University of Milano-Bicocca
Department of Computer Science, Systems and Communication
Master's Degree in Data Science

Motor Vehicle Collisions Analysis
New York City – 2018

Course: Data Management
Teacher: Andrea Maurino

Project Members:

Giulia Saresini (Mat. 864967)
Sara Nava (Mat. 870885)

Academic Year 2023-2024

Table of Contents

1. Introduction

Our project focuses on **analysing vehicle collisions that occurred in New York City in 2018**, with particular attention to the dynamics leading to the collisions, the involved parties, and the surrounding spatiotemporal conditions. To access this wide range of information, we **integrated data from four different sources**, each providing specific details on each collision, the involved vehicles and drivers, the meteorological conditions at the time of the collision and the day of the week when it occurred.

Our **research questions** cover various aspects of the collisions:

A. Observing the **distribution of collisions across the five boroughs** of New York City (Brooklyn, Manhattan, Queens, Staten Island, Bronx).
B. Investigating the **pre-collision dynamics** to identify situations that may lead to collisions more frequently.
C. Evaluating which **driving behaviours** may increase the probability of a collision.
D. Exploring the **meteorological conditions** in which collisions primarily occur.
E. Verifying if there are certain days of the **week or holidays when collisions occur more frequently**, and if there is a variation compared to the daily average.
F. Analysing if there are **specific moments of the day** when there is a higher concentration of collisions.
G. Analyzing the **severity of each collision** by considering the number of injured and deceased individuals involved.
H. Examining the **driver's license status** of the involved drivers to determine if it may influence the likelihood of a collision.
I. Examining whether the **number of passengers in the vehicle** may contribute to the collision, hypothesizing that they may distract the driver.

We have organized our work into a series of distinct steps, each of which performs a clear and defined function:

1. **Data Acquisition**: Data has been extracted from respective sources through API queries or Web Scraping, ensuring accurate and complete collection of necessary information.
2. **Data Cleaning**: Each dataset has undergone a targeted cleaning process to eliminate impurities, such as missing values or formatting errors, to ensure the reliability and consistency of the analysed data.
3. **Database Construction**: After analysing the available data, a suitable structure has been designed to host them, using the most appropriate tools for effective database implementation. Subsequently, the data has been loaded and organized within this structure to facilitate consultation and analysis.
4. **Data Integration**: All collected data has been integrated and correlated with each other to create a unified and cohesive source of information, thus allowing for a comprehensive and detailed view of the analysed context.
5. **Data Quality Assessment**: Following the completion of cleaning and integration processes, a careful assessment of data quality has been conducted to ensure the reliability and accuracy of subsequent analyses.
6. **Data Querying**: Targeted data queries have been executed to identify relevant information and obtain answers to initial research questions, thus providing a comprehensive and exhaustive overview of the analysed phenomenon.

Each phase has been addressed in a dedicated chapter within this report, providing a detailed description of each step performed.

Allocation of work:

- **Data Acquisition**:
  - Giulia: API querying for the Collisions dataset and a portion of web scraping necessary to feed another API querying.
  - Sara: API querying for the Vehicles and Weather datasets, and the web scraping portion to extract the Calendar dataset.
- **Data Cleaning**: Giulia
- **Data Storage and Integration**: Sara
- **Data Quality**: Sara
- **Data Querying**: Giulia
- **Report and PowerPoint**: Compilation of the report and PowerPoint for ourselves respective parts of expertise.

2. Data Acquisition

To obtain the fundamental data for our analyses, we adopted a series of data extraction techniques. Specifically, we used **API queries** and, when open APIs were not available, we opted for **web scraping**. In the following paragraphs, we provide a detailed description of the extraction process for each identified data source.

2.a Collisions

The **Collisions dataset** (2) contains **details on the crash event**, where each row represents a crash event. The Motor Vehicle Collisions data tables contain information from all police reported motor vehicle collisions in NYC. The police report (MV104-AN) is required to be filled out for collisions where someone is injured or killed, or where there is at least $1000 worth of damage.
These pieces of information can be found on the **NYC Open Data** website (1), a platform that hosts a vast collection of data related to the city of New York. The data is accessible in both .csv and .json formats, but we chose to retrieve data through API queries to the specific endpoint provided by **NYC Open Data API (3)** (powered by Socrata). This choice was motivated by the desire to obtain the data in a customized manner, selecting only the information of interest rather than downloading the entire dataset.

The dataset offers a wide range of information, from which we extracted only the relevant ones, including:

- **collision_id**: a unique identifier generated by the system for each incident.
- **borough**: the borough of New York City where the incident occurred.
- **zip_code**: the postal code of the location where the incident occurred.
- **latitude**: latitude coordinate in the WGS 1984 global coordinate system, expressed in decimal degrees (EPSG 4326).
- **longitude**: longitude coordinate in the WGS 1984 global coordinate system, expressed in decimal degrees (EPSG 4326).

- **crash_date**: the date of the incident in floating timestamp format (e.g., "2014-10-13T00:00:00.000").
- **crash_time**: the time of the incident, separated from the date and expressed in textual format (e.g., "1:45").
- **number_of_persons_injured**: the number of people injured in the incident.
- **number_of_persons_killed**: the number of people killed in the incident.

In order to make the API request and export the data, we had to import some libraries, among them, we list only the most important ones:

- **Socrata** [4]: A Python library that allows access and interaction with public data APIs provided by Socrata platforms, such as Open Data portals of some governments and organizations, including NYC Open Data.
- **Json** [5]: A built-in Python module for working with data in JSON (JavaScript Object Notation) format.
- **ReadTimeout** [6]: This is a specific exception that is part of the Requests library, commonly used to make HTTP requests in Python. ReadTimeout is raised when a request does not receive a response within the specified maximum reading time.

Subsequently, to access the data, it was necessary to create a personal profile on the NYC Open Data platform [7] and generate an App Token with which to make requests. After creating the account and generating an App Token, we initialized a Socrata client to which we provided the website from which to access the data, the generated token, the username associated with the created profile, and the corresponding password.

To ensure the security and privacy of platform access credentials, we utilized the **input()** and **getpass()** functions (the latter belonging to the *getpass* library). This allows us to prompt the user to enter their credentials directly during the execution of the Python script, without displaying them on the screen. The entered credentials are then saved within three separate variables, ensuring the protection of sensitive data.

```
# Insert credential
from getpass import getpass
user = input('Insert your email: ')
password = getpass('Insert your password: ')
token = getpass('Insert your app token: ')

client = Socrata("data.cityofnewyork.us",
                 token,
                 username=user,
                 password=password)
```

Before making the request, we initialized some variables:

- **start_date_formatted**: The start date of the time period of interest in timestamp format. In our case, considering 2018, it was set to '2018/01/01T00:00:00'.
- **end_date_formatted:** The end date of the time period of interest in timestamp format. Therefore, it was set to '2018/12/31T23:59:59'.
- **vars_to_select:** The list of variables, separated by commas, of interest for our analyses.
- **dataset_id:** The identifying code of the dataset on the NYC Open Data platform.

Therefore, it was possible to execute the data request. In particular, we used the **get** method on the previously initialized **Socrata client** to retrieve data from the dataset identified by the **dataset_id**. Within this request, we specified the variables to select and filtered the time period of interest. This was made possible thanks to the rich query functionalities offered by Socrata's APIs, using a query

language known as **'Socrata Query Language'** or **'SoQL'** (8). As the name suggests, SoQL is closely inspired by Structured Query Language (SQL), widely used in relational database systems.

Furthermore, we structured the code so that, in case of connection errors or exceeding allowed data access times, the request is automatically retried to make the process smoother for the user.

```
max_attempts = 5 # Maximum number of attempts
attempts = 0 # Initialize attempts counter

# Execute the loop until the max number of attempts is reached or the timeout error is solved
while attempts < max_attempts:
    try:
        # Retrieve data from the dataset using the Socrata client
        results = client.get(dataset_id,
                             select=vars_to_select,
                             where=f"crash_date >= '{start_date_formatted}' AND crash_date <=
                              '{end_date_formatted}'",
                             limit=600000)
        # Exit the loop if the query is successful
        break
    except ReadTimeout:
        # Handle the timeout error
        print("Read timeout. Retrying...")
        attempts += 1  # Increment the attempts counter

if attempts == max_attempts:
    print(f"Maximum number of attempts ({max_attempts}) reached. Unable to complete the
query. Try again.")
```

Finally, the data was exported into a .csv file.

