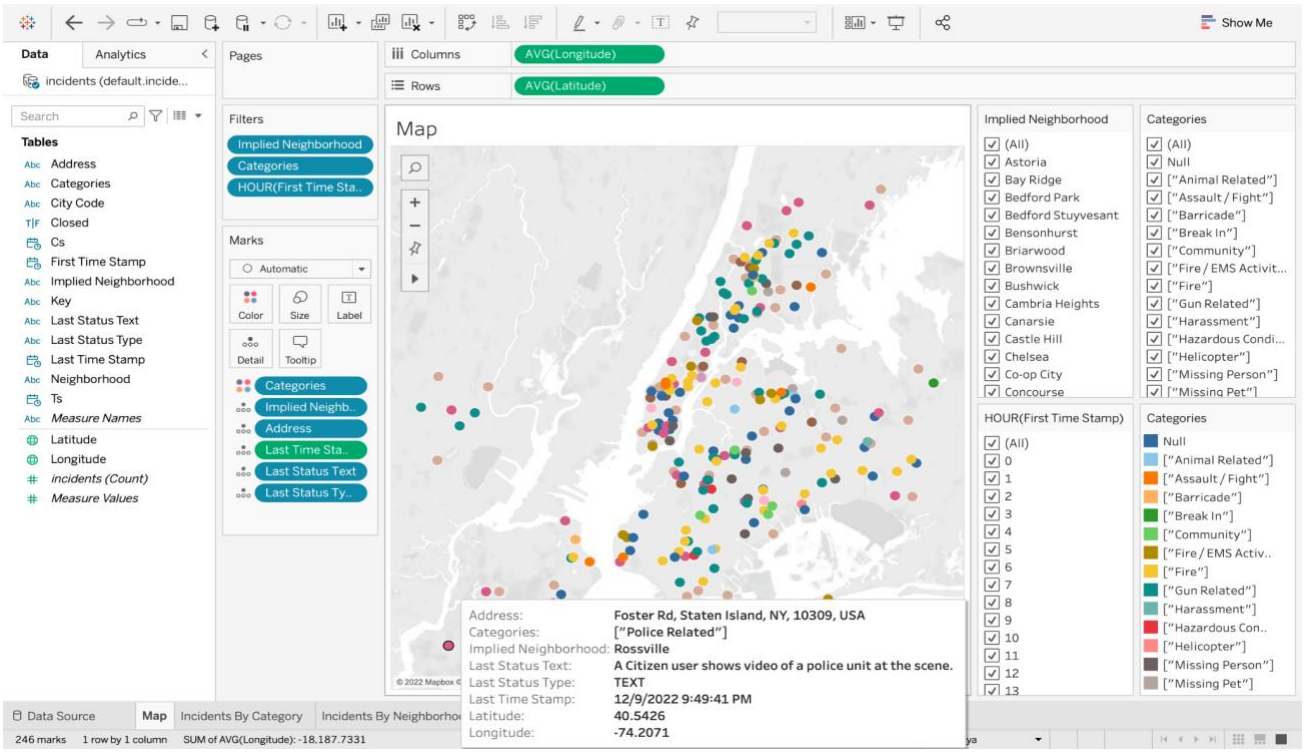
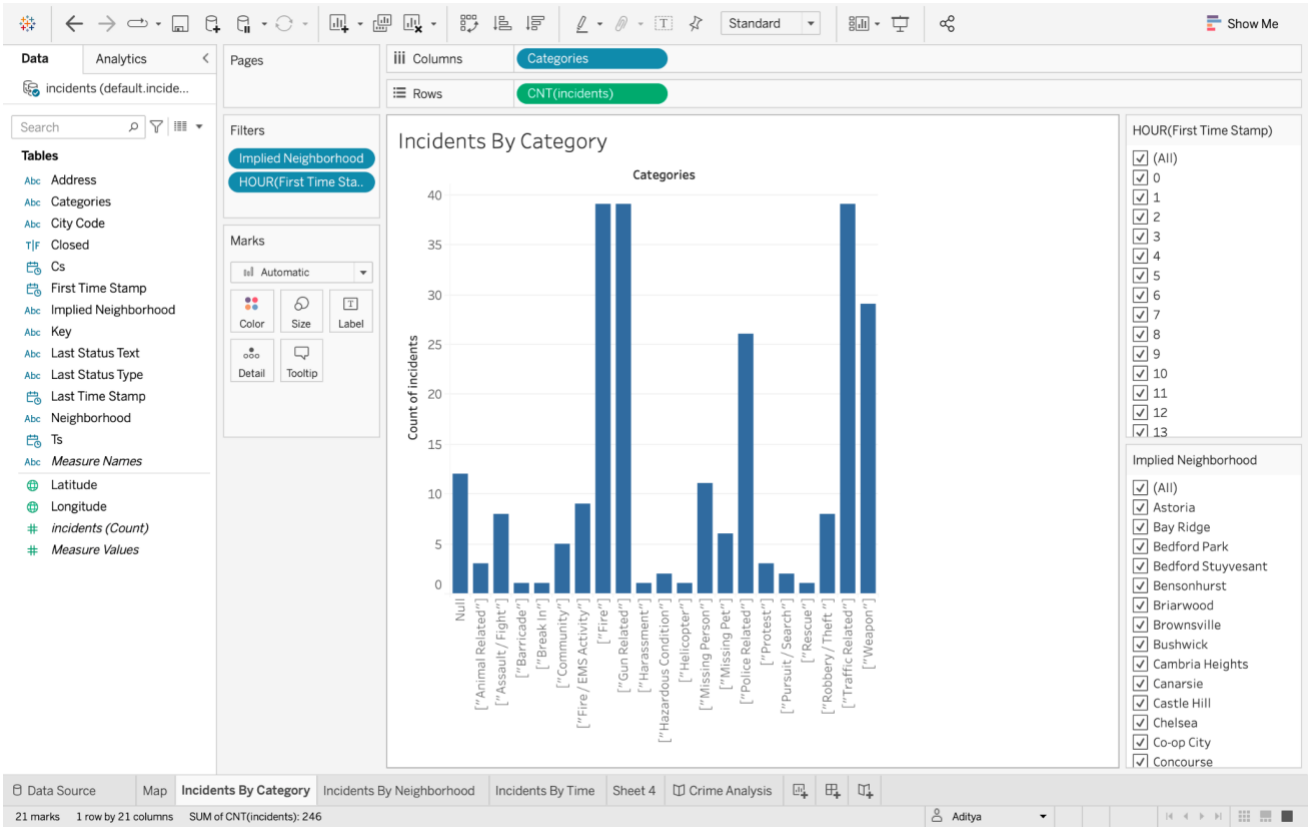
