NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
Data Science and Machine Learning Master's Programme

Large Scale Data Management

ACADEMIC YEAR 2023-24, SPRING SEMESTER

Professors: Δημήτριος Τσουμάκος, Ιωάννης Κωνσταντίνου
Lab Director: Νικόλαος Χαλβαντζής

Final Project

Students:

Θοδωρής Παπαρρηγόπουλος, tpaparrigopoulos, 03003199

Στούμπου Χαραλαμπία, harastoumpou, 03400233

Data Description

Los Angeles Crime Data

The main dataset used in this project is the Los Angeles Crime Data, obtained from the public data repository of the City of Los Angeles. This dataset contains detailed records of crime incidents reported in the city, divided into two separate CSV files based on the time period they cover:

File 1: Contains crime data from the year 2010 up to 2019

File 2: Contains crime data from the year 2020 to present.

In addition to the above data, a series of smaller datasets that include information on police station locations, median household income, and ZIP code mappings will be used.

LA Police Stations

This dataset contains the locations of the 21 police stations in the city of Los Angeles. It is sourced from the public data repository of the City of Los Angeles and contains records described by the variables:

- OBJECTID: a unique identifier for each record
- DIVISION: the name of the division
- LOCATION: the address of the police station
- PREC: the precinct code
- x: the longitude coordinate of the police station
- y: the latitude coordinate of the police station

This dataset will help analyze the proximity of crime incidents to police stations and assess if there's any spatial correlation between crime rates and the presence of police stations.

Median Household Income by Zip Code (Los Angeles County):
This dataset contains information on the median household income for various areas in Los Angeles County, broken down by ZIP code. The data is based on census results from the years 2015, 2017, 2019, and 2021. For this project, only 2015 data will be used. Each record in this dataset is described by the variables:

- ZIP Code
- Community
- Estimated Median Income

This dataset will allow for an analysis of the socio-economic factors that may influence crime rates in different parts of Los Angeles.

Reverse Geocoding:

Geocoding refers to translating an address into a location in a coordinate system, while reverse geocoding is the process of mapping a pair of coordinates (latitude, longitude) to an address or a ZIP code. For this project, reverse geocoding is used, and this dataset provides the necessary mapping of coordinates to ZIP codes for the city of Los Angeles. Each record is described by the following variables:

- X: the longitude coordinate
- Y: the latitude coordinate
- FID: a unique identifier for each record
- DIVISION: the police division or precinct name
- LOCATION: the address of the location
- PREC: the precinct code