2.b Vehicles

The **Vehicles** dataset (9), similar to the Collision dataset, is a resource from the **NYC Open Data** website providing details about the **vehicles involved in the incidents**, with each row corresponding to a single vehicle involved in an incident. Among the information available in this source, we have identified the following attributes relevant for our analyses:

- **unique_id:** A unique code associated with each record in the table, representing the specific details of each vehicle involved in a particular incident. Note that this is not the vehicle's identification code, as a vehicle could be involved in multiple incidents during the same year.
- **collision_id:** The identification code of the incident in which the vehicle was involved.
- **pre_crash:** The driving action performed before the incident, such as "Going straight", "Making right turn", "Passing", "Backing", etc.
- **vehicle_occupants**: The number of passengers inside the vehicle during the incident.
- **contributing_factor_1**: A factor related to the driver's behaviour that might have contributed to the incident, such as "Driver Inattention/Distraction".

- **driver_license_status**: Indicates the driver's license status, whether they possess it, do not possess it, or have a learner's permit (for example, for those who are learning to drive but have not yet obtained a license).

Similarly to the Collision dataset, data for the Vehicles dataset was also obtained through **API queries** to the specific **endpoint** provided by **NYC Open Data** (10). Therefore, the structure of the code is analogous to that used previously. The only variation made concerned the variables specified within the SELECT clause of the SoQL query and the identification code of the data source being accessed.

2.c Calendar

We decided to explore which **days of the week might be more frequently associated with accidents** and if there was a **variation in the number of accidents during holidays** compared to regular days of the year. To do this, we chose to acquire information related to the 2018 calendar in the United States, in order to obtain a dataset containing, for each day of the year, the respective day of the week and an indication of whether it was a national holiday or not.

To tackle this challenge flexibly, we opted to use **Web Scraping**. To achieve our goal, we identified a website hosting a US calendar (11). This site allows for viewing the entire calendar of a specific year within a single page or viewing the calendar month by month, with a dedicated page for each month, which URL is structured as follows:

https://www.timeanddate.com/calendar/custom.html?year=2018&month=1&country=1&typ=1&cols=1&display=1&df=1&hol=1

Considering the structure of the website URL, to simplify our work, we decided to focus on extracting data from a single month and then iterate the extraction process over all the months of the reference year. In fact, each month is associated to a specific number (January associated to 1, February to 2, March to 3 and so on until December to 12) so, to consider a specific month, it's simply needed to change the highlighted parameter into the web page link (**&month=monthnumber**). This approach allowed us to handle data acquisition more efficiently and better organize our code for subsequent information processing.

As it's possible to see in the next picture, the data is organized in a table with the days of the week as headers and the numbers associated with each day of the month inside the cells. Holidays are highlighted in red, while regular days appear in black.

To extract this data using Python, we imported some libraries, including:

- **requests** (12): To send HTTP requests to web servers and receive responses.
- **BeautifulSoup** (13) (or bs4): For extracting data from HTML and XML documents, useful for analyzing the structure of web pages and retrieving desired information.

As previously said, the **process** was **designed to iterate through the months of** the year **2018**.

For each month, we proceeded by taking the corresponding URL to its calendar webpage and then, we made a GET request to this webpage using the requests module and parsed the content of the page with **BeautifulSoup** to extract and manipulate the data within the HTML code.
Finally, we identified the content of the table within the HTML code of the page, using specific CSS classes as criteria for locating the table elements.

```
# Create a list of months' number
list_month = [num for num in range(1,13)]
for month in list_month:
    # URL of the webpage to be downloaded
    url_page = f"https://www.timeanddate.com/calendar/
      custom.html?year={year}&month={month}&country=1 &typ=1&cols=1&display=1&df=1&hol=1"
        # Make a GET request to the webpage
        response = requests.get(url_page)
        # Use BeautifulSoup to parse the HTML content of the webpage
        soup = BeautifulSoup(response.content, "html.parser")
        # Identify the content of the table
        table_content = soup.find_all("td", class_="cbm cba cmi")[0]
```

Then, to create the desired table, we **identified the header** and extracted the **names of the days of the week**, creating a list that **repeated the sequence** ('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat') **as many times as there were rows within the table** (e.g., for the month of January, repeating the sequence five times). We also converted the abbreviated names of the days of the week to their full names for clarity.

```
    …
    # Identify the dimensions
    nrow_table = len(table_content.find_all("tr")) # each row of the table is identified with "tr",
    including the header
    ncol_table = len(table_content.find_all("thead")[0].find_all("td"))
    # Create a list containing the weekdays
    weekdays = (table_content.find_all("thead")[0].find_all("td"))*(nrow_table-1)
    weekdays = [weekdays[x].text for x in range(len(weekdays))]
    # Re-code the weekdays
    weekdays = [recode_days[value] for value in weekdays if value in recode_days]
```

Next, we identified the holidays by recognizing them for their red color, then searched for those table cells (identified by `<div>`) that were characterized by the class `"ccd co1"` (red text color code), extracted their content (the day of the month number), and created a list called "find_holidays" to store these days.

```
    …
    # Create a list containing the holidays of the month
    find_holidays = table_content.find_all("div", class_="ccd co1")
    holidays = [int(hol.text) for hol in find_holidays]
```

For example, for January, we obtained:

```
find_holidays = [1, 15]
```

Then, we created a list called "numb" containing all the values present within the table cells. If a cell was empty, the extracted value was "\xa0", otherwise it was the corresponding day number.
For example, for the month of January, we obtained two lists as follows:

```
weekdays = ['Sunday', 'Monday', 'Tuesday', ...]
numb = ['\xa0', 1, 2, ...]
```

Subsequently, we created a list called "Flag_Holiday", where the value is True if the corresponding day number is present in the "find_holidays" list, otherwise False.
For example, for January, we get:

```
find_holidays = [1, 15]
Flag_Holiday = [False, True, False, ...].
```

Finally, we created a list called "date", where the elements are None if the corresponding element of "numb" is '\xa0', otherwise we created the date using the datetime( ) function, where year was always set to 2018, month took the value of the reference month, and day took the number present in the "numb" list.

```
…
# Create a list containing the number of days of the month
    numb = [0]*((nrow_table-1)*ncol_table)
    numb = [table_content.find_all("td")[iter+ncol_table].text for iter in range(len(numb))]
    # Format the values, create the date, and the Flag_Holiday field (if the day is a holiday or
not)
    date = numb.copy()
    Flag_Holiday = numb.copy()
    for i in range(len(numb)):
        if numb[i]=="\xa0": # Empty cell
            date[i]=None
        else:
            numb[i] = int(numb[i]) # Cell containing the day number
            date[i] = datetime(year=year, month=month, day=numb[i]).strftime('%Y-%m-%d')
            if numb[i] in holidays:
                Flag_Holiday[i] = True
            else:
                Flag_Holiday[i] = False
```

For example, for January, we get these final results:

```
date = [None, '2018-01-01', '2018-01-02', …]
weekdays = ['Sunday', 'Monday', 'Tuesday', …]
Flag_Holiday = [False, True, False, ...]
```

After obtaining the "date", "weekdays", and "Flag_Holiday" lists, we created a pandas DataFrame and removed the None values to eliminate empty cells.

This process was repeated for each month, and the results were concatenated until the complete calendar of the year was obtained. This is the result:

	Date	Day	Flag_Holiday
0	2018-01-01	Monday	True
1	2018-01-02	Tuesday	False
2	2018-01-03	Wednesday	False
3	2018-01-04	Thursday	False
4	2018-01-05	Friday	False
...	...	...	...
26	2018-12-27	Thursday	False
27	2018-12-28	Friday	False
28	2018-12-29	Saturday	False
29	2018-12-30	Sunday	False
30	2018-12-31	Monday	False

365 rows × 3 columns

Then we exported data into a .csv file.

2.d Weather

In order to **integrate weather information with the incidents**, and having the geographical coordinates of the incident location, the postal code of the location, borough, date, and time of the incident, we identified an API that allowed us to utilize this information present in the 'Collision' dataset to perform as accurate a query as possible.

The chosen **API** was **Open-Meteo** (14), a free API that does not require any key and provides free access to data for non-commercial purposes. It is an API that gives users access to various information, including the **historical weather data** (15). By inputting valid geographical coordinates and a time period (data is available from 1940 to present), it is possible to extract both **hourly and daily weather information**. To make requests, it is necessary to be aware of the terms and conditions regarding the allowed limits(16).