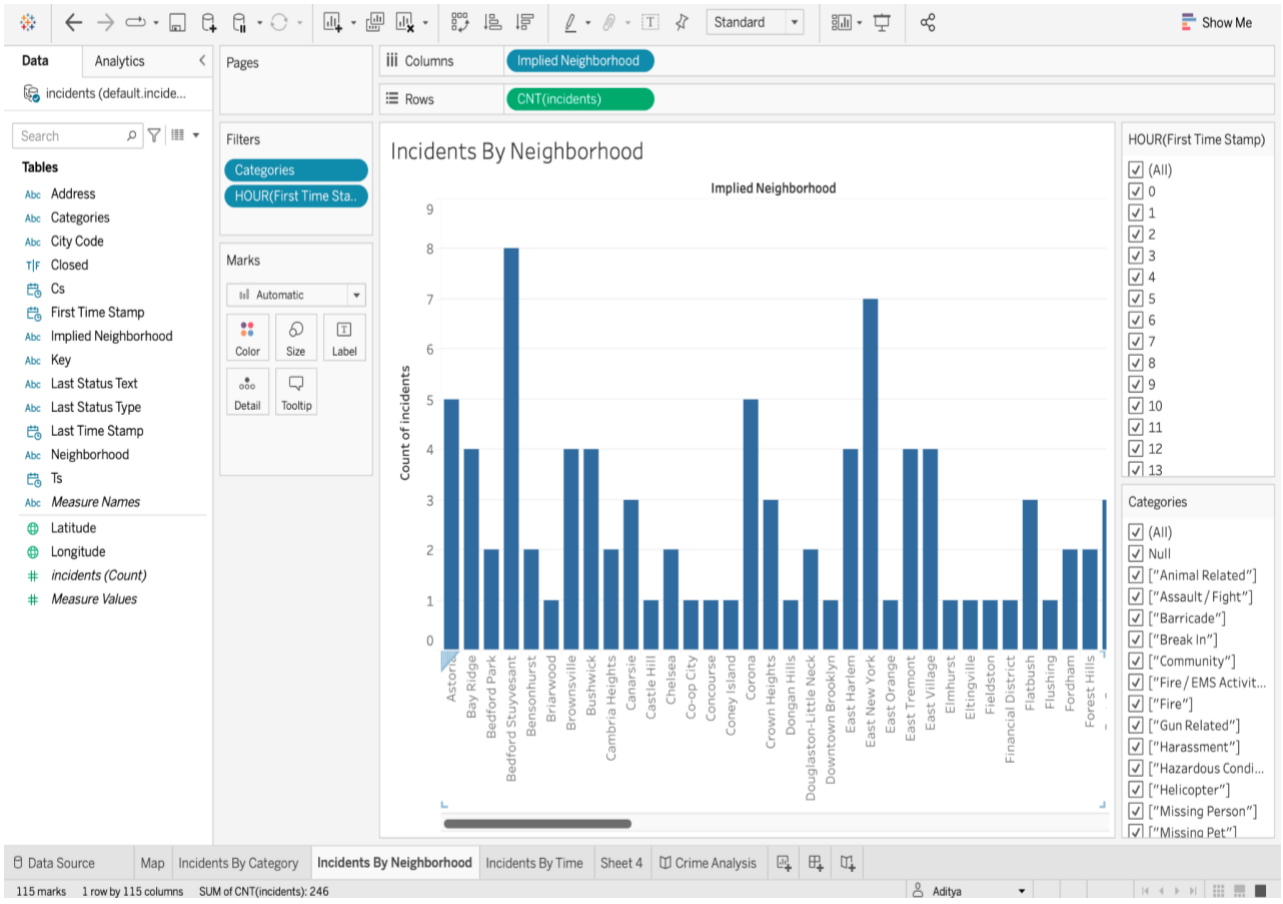