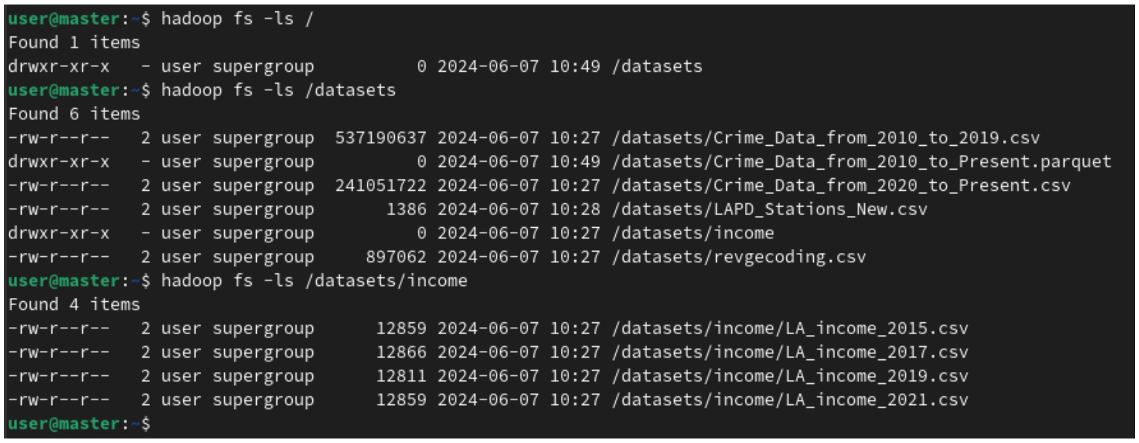
Task 1

To set up Apache Spark and the Hadoop Distributed File System (HDFS) in a distributed environment, we began by creating two VirtualBox virtual machines (VMs) using a pre-configured Ubuntu Server 22.04 image. One VM was designated as the master node and the other as the worker node. Each VM was allocated 4GB of RAM and 2 CPUs to ensure adequate resources. We configured the VMs with bridged network adapters to facilitate network communication. Unique hostnames were assigned, and IP addresses were obtained for both VMs to ensure proper connectivity. Next, we installed Hadoop on both VMs, configuring the core-site.xml, hdfs-site.xml, mapred-site.xml, and yarn-site.xml files. After formatting the namenode, we started the HDFS services. Following the Hadoop setup, we installed Apache Spark on both VMs. We configured the spark-env.sh and slaves files to define the master and worker nodes, ensuring that the Spark master could recognize the worker nodes. We then started the Hadoop HDFS and YARN services, followed by the Spark master and worker services. Finally, we verified the successful setup by connecting to the VMs via SSH, checking the service statuses, and ensuring that the web interfaces for HDFS and the Spark Job History Server were accessible from a web browser. This process resulted in a fully distributed Apache Spark and HDFS environment with accessible web applications for monitoring and management.

Task 2

To create a directory in HDFS for storing datasets in .csv format, we executed the command hadoop fs -mkdir -p /datasets. This command created the necessary directory structure in HDFS. We then used SCP to copy the dataset files from our local machine to the master VM. Following that, we transferred the files to the HDFS directory using the command hadoop fs -put filename.csv /datasets/ After setting up the HDFS directory, we wrote a Spark application to transform the main dataset from CSV format to Parquet format and store the resulting Parquet files back in HDFS. This involved initializing a Spark session, reading the CSV file from HDFS, and writing the data in Parquet format to a specified HDFS location. This transformation improved the efficiency and performance of data processing tasks. Finally, we verified the status of the file system to ensure that the data was correctly stored and accessible. This setup allowed us to leverage the benefits of both HDFS for distributed storage and Spark for efficient data processing and transformation.

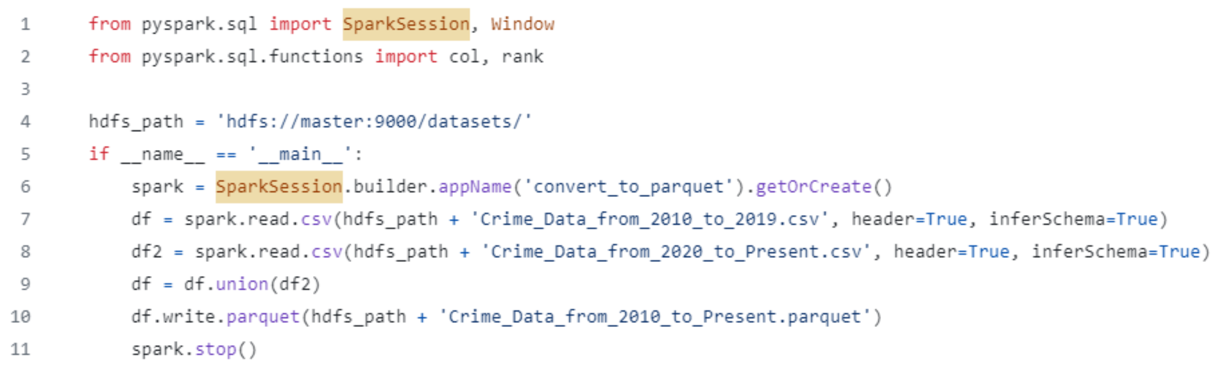
The next screenshot shows the state of the file system with the available data:



```
user@master:~$ hadoop fs -ls /
Found 1 items
drwxr-xr-x   - user supergroup          0 2024-06-07 10:49 /datasets
user@master:~$ hadoop fs -ls /datasets
Found 6 items
-rw-r--r--   2 user supergroup  537190637 2024-06-07 10:27 /datasets/Crime_Data_from_2010_to_2019.csv
drwxr-xr-x   - user supergroup          0 2024-06-07 10:49 /datasets/Crime_Data_from_2010_to_Present.parquet
-rw-r--r--   2 user supergroup  241051722 2024-06-07 10:27 /datasets/Crime_Data_from_2020_to_Present.csv
-rw-r--r--   2 user supergroup       1386 2024-06-07 10:28 /datasets/LAPD_Stations_New.csv
drwxr-xr-x   - user supergroup          0 2024-06-07 10:27 /datasets/income
-rw-r--r--   2 user supergroup     897062 2024-06-07 10:27 /datasets/revgecoding.csv
user@master:~$ hadoop fs -ls /datasets/income
Found 4 items
-rw-r--r--   2 user supergroup      12859 2024-06-07 10:27 /datasets/income/LA_income_2015.csv
-rw-r--r--   2 user supergroup      12866 2024-06-07 10:27 /datasets/income/LA_income_2017.csv
-rw-r--r--   2 user supergroup      12811 2024-06-07 10:27 /datasets/income/LA_income_2019.csv
-rw-r--r--   2 user supergroup      12859 2024-06-07 10:27 /datasets/income/LA_income_2021.csv
user@master:~$
```