To obtain **meteorological information at the time of the incident**, we decided to focus on hourly information. However, considering the high number of incidents (over 200,000) and the need to extract 8,760 rows (24 hours per 365 days, total days in 2018) for each incident, we would have had to handle a considerable amount of data, over 1 billion total pieces of information. This approach would have been redundant and inefficient, as precise meteorological data for the exact point of the incident is not necessary. It is **sufficient to consider the weather conditions of the surrounding area where the incident occurred**. To optimize the process, we decided to **use the postal code as a reference information**. This allowed us to **make a request by postal code instead of specific coordinates**, thus reducing the complexity and amount of data to manage. This way, we could detach the search from the 'Collisions' dataset and conduct an independent query, still maintaining an acceptable level of accuracy for our meteorological analyses.

We proceeded as follows.
From the **NYC Open Data government platform** (1), from which we could access information regarding incidents and vehicles involved, we **downloaded** a **CSV file** containing information on **the zip codes of New York City** (there are a total of 248), where each row is associated with a single zip code (17). Needing only a single list of postal codes, it was sufficient to consider the relevant field, ignoring the rest of the information which, for our purposes, is redundant.

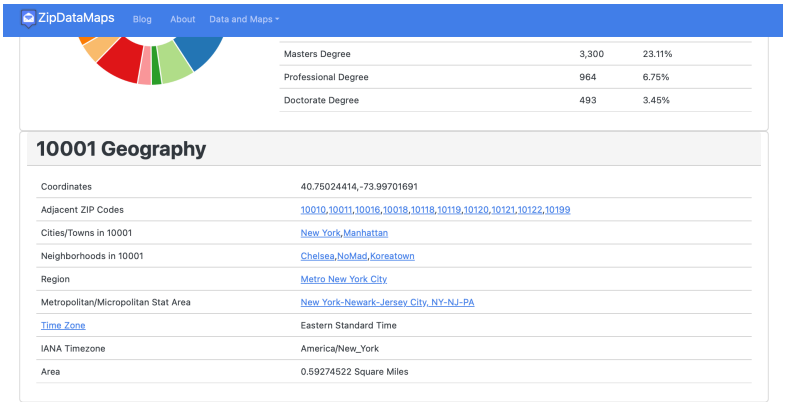
Since the Open-Meteo API requires geographical coordinates as input and we only had the postal code, we sought a **solution to obtain approximate coordinates associated with each postal code**. To this end, we performed Web Scraping of a website (**ZipDataMaps** (18)) that provides this information.

Using the same libraries mentioned earlier in the *subsection 2.c* dedicated to the Calendar dataset, we developed a code that allowed us to obtain this information.

The website in question features a URL structured as follows (let's take the postal code 10001 as an example):

https://www.zipdatamaps.com/10001

The information of interest can be easily found in the section of the page where the word "Coordinates" appears.



To obtain this information for all 248 available postal codes, we implemented a for loop. In this loop, **we varied the postal code within the highlighted link**, allowing us to visit the pages related to each postal code.

```
# Read the dataset containing
nyc_zipcodes_data = pd.read_csv('Broadband_Adoption_and_Infrastructure_by_Zip_Code_20240209.csv')
# Filter the necessary data and remove duplicates
zip_codes = list(nyc_zipcodes_data['Zip Code'].dropna().drop_duplicates())
zip_codes = [int(x) for x in zip_codes]
# Create an empty dataframe to store the results
result_df = pd.DataFrame(columns=["Zip Code", "Latitude", "Longitude"])
# Loop through the zip codes in the list
for zip_code in zip_codes:
...
```

To extract the desired information, we structured the code within the loop as follows. We constructed the URL using the current ZIP code. Subsequently, we made a GET request to retrieve the corresponding web page using the BeautifulSoup library to parse the HTML content of the page. Since the page is structured in a tabular form, we searched for all `<td>` elements that identify the different rows of the table.

```
…
url = f"https://www.zipdatamaps.com/{zip_code}"
page = requests.get(url)
soup = BeautifulSoup(page.content, "html.parser")
coordinates = soup.find_all('td')
```

We then initialized a variable to track the index of the element containing the coordinates, determining the index based on the position of the element containing the string "Coordinates". If the coordinates are not present for a certain ZIP code, a warning message is printed.
Finally, we extracted the text from the element following the "Coordinates" string, which represents the pair of coordinates separated by a comma. These coordinates were split into latitude (lat) and longitude (lon) and converted into float format.

```
…
# Find the index of the "Coordinates" element in the list
    coordinates_index = None
    for i, td in enumerate(coordinates):
        if "Coordinates" in str(td):
            coordinates_index = i
            break
    # If the index of "Coordinates" is found, take the next element and split it
    if coordinates_index is not None and coordinates_index + 1 < len(coordinates):
        coordinates_text = coordinates[coordinates_index + 1].text.strip()
        # Split the coordinates into latitude and longitude
        lat, lon = map(float, coordinates_text.split(','))
        # Create a temporary dataframe for the current element
        temp_df = pd.DataFrame({"Zip Code": [zip_code], "Latitude": [lat], "Longitude": [lon]})
        # Concatenate the temporary dataframe with the main dataframe
        result_df = pd.concat([result_df, temp_df], ignore_index=True)
    else:
        print(f"Coordinates not found for the zip code {zip_code}")
```

The obtained results were concatenated into a pandas dataframe with three columns, containing the zip_code, latitude, and longitude associated with each postal code.

```
Coordinates not found for the zip code 83
Coordinates not found for the zip code 10047
Coordinates not found for the zip code 10048
Coordinates not found for the zip code 10096
Coordinates not found for the zip code 10097
Coordinates not found for the zip code 10196
     Zip Code   Latitude  Longitude
0       10001  40.750244 -73.997017
1       10002  40.713882 -73.985924
2       10003  40.731991 -73.988869
3       10004  40.694939 -74.016922
4       10005  40.706150 -74.008568
..        ...        ...        ...
237     11691  40.600616 -73.762558
238     11692  40.595150 -73.796173
239     11693  40.611607 -73.815712
240     11694  40.575130 -73.851662
241     11697  40.557446 -73.913467

[242 rows x 3 columns]
```

To make the API request, we first imported the main libraries required by the API, which include:

- **openmeteo_requests** (19): This is an API client based on the `requests` library that allows us to extract information from Open-meteo.
- **requests_cache** (20): This is a persistent HTTP cache that provides a simple way to achieve better performance with the Python requests library.
- **retry_requests** (21): This library configures the requests session to retry in case of failed requests due to connection errors, timeouts, specific HTTP response codes (usually 5XX), and 30X redirects.

Before making the request, we defined the parameters to be passed as input to the API. Firstly, we defined the **start and end date**, set respectively to **'2017-12-31'** and **'2018-12-31'**. The reason we

started from the last day of 2017 is because the API returns results from 05:00:00 on the first day passed in input to 04:00:00 on the day following the last day passed in input. Therefore, to not lose the first five hours of the morning of '2018-01-01', we start from the previous day and, before exporting the data, we remove the additional information (thus the data that the API will download for the dates '2017-12-31' and '2019-01-01').

Additionally, we selected the variables of interest, which are as follows:

- **temperature_2m**: Air temperature at 2 meters above ground in °C.
- **relative_humidity_2m**: Relative humidity at 2 meters above ground in %.
- **rain**: Only liquid precipitation of the preceding hour including local showers and rain from large scale systems in mm.
- **snowfall**: Snowfall amount of the preceding hour in centimeters.
- **cloud_cover**: Total cloud cover as an area fraction in %.

Considering the number of available coordinates and the required time frame, based on **how the weight of the number of requests to the API is calculated** (22), it was necessary to structure the request in such a way that the first half of the coordinates was initially passed as input to the Endpoint, and after 90 seconds (using the `sleep()` function from the `time` library), the second half of the coordinates was passed, in order to **not exceed the request limit per minute and avoid receiving an error**.

```
latitudes = list(result_df['Latitude'])
longitudes = list(result_df['Longitude'])
# Split coordinates into two different lists
split_point = (len(latitudes)+1)//2
subsets_lat = [latitudes[:split_point], latitudes[split_point:]]
subsets_long = [longitudes[:split_point], longitudes[split_point:]]
# Setup the Open-Meteo API client with cache and retry on error
cache_session = requests_cache.CachedSession('.cache', expire_after = -1)
retry_session = retry(cache_session, retries = 5, backoff_factor = 0.2)
openmeteo = openmeteo_requests.Client(session = retry_session)
# The order of variables in hourly or daily is important to assign them correctly below
url = "https://archive-api.open-meteo.com/v1/archive"
# Set the time interval
start_date = '2017-12-31'
stop_date = '2018-12-31'
import time
responses = [0]*len(subsets_lat)

# Repeat the request for the two lists of coordinates
for i in range(len(subsets_lat)):
  # Define parameters to make the API request
  params = {
      "latitude": subsets_lat[i],
      "longitude": subsets_long[i],
      "hourly": ["temperature_2m", "relative_humidity_2m", "rain", "snowfall",
"cloud_cover"],
      "timezone": "auto",
```

```
    "start_date": start_date,
    "end_date": stop_date
}
# Make the API request
response = openmeteo.weather_api(url, params=params)
responses[i] = response
time.sleep(90)
```

The data was extracted from responses, following how the documentation (14) suggests to do, and then inserted into a pandas data frame and exported to a .csv file.