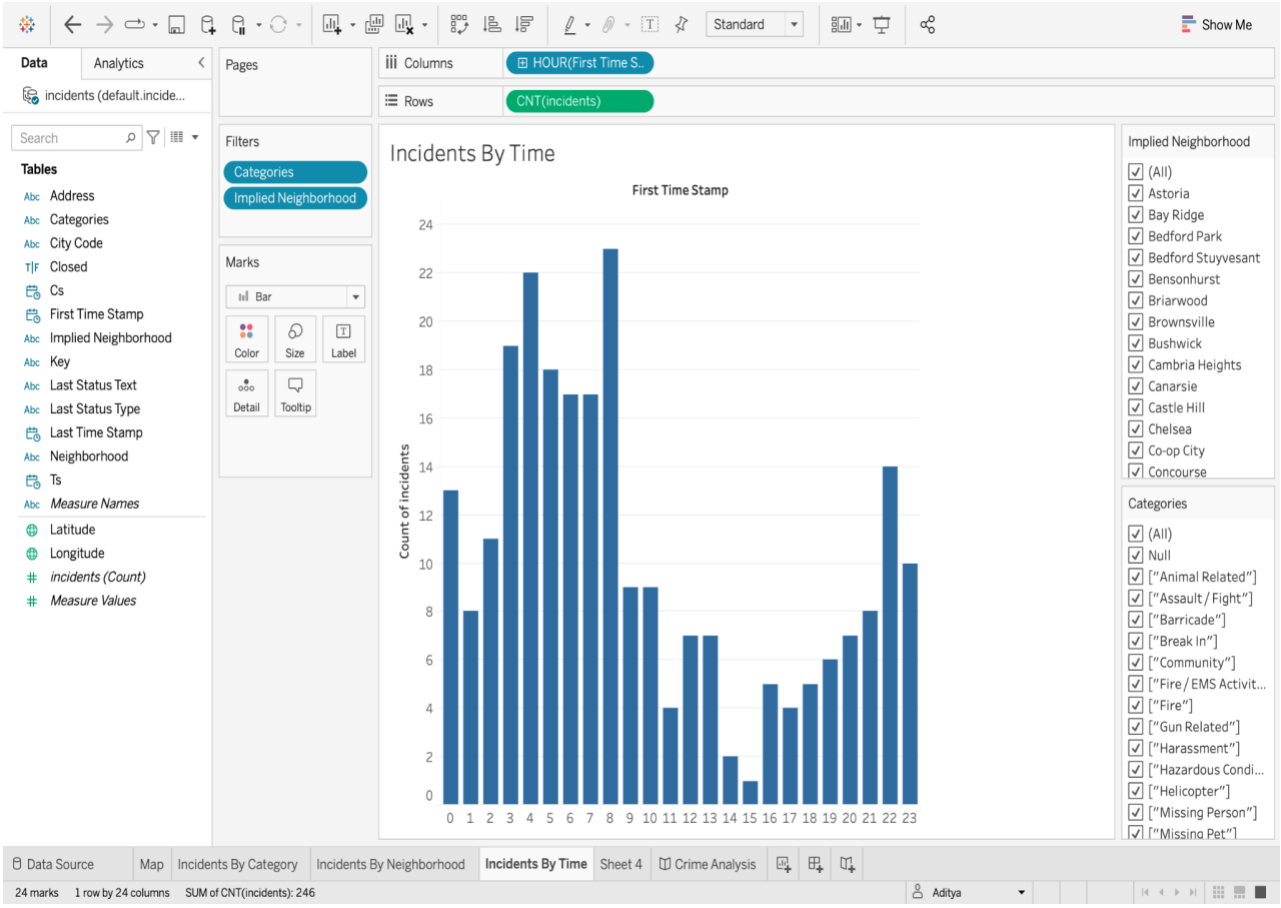


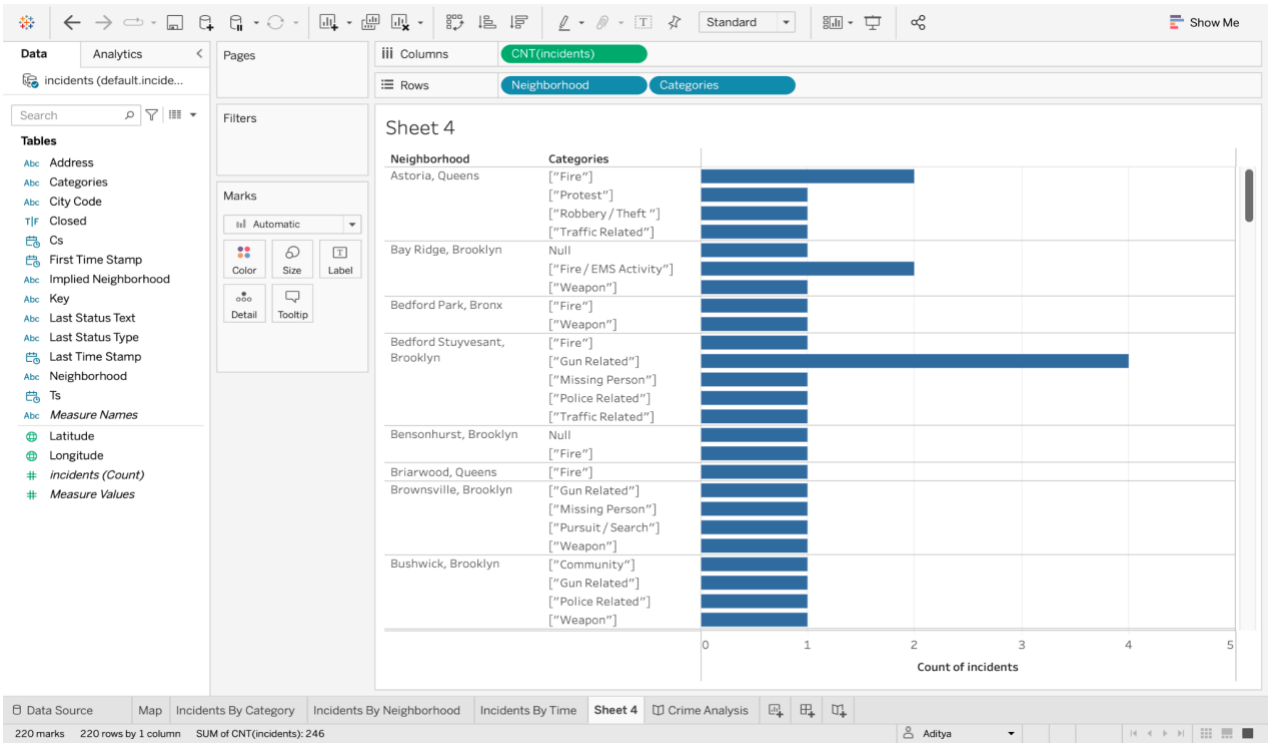
Tableau:


Show Me
Data
Analytics
incidents (default.incide...