Figure 1: Screenshot of the state of the file system with the available data

Finally, the Spark code below converts the main dataset to Parquet file format and stores the generated Parquet files in HDFS:



```python
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import col, rank

hdfs_path = 'hdfs://master:9000/datasets/'
if __name__ == '__main__':
    spark = SparkSession.builder.appName('convert_to_parquet').getOrCreate()
    df = spark.read.csv(hdfs_path + 'Crime_Data_from_2010_to_2019.csv', header=True, inferSchema=True)
    df2 = spark.read.csv(hdfs_path + 'Crime_Data_from_2020_to_Present.csv', header=True, inferSchema=True)
    df = df.union(df2)
    df.write.parquet(hdfs_path + 'Crime_Data_from_2010_to_Present.parquet')
    spark.stop()
```

Figure 2: Spark code that converts the main dataset to Parquet file format and stores the generated Parquet files in HDFS

Task 3

Query 1: For each year in the dataset, we need to determine the three months with the highest number of recorded crimes. The output should display the year, the month, the total number of crime incidents for that month, and the ranking of that month within the respective year. The results should be sorted in ascending order by year and in descending order by the number of crime records.

We will execute Query 1 using DataFrame and SQL APIs, applying them to both CSV and Parquet format of the datasets. The process for each API will be explained (same logic importing either csv or parquet format), and the results for all four combinations will be presented in a clear and organized manner. After presenting the results, we will discuss our findings, comparing the performance between the different APIs and file types. This analysis will help us understand the efficiency and effectiveness of each approach and file format.

The implementation starts with initializing a Spark session, reading and processing crime data from CSV and Parquet files.

DataFrame API (function/method named spark_sql)

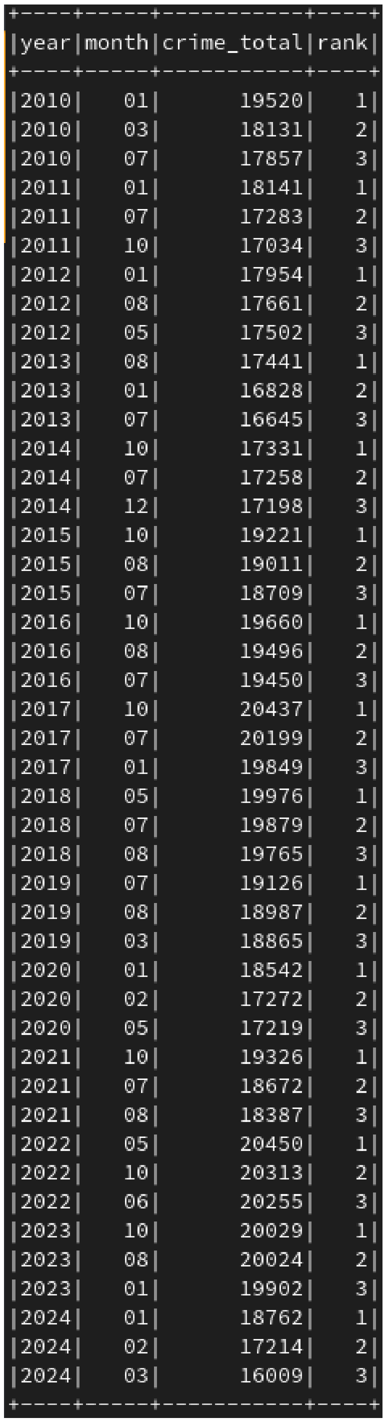
- The method first extracts the year and month from the DATE OCC column. The substr function is used to get the relevant parts of the date string.
- The df is grouped by the year and month columns, and the count of crimes for each group is calculated. The resulting count is renamed to crime_total.
- A window specification is defined to partition the data by year and order it by crime_total in descending order. The rank function is applied over this window to assign a rank to each month within each year based on the number of crimes.