Here's how the data frame looks like:

	date	temperature_2m	relative_humidity_2m	rain	snowfall	cloud_cover	zip_code
0	2018-01-01 00:00:00	-11.676	48.092360	0.0	0.0	0.0	10001
1	2018-01-01 01:00:00	-11.876	48.238712	0.0	0.0	0.0	10001
2	2018-01-01 02:00:00	-12.226	48.972702	0.0	0.0	0.0	10001
3	2018-01-01 03:00:00	-12.526	49.521194	0.0	0.0	0.0	10001
4	2018-01-01 04:00:00	-12.826	51.399673	0.0	0.0	0.0	10001

3. Data Cleaning

The Data Cleaning process involved the datasets: Collisions, Vehicles, and Weather.

3.a Collisions

The dataset concerning the incidents' perimeter in NY in 2018 is **Collisions**. Before its Data Cleaning process, which also involved **imputing some missing data**, the dataset consisted of 231.564 rows and was structured as follows:

collision_id INT	borough VARCHAR	zip_code INT	latitude FLOAT	longitude FLOAT	crash_date TIMESTAMP	crash_time VARCHAR(2	number_of_persons_injured INT	number_of_persons_killed INT
3820157	MANHATTAI	10025	40.8018	-73.96108	2018-01-01T00:00:00.0(	4:16	1	0
3818846	QUEENS	11373	40.743973	-73.8851	2018-01-01T00:00:00.0(	20:30	0	0
3820776			40.72143	-73.892746	2018-01-01T00:00:00.0(	23:41	0	0
3818947	MANHATTAI	10025	40.80174	-73.96477	2018-01-01T00:00:00.0(	15:30	0	0
3820540			40.666225	-73.80086	2018-01-01T00:00:00.0(	15:00	0	0
3820645	QUEENS	11354	40.763073	-73.816345	2018-01-01T00:00:00.0(	12:10	0	0
3819261	BRONX	10459	40.820305	-73.89083	2018-01-01T00:00:00.0(	18:35	0	0
3820853	BROOKLYN	11207	40.65892	-73.889824	2018-01-01T00:00:00.0(	13:50	0	0
3819256	BROOKLYN	11212	40.662277	-73.91078	2018-01-01T00:00:00.0(	1:37	0	0
3819167			40.5744	-74.099304	2018-01-01T00:00:00.0(	11:50	0	0
3818793			40.665276	-73.82955	2018-01-01T00:00:00.0(	10:15	0	0
3818787	QUEENS	11428	40.722363	-73.736885	2018-01-01T00:00:00.0(	12:50	1	0

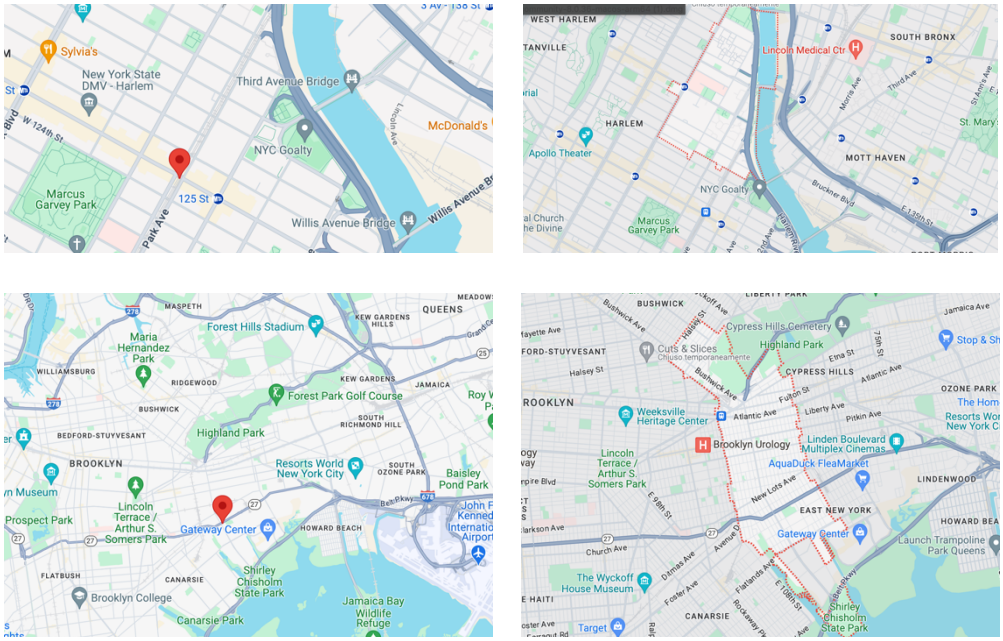
The main task performed on this dataset was the **imputation of districts** ('boroughs') necessary to answer some of our research questions and **missing 'zip_codes'** required for the join with weather data. We were able to impute this data using the Python library **geopy**. Before any imputation, the dataset was filtered to work only with the portion of data that actually needed to be imputed, resulting in a dataset with 71.676 records. To impute the data, it was necessary to have the 'Latitude' and 'Longitude' of the collision. Although it would have been possible to infer the borough from the 'zip_code' using Python functions we observed that the missing values appeared in pairs in the dataset making this alternative imputation not feasible:

collision_id INT	borough VARCHAR	zip_code INT	latitude FLOAT	longitude FLOAT	crash_date TIMESTAMP	crash_time VARCHAR(2	number_of_persons_injured INT	number_of_persons_killed INT
3820776			40.72143	-73.892746	2018-01-01T00:00:00.0(	23:41	0	0
3820540			40.666225	-73.80086	2018-01-01T00:00:00.0(	15:00	0	0
3819167			40.5744	-74.099304	2018-01-01T00:00:00.0(	11:50	0	0
3818793			40.665276	-73.82955	2018-01-01T00:00:00.0(	10:15	0	0
3827479			40.75837	-201.23706	2018-01-01T00:00:00.0(	0:51	0	0
3818998			40.762234	-73.98987	2018-01-01T00:00:00.0(	11:00	0	0
3821214			40.73635	-73.97502	2018-01-01T00:00:00.0(	3:30	0	0
3831289			40.67308	-73.91124	2018-01-01T00:00:00.0(	16:30	0	0
3819572			40.744465	-73.77179	2018-01-01T00:00:00.0(	2:10	1	0

Since Latitude and Longitude were essential, all records with **missing values for 'Latitude' and 'Longitude' were removed** from the filtered dataset called 'collision_impute'. After this step, the function 'coordinate_to_borough_and_zipcode' (which accepts a pair of geographical coordinates (latitude and longitude) and returns the respective district (borough) and postal code (zipcode)) was implemented. To do this, a geocoding service (Nominatim) was used to translate the coordinates into an address readable by humans (23). Subsequently, we used a rule to convert the district names returned by Nominatim into the names used in the dataset to ensure **Syntactic Accuracy** of the field:

```
# Map desired borough names
borough_map = {
    'Queens County': 'QUEENS',
    'Manhattan': 'MANHATTAN',
    'The Bronx': 'BRONX',
    'Brooklyn': 'BROOKLYN',
    'Staten Island': 'STATEN ISLAND',
    'Queens': 'QUEENS',
    'Richmond County': 'STATEN ISLAND',
    'Kings County': 'BROOKLYN'
}
```

To complete the imputation operation, it took approximately 9.5 hours.
Sample checks were performed to ensure that Nominatim provided reliable results by imputing data for 1000 cases where both the district and zip_code were already known. Some records were identified where the original zip_code and the zip_code returned by the function were different. To investigate the reasons behind this discrepancy we used **Google Maps** (24). We observed that the zip_codes associated with these 'misclassified' points were very geographically close to the correct zip_code; they were always adjacent zip_codes:



In the left images, the coordinates related to the location of the incident are shown, while on the right, the district found by the function is displayed.