Search
Tables
Address
Categories
City Code
Closed
Cs
First Time Stamp
Implied Neighborhood
Key
Last Status Text
Last Status Type
Last Time Stamp
Neighborhood
Ts
Measure Names
Latitude
Longitude
incidents (Count)
Measure Values
Pages
Columns
AVG(Longitude)
Rows
AVG(Latitude)
Filters
Implied Neighborhood
Categories
HOUR(First Time Sta..
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Categories
Implied Neighb..
Address
Last Time Sta..
Last Status Text
Last Status Ty..
Map
Implied Neighborhood
(All)
Astoria
Bay Ridge
Bedford Park
Bedford Stuyvesant
Bensonhurst
Briarwood
Brownsville
Bushwick
Cambria Heights
Canarsie
Castle Hill
Chelsea
Co-op City
Concourse
Categories
(All)
Null
["Animal Related"]
["Assault / Fight"]
["Barricade"]
["Break In"]
["Community"]
["Fire / EMS Activit...
["Fire"]
["Gun Related"]
["Harassment"]
["Hazardous Condi...
["Helicopter"]
["Missing Person"]
["Missing Pet"]
HOUR(First Time Stamp)
(All)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Categories
Null
["Animal Related"]
["Assault / Fight"]
["Barricade"]
["Break In"]
["Community"]
["Fire / EMS Activ..
["Fire"]
["Gun Related"]
["Harassment"]
["Hazardous Con..
["Helicopter"]
["Missing Person"]
["Missing Pet"]
Address: Foster Rd, Staten Island, NY, 10309, USA
Categories: ["Police Related"]
Implied Neighborhood: Rossville
Last Status Text: A Citizen user shows video of a police unit at the scene.
Last Status Type: TEXT
Last Time Stamp: 12/9/2022 9:49:41 PM
Latitude: 40.5426
Longitude: -74.2071
Data Source
Map
Incidents By Category
Incidents By Neighborhoo
246 marks
1 row by 1 column
SUM of AVG(Longitude): -18,187.7331


Standard
Show Me
Data
Analytics
incidents (default.incide...
Search
Tables
Address
Categories
City Code
Closed
Cs
First Time Stamp
Implied Neighborhood
Key
Last Status Text
Last Status Type
Last Time Stamp
Neighborhood
Ts
Measure Names
Latitude
Longitude
incidents (Count)
Measure Values
Pages
Columns
Categories
Rows
CNT(incidents)
Filters
Implied Neighborhood
HOUR(First Time Sta..
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Incidents By Category
Categories
Count of incidents
0
5
10
15
20
25
30
35
40
Null
["Animal Related"]
["Assault / Fight"]
["Barricade"]
["Break In"]
["Community"]
["Fire / EMS Activity"]
["Fire"]
["Gun Related"]
["Harassment"]
["Hazardous Condition"]
["Helicopter"]
["Missing Person"]
["Missing Pet"]
["Police Related"]
["Protest"]
["Pursuit / Search"]
["Rescue"]
["Robbery / Theft"]
["Traffic Related"]
["Weapon"]
HOUR(First Time Stamp)
(All)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Implied Neighborhood
(All)
Astoria
Bay Ridge
Bedford Park
Bedford Stuyvesant
Bensonhurst
Briarwood
Brownsville
Bushwick
Cambria Heights
Canarsie
Castle Hill
Chelsea
Co-op City
Concourse
Data Source
Map
Incidents By Category
Incidents By Neighborhood
Incidents By Time
Sheet 4
Crime Analysis
21 marks
1 row by 21 columns
SUM of CNT(incidents): 246
Aditya


Standard
Show Me
Data
Analytics
incidents (default.incide...
Search
Tables
Address
Categories
City Code
Closed
Cs
First Time Stamp
Implied Neighborhood
Key
Last Status Text
Last Status Type
Last Time Stamp
Neighborhood
Ts
Measure Names
Latitude
Longitude
incidents (Count)
Measure Values
Pages
Columns
Implied Neighborhood
Rows
CNT(incidents)
Filters
Categories
HOUR(First Time Sta..