Considering our requirement for precise imputation solely regarding the borough, as we decided to **analyze data by district** (with borough classification in the test sample appearing flawless), because the zip_codes of the collisions' location were exclusively utilized for weather data integration, we presumed that neighboring zip_codes would yield similar meteorological conditions, so we deemed the results satisfactory.

After imputing the missing zip_codes and boroughs, we proceeded to **check if the function failed** to retrieve results for any records. There were cases where the function indeed returned no results, as in the following records:

collision_id INT	borough VARCHAR	zip_code INT	latitude FLOAT	longitude FLOAT	crash_date TIMESTAMP	crash_time VARCHAR(2	number_of_persons_injured INT	number_of_persons_killed INT
3827479			40.75837	-201.23706	2018-01-01T00:00:00.0(	0:51	0	0
3822610			40.75837	-201.23706	2018-01-04T00:00:00.0(	10:46	0	0
3825619			40.75837	-201.23706	2018-01-09T00:00:00.0(	17:22	0	0
3826894			40.75837	-201.23706	2018-01-11T00:00:00.0(	9:15	0	0
3828824			40.75837	-201.23706	2018-01-12T00:00:00.0(	19:30	0	0
3829449			40.75837	-201.23706	2018-01-14T00:00:00.0(	6:45	0	0
3829655			40.75837	-201.23706	2018-01-16T00:00:00.0(	8:03	0	0
3832438			40.75837	-201.23706	2018-01-19T00:00:00.0(	16:32	0	0
3836227			40.75837	-201.23706	2018-01-29T00:00:00.0(	4:15	0	0
3849177			40.75837	-201.23706	2018-02-17T00:00:00.0(	18:50	0	0

However, as can be seen, the lack of results was not due to a malfunction of Nominatim but rather due to **incorrect data**. These records were removed from the dataset. Subsequently, the dataset was merged with the one containing already corrected values, and the 'Latitude' and 'Longitude' columns were dropped as they were unnecessary for the final analysis, resulting in a dataset with 220.679 rows.

On this dataset, we performed the following operations: the 'crash_date' column (originally in datetime format) and the 'crash_time' columns were modified to match the formats of the date and time columns present in the weather dataset (necessary for performing joins). We created a new column called 'day_period' to categorize the 'crash_time' into 'Morning', 'Afternoon', 'Evening', and 'Night' (25), and the values in the columns 'number_of_persons_injured', 'number_of_persons_killed', and 'zip_code' were converted to integers. The cleaned dataset is presented as follows:

collision_id INT	borough VARCHAR	zip_code INT	crash_date DATE	crash_time TIME	number_of_persons_inj INT	number_of_persons_kill INT	day_period VARCHAR(255)
3820157	MANHATTAI	10025	2018-01-01	04:00:00	1	0	Night
3818846	QUEENS	11373	2018-01-01	20:00:00	0	0	Evening
3818947	MANHATTAI	10025	2018-01-01	15:00:00	0	0	Afternoon
3820645	QUEENS	11354	2018-01-01	12:00:00	0	0	Night
3819261	BRONX	10459	2018-01-01	18:00:00	0	0	Evening
3820853	BROOKLYN	11207	2018-01-01	13:00:00	0	0	Afternoon
3819256	BROOKLYN	11212	2018-01-01	01:00:00	0	0	Night
3818787	QUEENS	11428	2018-01-01	12:00:00	1	0	Night
3825914	MANHATTAI	10003	2018-01-01	03:00:00	0	0	Night
3821641	QUEENS	11372	2018-01-01	02:00:00	0	0	Night
3820673	BROOKLYN	11229	2018-01-01	05:00:00	0	0	Morning

3.b Vehicles

The dataset **Vehicles** before the Data Cleaning process consisted of 465.823 rows and was structured as follows:

unique_ INT	collision_id INT	pre_crash VARCHAR(255)	vehicle_occupants INT	contributing_factor_1 VARCHAR(255)	driver_license_status VARCHAR(255)
17675653	3824399	Going Straight Ahead			
17675094	3820775	Parked	0	Unspecified	
17674184	3819574	Going Straight Ahead	1	Unspecified	Licensed
17674943	3818864	Parked	0	Unspecified	
17674640	3819262	Going Straight Ahead	2	Unspecified	Licensed
18490595	4060558	Going Straight Ahead	1	Driver Inattention/Distract	
17674359	3821451	Starting from Parking	1	Unsafe Lane Changing	
17674935	3820976	Going Straight Ahead	1	Following Too Closely	
17674741	3821415	Going Straight Ahead	1	Unspecified	Licensed
17676634	3820546	Going Straight Ahead	1	Unspecified	Unlicensed
17674438	3819736				
17675031	3822098	Going Straight Ahead	2	Other Vehicular	Licensed

Within the dataset, there was a primary key 'unique_id' used to identify each vehicle involved in an accident in New York in 2018. Additionally, there was a 'collision_id' column, which serves as the foreign key for the Collisions dataset and connected one or more vehicles to be linked to each accident.

For this dataset, all rows containing null values in the 'pre_crash' column were excluded as they were considered uninformative for our project's objectives. To remove these rows, the 'collision_id' values

were retained to subsequently eliminate all vehicles involved in accidents where one of the involved vehicles was removed. After implementing these adjustments, the dataset now consists of 414.559 rows, with 24.344 accidents removed:

unique_ INT	collision_id INT	pre_crash VARCHAR(255)	vehicle_occupants INT	contributing_factor_1 VARCHAR(255)	driver_license_status VARCHAR(255)
17675653	3824399	Going Straight Ahead			
17674184	3819574	Going Straight Ahead	1	Unspecified	Licensed
17674640	3819262	Going Straight Ahead	2	Unspecified	Licensed
18490595	4060558	Going Straight Ahead	1	Driver Inattention/Distract	
17674359	3821451	Starting from Parking	1	Unsafe Lane Changing	
17674935	3820976	Going Straight Ahead	1	Following Too Closely	
17674741	3821415	Going Straight Ahead	1	Unspecified	Licensed
17676634	3820546	Going Straight Ahead	1	Unspecified	Unlicensed
17675031	3822098	Going Straight Ahead	2	Other Vehicular	Licensed
17674187	3820562	Stopped in Traffic	2	Unspecified	Licensed

3.c Weather

The third dataset that underwent Data Cleaning operations is the **Weather** dataset, which comprises 2.119.920 rows. For this dataset, since the query was designed to cover all days and hours of 2018, it was unnecessary to discard any rows. However, it was deemed appropriate to modify some of the variables to increase the granularity of the dataset and obtain more useful information for our purposes. The dataset before any modifications was as follows:

date TIMESTAMP	temperature_2m FLOAT	relative_humidity_2m FLOAT	rain FLOAT	snowfall FLOAT	cloud_cover FLOAT	zip_code INT
2018-01-01 00:00:00	-11.676	48.09236	0.0	0.0	0.0	10001
2018-01-01 01:00:00	-11.876	48.238712	0.0	0.0	0.0	10001
2018-01-01 02:00:00	-12.226	48.972702	0.0	0.0	0.0	10001
2018-01-01 03:00:00	-12.526	49.521194	0.0	0.0	0.0	10001
2018-01-01 04:00:00	-12.826	51.399673	0.0	0.0	0.0	10001
2018-01-01 05:00:00	-13.126	53.815636	0.0	0.0	0.0	10001
2018-01-01 06:00:00	-13.276	55.185253	0.0	0.0	0.0	10001
2018-01-01 07:00:00	-13.476	56.33168	0.0	0.0	0.0	10001
2018-01-01 08:00:00	-13.676	57.008984	0.0	0.0	0.0	10001
2018-01-01 09:00:00	-13.826	57.461197	0.0	0.0	0.0	10001
2018-01-01 10:00:00	-13.976	57.418873	0.0	0.0	0.0	10001

The operations performed included: **rounding** the temperature and relative humidity, **splitting** the 'date' column into two separate columns for date and time for better temporal data management, and **defining a replacement function** to convert the numerical values of the 'rain', 'snowfall', and 'cloud_cover' columns into more descriptive categories. If the meteorological condition was not present, the value was converted to: 'No Rain', 'No Snow' and 'No Cloud'. However, if the condition was detected, the value was converted to one of the following: 'Rain', 'Snow' or 'Cloud'.