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Incidents By Neighborhood
Implied Neighborhood
Count of incidents
0
1
2
3
4
5
6
7
8
9
Astoria
Bay Ridge
Bedford Park
Bedford Stuyvesant
Bensonhurst
Briarwood
Brownsville
Bushwick
Cambria Heights
Canarsie
Castle Hill
Chelsea
Co-op City
Concourse
Coney Island
Corona
Crown Heights
Dongan Hills
Douglaston-Little Neck
Downtown Brooklyn
East Harlem
East New York
East Orange
East Tremont
East Village
Elmhurst
Eltingville
Fieldston
Financial District
Flatbush
Flushing
Fordham
Forest Hills
HOUR(First Time Stamp)
(All)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Categories
(All)
Null
["Animal Related"]
["Assault / Fight"]
["Barricade"]
["Break In"]
["Community"]
["Fire / EMS Activit...
["Fire"]
["Gun Related"]
["Harassment"]
["Hazardous Condi...
["Helicopter"]
["Missing Person"]
["Missing Pet"]
Data Source
Map
Incidents By Category
Incidents By Neighborhood
Incidents By Time
Sheet 4
Crime Analysis
115 marks
1 row by 115 columns
SUM of CNT(incidents): 246
Aditya


Data
Analytics
incidents (default.incide...
Search
Tables
Address
Categories
City Code
Closed
Cs
First Time Stamp
Implied Neighborhood
Key
Last Status Text
Last Status Type
Last Time Stamp
Neighborhood
Ts
Measure Names
Latitude
Longitude
incidents (Count)
Measure Values
Pages
Filters
Categories
Implied Neighborhood
Marks
Bar
Color
Size
Label
Detail
Tooltip
Columns
HOUR(First Time S..
Rows
CNT(incidents)
Incidents By Time
First Time Stamp
Count of incidents
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Implied Neighborhood
(All)
Astoria
Bay Ridge
Bedford Park
Bedford Stuyvesant
Bensonhurst
Briarwood
Brownsville
Bushwick
Cambria Heights
Canarsie
Castle Hill
Chelsea
Co-op City
Concourse
Categories
(All)
Null
["Animal Related"]
["Assault / Fight"]
["Barricade"]
["Break In"]
["Community"]
["Fire / EMS Activit...
["Fire"]
["Gun Related"]
["Harassment"]
["Hazardous Condi...
["Helicopter"]
["Missing Person"]
Standard
Show Me
Data Source
Map
Incidents By Category
Incidents By Neighborhood
Incidents By Time
Sheet 4
Crime Analysis
24 marks
1 row by 24 columns
SUM of CNT(incidents): 246
Aditya


Data
Analytics
incidents (default.incide...
Search
Tables
Address
Categories
City Code
Closed
Cs
First Time Stamp
Implied Neighborhood
Key
Last Status Text
Last Status Type
Last Time Stamp
Neighborhood
Ts
Measure Names
Latitude
Longitude
incidents (Count)
Measure Values
Pages
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Columns
CNT(incidents)
Rows
Neighborhood
Categories
Standard
Show Me
Sheet 4
Neighborhood
Categories
Astoria, Queens
["Fire"]
["Protest"]
["Robbery / Theft "]
["Traffic Related"]
Bay Ridge, Brooklyn
Null
["Fire / EMS Activity"]
["Weapon"]
Bedford Park, Bronx
["Fire"]
["Weapon"]
Bedford Stuyvesant, Brooklyn
["Fire"]
["Gun Related"]
["Missing Person"]
["Police Related"]
["Traffic Related"]
Bensonhurst, Brooklyn
Null
["Fire"]
Briarwood, Queens
["Fire"]
Brownsville, Brooklyn
["Gun Related"]
["Missing Person"]
["Pursuit / Search"]
["Weapon"]
Bushwick, Brooklyn
["Community"]
["Gun Related"]
["Police Related"]
["Weapon"]
0
1
2
3
4
5
Count of incidents
Data Source
Map
Incidents By Category
Incidents By Neighborhood
Incidents By Time
Sheet 4
Crime Analysis
220 marks
220 rows by 1 column
SUM of CNT(incidents): 246
Aditya