The resulting dataset from these operations is as follows:

date DATE	time TIME	zip_code INT	temperature_2m INT	relative_humidity_2m FLOAT	rain VARCHAR	snowfall VARCHAR	cloud_cover VARCHAR(25
2018-01-01	17:00:00	10001	-9	0.3	No Rain	No Snow	No Cloud
2018-01-01	18:00:00	10001	-8	0.3	No Rain	No Snow	No Cloud
2018-01-01	19:00:00	10001	-7	0.2	No Rain	No Snow	Cloud
2018-01-01	20:00:00	10001	-7	0.2	No Rain	No Snow	Cloud
2018-01-01	21:00:00	10001	-7	0.2	No Rain	No Snow	Cloud
2018-01-01	22:00:00	10001	-8	0.3	No Rain	No Snow	No Cloud
2018-01-01	23:00:00	10001	-10	0.3	No Rain	No Snow	Cloud
2018-01-02	00:00:00	10001	-10	0.4	No Rain	No Snow	No Cloud
2018-01-02	01:00:00	10001	-10	0.4	No Rain	No Snow	No Cloud
2018-01-02	02:00:00	10001	-10	0.5	No Rain	No Snow	No Cloud
2018-01-02	03:00:00	10001	-10	0.5	No Rain	No Snow	No Cloud

4. Database Construction and Data Storage

Having access to 4 data sources where there is an evident relationship between the Vehicles, Calendar, and Weather datasets with the Collision dataset, and considering the need to combine these data sources to answer our initial questions, we opted for a **relational database structure**.

We chose to implement an **SQLite database** (26) due to the strengths it offers, such as ease of use, lightweight nature, and the fact that it operates without a separate client/server setup. Being a file-based database, the data is stored in a single database file that can be easily transferred, shared, or copied across different platforms. Furthermore, SQLite supports the standard SQL language and provides many features of a full relational database, including integrity constraints, ACID transactions, and index management.

To interact with our database, we adopted a versatile approach that allowed for complete and flexible data management. We used the **macOS terminal** for creating and populating the database. For accessing the database, we utilized some **extensions available on Visual Studio Code**, specifically the **Database Client** extension and **dBizzy** extension.
We employed **Database Client (27)**, for designing the database structure, defining key constraints, and querying data. Instead, for designing and implementing the entity-relationship schema of the database, we employed the **dBizzy** extension (28).

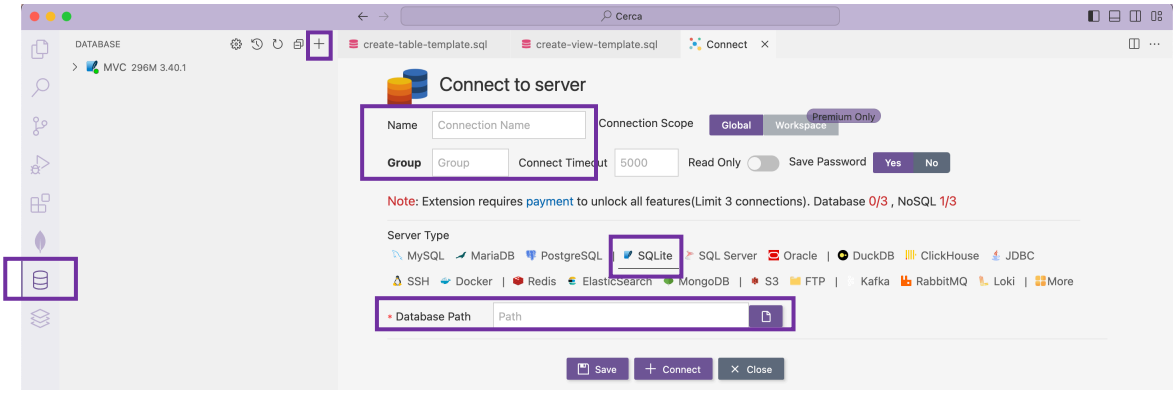
4.a Database Construction

Before creating the database, we **installed SQLite3** following the instructions available on the official SQLite website (29). Next, we opened the **terminal** and entered the command

```
sqlite3 MCV.db
```

to create the database, thus generating the associated file.

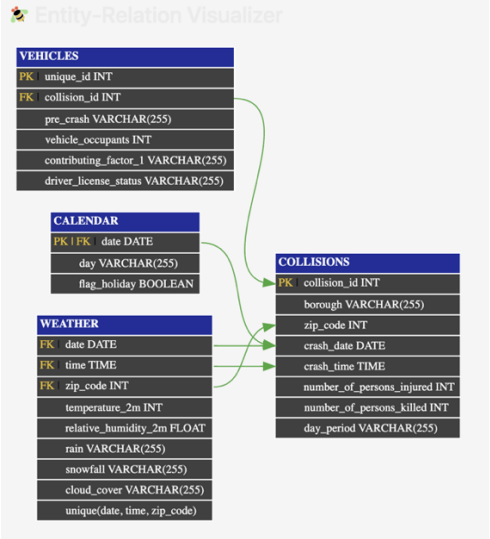
To access the database from Visual Studio Code, we clicked on the **Database Client extension**, created a **new connection** to the database, entered the **path of the .db file** generated during the creation of the database, and, finally, **name** the **connection**.



Once the new connection was created and appropriately named, we proceeded to **define the tables** directly from the Database Client extension on Visual Studio Code **using SQL code**. For each table, we defined the corresponding **fields**, specifying the **data format** and **all existing key constraints**, including the identification of primary and foreign keys of the tables.

Collisions	Vehicles	Weather	Calendar
CREATE TABLE COLLISIONS (collision_id INT PRIMARY KEY, borough VARCHAR(255), zip_code INT, crash_date DATE, crash_time TIME, number_of_persons_injured INT, number_of_persons_killed INT, day_period VARCHAR(255));	CREATE TABLE VEHICLES (unique_id INT PRIMARY KEY, collision_id INT, pre_crash VARCHAR(255), vehicle_occupants INT, contributing_factor_1 VARCHAR(255), driver_license_status VARCHAR(255), FOREIGN KEY (collision_id) REFERENCES COLLISIONS(collision_id));	CREATE TABLE WEATHER (date DATE, time TIME, zip_code INT, temperature_2m INT, relative_humidity_2m FLOAT, rain VARCHAR(255), snowfall VARCHAR(255), cloud_cover VARCHAR(255), unique(date, time, zip_code), FOREIGN KEY (date) REFERENCES COLLISIONS(crash_date), FOREIGN KEY (time) REFERENCES COLLISIONS(crash_time), FOREIGN KEY (zip_code) REFERENCES COLLISIONS(zip_code));	CREATE TABLE CALENDAR (date DATE PRIMARY KEY, day VARCHAR(255), flag_holiday BOOLEAN, FOREIGN KEY (date) REFERENCES COLLISIONS(crash_date));

With the **dBizzy** extension, we then created the **entity-relationship schema** starting from the .sql file where we previously defined the table structure, resulting in the following schema:

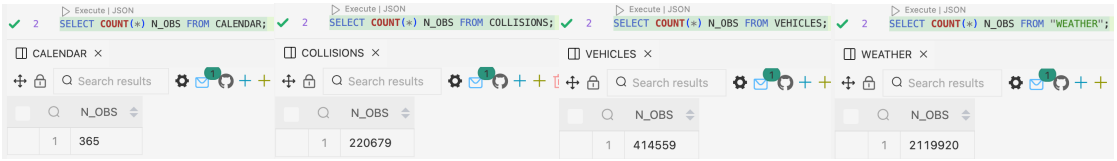


4.b Data Storage

To populate the tables, we proceeded again from the terminal by entering the following sequence of commands:

```
sqlite> .mode csv
sqlite> .import –skip 1 file_path/file.csv TABLE_NAME
```

As illustrated below, the tables have been correctly populated.



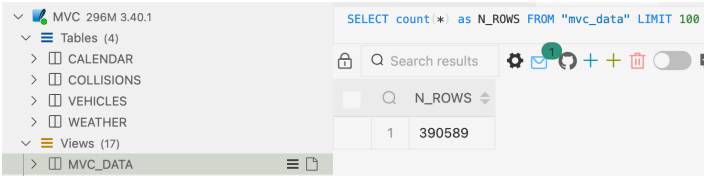
5. Data Integration

The data, once loaded into the database tables, were **combined using an SQL query** utilizing the **primary and foreign keys defined previously**. We opted to use an INNER JOIN instead of a LEFT JOIN starting from the main table (Collisions). This choice was motivated by the fact that, after performing data cleaning operations, some information within the Vehicles dataset had been removed, and some ZIP CODES in the Weather dataset were not present as they were not recognized as valid. Therefore, to avoid obtaining a dataset with missing fields, we preferred to consider only the common observations among the datasets.

```
CREATE VIEW MVC_DATA AS
SELECT A.*
, B.pre_crash, B.vehicle_occupants, B.contributing_factor_1 as
contributing_factor, B.driver_license_status
, C.day, C.flag_holiday
, D.cloud_cover, D.rain, D.relative_humidity_2m, D.snowfall, D.temperature_2m
FROM "COLLISIONS" AS A
INNER JOIN "VEHICLES" AS B
ON A.collision_id = B.collision_id
```

```
INNER JOIN "CALENDAR" as C
ON A.crash_date = C."date"
INNER JOIN "WEATHER" AS D
ON A.crash_date = D."date"
AND A.crash_time = D.time
AND A.zip_code = D.zip_code;
```

The obtained dataset thus contains a total of 390.589 observations.



6. Data Quality

We then proceeded to assess the quality of the data at our disposal by applying some data quality techniques to the integrated data.

6.1 Completeness

First, we decided to assess the completeness of each field within the table. As highlighted by the results obtained, the only fields presenting missing values are the following: **vehicle_occupants, contributing_factor,** and **driver_license_status**. Considering that the total number of observations in the integrated dataset MVC_DATA is 390.589, the percentage of missing values for these fields is less than 50%, so we decided not to remove the columns as they are relevant to our initial research questions. We exclude the option of directly removing instances with missing values as we consider it suboptimal to lose approximately 20% of the information from our dataset, as we could lose significant observations. However, during the data querying phase, we will still keep in mind that this information may not always be available in the dataset.

	VARIABLE	NULL_COUNT
1	unique_id	0
2	borough	0
3	zip_code	0
4	crash_date	0
5	crash_time	0
6	number_of_persons_injured	0
7	number_of_persons_killed	0
8	day_period	0
9	pre_crash	0
10	vehicle_occupants	12321
11	contributing_factor	576
12	driver_license_status	68701
13	day	0
14	flag_holiday	0
15	cloud_cover	0
16	rain	0
17	relative_humidity_2m	0
18	snowfall	0
19	temperature_2m	0

6.2 Syntactic Accuracy

We then considered it useful to assess the syntactic accuracy of the qualitative variables in the table that had a finite domain.

We then examined the values in the **borough** field, verifying that there were no data with values other than the five possible districts in the city of New York.
Additionally, we conducted the same check on the **day** field, confirming the absence of values outside the relevant domain of the variable.

	borough VARCHAR(255)
1	BRONX
2	BROOKLYN
3	MANHATTAN
4	QUEENS
5	STATEN ISLAND

	day VARCHAR(255)
1	Friday
2	Monday
3	Saturday
4	Sunday
5	Thursday
6	Tuesday
7	Wednesday

6.3 Consistency

Since the unique_id field remained as the primary key of the table, which identifies the vehicle involved in a specific incident (and not the vehicle itself, as it can be involved in multiple incidents during the year and therefore cannot function as the primary key), **we verified if**, after the join, there **were any duplicates in the data**. The verification revealed that the data was integrated correctly.

	unique_id_duplicate
1	0

7. Query Data

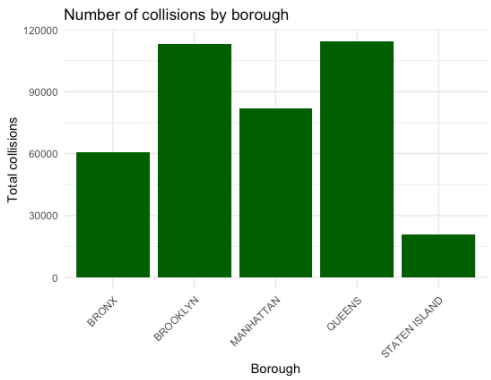
In this section, we displayed the responses to research questions that have been deemed most significant following the execution of queries.

7.a Observing the distribution of collisions across the five boroughs of New York City

```
CREATE VIEW VIEWA AS
SELECT borough, COUNT(*) AS total_collisions
FROM "MVC_DATA"
GROUP BY borough
ORDER BY TOTAL_COLLISIONS DESC;
```

The districts that appear to have a significantly higher number of collisions for the year 2018 are Queens and Brooklyn.

borough VARCHAR(255)	total_collisions
QUEENS	114329
BROOKLYN	113146
MANHATTAN	81716
BRONX	60731
STATEN ISLAND	20667

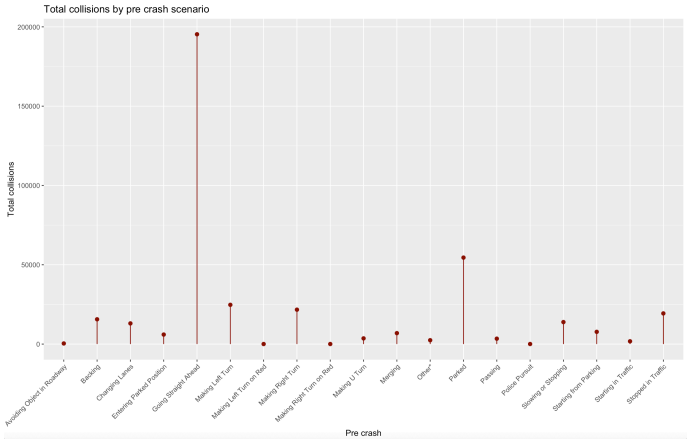


7.b Investigating the pre-collision dynamics to identify situations that may lead to collisions more frequently

```
CREATE VIEW VIEWG AS
SELECT pre_crash, COUNT(*) AS
total_collisions
FROM "MVC_DATA"
GROUP BY pre_crash
ORDER BY total_collisions DESC;
```

In the moments before the incident, most of the cars were traveling straight; a second factor that stands out more than others is the number of parked cars that were involved in the accidents.

pre_crash VARCHAR(255)	total_collisions
Going Straight Ahead	195324
Parked	54528
Making Left Turn	24759
Making Right Turn	21699
Stopped in Traffic	19343
Backing	15629
Slowing or Stopping	13874

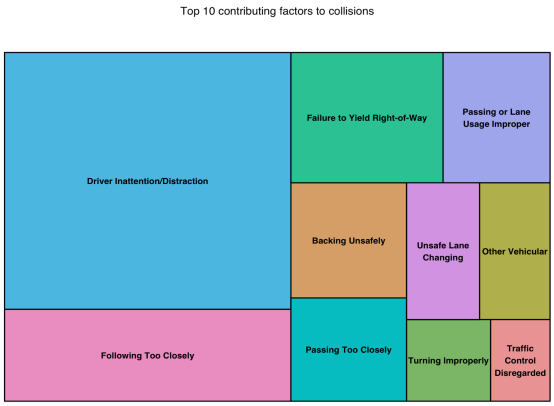


7.c Evaluating which driving behaviors may increase the probability of a collision

```
CREATE VIEW VIEWH AS
SELECT contributing_factor, COUNT(*) AS
total_collisions
FROM "MVC_DATA"
WHERE contributing_factor <> 'Unspecified'
GROUP BY contributing_factor
ORDER BY total_collisions DESC
LIMIT 10;
```

The most common cause of accidents is driver inattention, followed by 'following too closely' (which often leads to rear-end collisions).

contributing_factor VARCHAR(255)	total_collisions
Driver Inattention/Distraction	64299
Following Too Closely	23005
Failure to Yield Right-of-Way	17331
Passing or Lane Usage Improper	12187
Backing Unsafely	11650
Passing Too Closely	10439
Unsafe Lane Changing	8729
Other Vehicular	8433
Turning Improperly	5992
Traffic Control Disregarded	4246



7.d Exploring the meteorological conditions in which collisions primarily occur.

To create a representative index that can show whether there is a correlation between the number of incidents and weather conditions, we decided to calculate, for each weather condition, the ratio of the number of incidents in that condition to the total number of daily hours with that same condition, as well as the number of incidents without that condition to the total number of daily hours without that condition. This will allow us to assess if the ratio is higher during certain weather conditions. The index was calculated for each zip_code and for each day of 2018. Before proceeding with the averaging of the indices to summarize the information, we eliminated all those days where the number of hours with a certain condition, or the absence of condition, was equal to 0, in order to not alter the meaning of the data.

```
CREATE VIEW weather_view AS
    SELECT wcr.zip_code,
        wcr.date,
        COUNT(DISTINCT CASE WHEN wcr.Condition != 'No Condition' THEN wcr.time END) AS Condition_hours,
        COUNT(DISTINCT CASE WHEN wcr.Condition = 'No Condition' THEN wcr.time END) AS no_Condition_hours,
    FROM weather wcr
    GROUP BY wcr.zip_code, wcr.date;
CREATE VIEW collisions_view AS
    SELECT wcr.zip_code,
        wcr.date,
        SUM(CASE WHEN wcr.Condition != 'No Condition' THEN 1 ELSE 0 END) AS collision_with_Condition,
        SUM(CASE WHEN wcr.Condition = 'No Condition' THEN 1 ELSE 0 END) AS collision_no_Condition
    FROM weather wcr
```

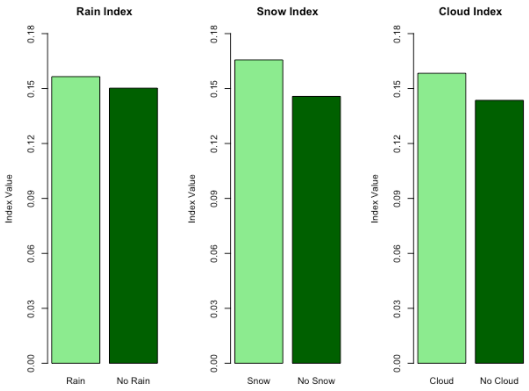
```
CREATE VIEW Condition_view as
SELECT wcr.zip_code,
    wcr.date,
    CASE WHEN wcr.Condition_hours = 0 THEN 9999 ELSE ROUND(cc.collision_with_Condition / CAST(wcr.Condition_hours AS REAL), 4) END AS Condition,
    CASE WHEN wcr.no_Condition_hours = 0 THEN 9999 ELSE ROUND(cc.collision_no_Condition / CAST(wcr.no_Condition_hours AS REAL), 4) END AS not_Condition
FROM weather_view wcr JOIN collisions_view cc ON wcr.zip_code = cc.zip_code and wcr.date = cc.date
group by wcr.zip_code, wcr.date
having Condition != 9999 and not_Condition != 9999;
create VIEW Condition_mean AS
    SELECT AVG(Condition) AS Condition_index,
        AVG(not_Condition) as no_Condition_index
        FROM Condition_view;

CREATE VIEW weather_indices AS
```

```
    INNER JOIN collisions cc ON wcr.zip_code =
cc.zip_code AND wcr.date = cc.crash_date AND
wcr.time = cc.crash_time
    GROUP BY wcr.zip_code,wcr.date;
```

```
SELECT 'Condition_index' AS weather_index,
Condition_index AS index_value FROM Condition_mean
UNION ALL
SELECT 'no_Condition_index', no_Condition_index FROM
Condition_mean
```

Unlike what we expected, weather conditions do not seem to be relevant regarding the likelihood of having accidents. There is a slight difference regarding snow indices, but we cannot draw conclusions; it might be the case.

weather_index	index_value
rain_index	0.156509587153215
no_rain_index	0.150231628617652
snow_index	0.165631042426569
no_snow_index	0.145726761374544
cloud_index	0.158354144668078
no_cloud_index	0.143577002653231

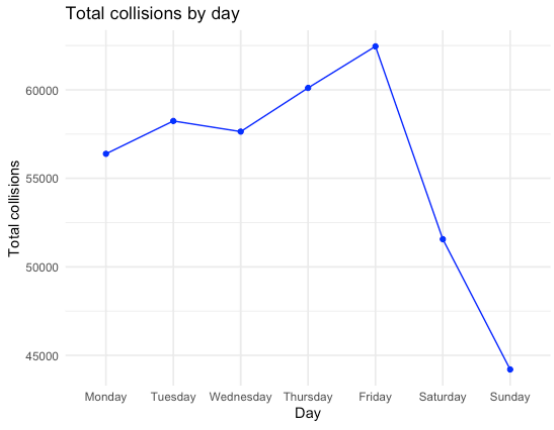


7.e Verifying if there are certain days of the week or holidays when collisions occur more frequently, and if there is a variation compared to the daily average

```
CREATE VIEW VIEWL AS
SELECT Day,
       COUNT(*) AS total_collisions
FROM MVC_DATA
GROUP BY Day
ORDER BY total_collisions;
```

Most accidents occur between Monday and Friday, decreasing on the weekends.

day VARCHAR	total_collisions
Sunday	44201
Saturday	51560
Monday	56384
Wednesday	57640
Tuesday	58240
Thursday	60108
Friday	62456

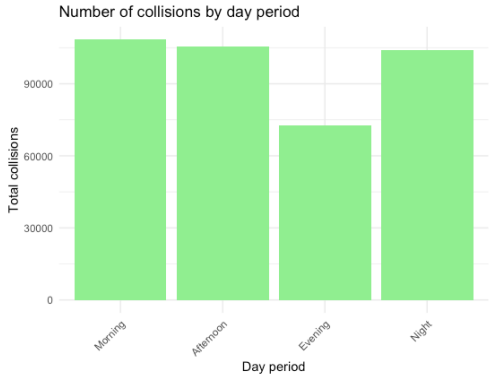


7.f Analysing if there are specific moments of the day when there is a higher concentration of collisions

```
CREATE VIEW VIEWN AS
SELECT day_period,
       COUNT(*) AS total_collisions
FROM MVC_DATA
GROUP BY day_period
ORDER BY total_collisions;
```

There is no evidence that more accidents occur during certain time periods, except for the evening, which has about 30,000 fewer accidents in 2018 compared to morning, afternoon, and night.

day_period VARCHAR(2	total_collisions
Evening	72611
Night	104162
Afternoon	105488
Morning	108328



8. Future Prospects

During our analyses, several aspects and characteristics related to incidents have emerged. However, they appear to have little significant impact. Consequently, we believe that a potential future development for this project could involve extending the examined time frame or expanding the analysis to a broader geographical area. This would allow us to comprehend not only urban areas but also non-metropolitan areas, where traffic influence may be lower, thus leading to the analysis of diversified scenarios.

9. References

1. NYC Open Data: https://opendata.cityofnewyork.us
2. Collisions Dataset: https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95
3. NYC Open Data API – Collisions: https://dev.socrata.com/foundry/data.cityofnewyork.us/h9gi-nx95
4. Socrata: https://dev.socrata.com
5. Json: https://docs.python.org/3/library/json.html
6. ReadTimeout: https://www.geeksforgeeks.org/exception-handling-of-python-requests-module/
7. Create an Account on NYC Open Data: https://data.cityofnewyork.us/profile/edit/developer_settings
8. SoQL: https://dev.socrata.com/docs/queries/
9. Vehicles Dataset: https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Vehicles/bm4k-52h4
10. NYC Open Data API – Vehicles: https://dev.socrata.com/foundry/data.cityofnewyork.us/bm4k-52h4
11. 2018 Calendar: https://www.timeanddate.com/calendar/custom.html?year=2018&month=1&country=1&typ=1&cols=1&display=1&df=1&hol=1
12. Requests: https://requests.readthedocs.io/en/latest/
13. BeautifulSoup: https://pypi.org/project/beautifulsoup4/
14. Open-Meteo: https://open-meteo.com
15. Historical Data Documentation: https://open-meteo.com/en/docs/historical-weather-api
16. Terms and Limits – Open-Meteo: https://open-meteo.com/en/terms
17. NYC Zip Codes: https://data.cityofnewyork.us/City-Government/Broadband-Adoption-and-Infrastructure-by-Zip-Code/qz5f-yx82
18. ZipDataMaps: https://www.zipdatamaps.com/10001
19. openmeteo-requests: https://pypi.org/project/openmeteo-requests/
20. requests_cache: https://pypi.org/project/requests-cache/
21. retry-requests: https://pypi.org/project/retry-requests/
22. Open-Meteo limits: https://github.com/open-meteo/open-meteo/issues/438
23. Geopy Documentation: https://geopy.readthedocs.io/en/stable/
24. Google Maps: https://www.google.com/maps/place/New+York,+Stati+Uniti/@40.6970193,-74.3093364,10z/data=!3m1!4b1!4m6!3m5!1s0x89c24fa5d33f083b:0xc80b8f06e177fe62!8m2!3d40.7127753!4d-74.0059728!16zL20vMDJfMjg2?hl=it&entry=ttu
25. Day Period: https://www.britannica.com/dictionary/eb/qa/parts-of-the-day-early-morning-late-morning-etc
26. SQLite: https://www.sqlite.org/about.html
27. Database Client: https://marketplace.visualstudio.com/items?itemName=cweijan.vscode-database-client2
28. dBizzy: https://marketplace.visualstudio.com/items?itemName=dBizzy.dbizzy
29: SQLite Download: https://www.sqlite.org/download.html