- The DataFrame is ordered by year and crime_total in descending order. Only the top 3 months (where rank is less than or equal to 3) for each year are retained.

SQL API (function/method named sql_api)

- SQL is used to extract the year and month from the DATE OCC column.
- Another SQL query groups the data by year and month and counts the number of crimes for each group, creating a new view called crime_counts.
- Another SQL query ranks the months within each year based on the crime count using the RANK() window function.
- Finally, the top 3 months for each year are filtered out and the result is stored in top3_months_df.

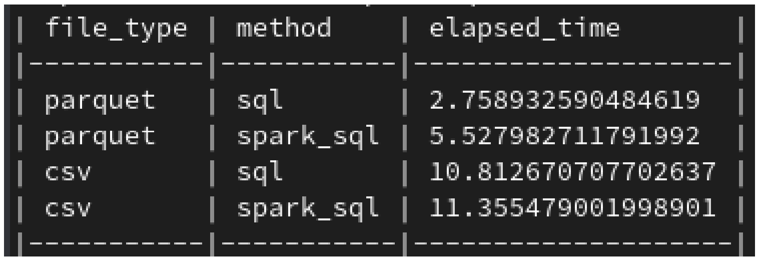
The output of the query execution is presented below. (Of course, the output of the query was the same regardless of the combination of APIs and file formats so it is presented once).



```
+----+-----+-----------+----+
|year|month|crime_total|rank|
+----+-----+-----------+----+
|2010|   01|      19520|   1|
|2010|   03|      18131|   2|
|2010|   07|      17857|   3|
|2011|   01|      18141|   1|
|2011|   07|      17283|   2|
|2011|   10|      17034|   3|
|2012|   01|      17954|   1|
|2012|   08|      17661|   2|
|2012|   05|      17502|   3|
|2013|   08|      17441|   1|
|2013|   01|      16828|   2|
|2013|   07|      16645|   3|
|2014|   10|      17331|   1|
|2014|   07|      17258|   2|
|2014|   12|      17198|   3|
|2015|   10|      19221|   1|
|2015|   08|      19011|   2|
|2015|   07|      18709|   3|
|2016|   10|      19660|   1|
|2016|   08|      19496|   2|
|2016|   07|      19450|   3|
|2017|   10|      20437|   1|
|2017|   07|      20199|   2|
|2017|   01|      19849|   3|
|2018|   05|      19976|   1|
|2018|   07|      19879|   2|
|2018|   08|      19765|   3|
|2019|   07|      19126|   1|
|2019|   08|      18987|   2|
|2019|   03|      18865|   3|
|2020|   01|      18542|   1|
|2020|   02|      17272|   2|
|2020|   05|      17219|   3|
|2021|   10|      19326|   1|
|2021|   07|      18672|   2|
|2021|   08|      18387|   3|
|2022|   05|      20450|   1|
|2022|   10|      20313|   2|
|2022|   06|      20255|   3|
|2023|   10|      20029|   1|
|2023|   08|      20024|   2|
|2023|   01|      19902|   3|
|2024|   01|      18762|   1|
|2024|   02|      17214|   2|
|2024|   03|      16009|   3|
+----+-----+-----------+----+
```

Figure 3: Output of Query 1

The execution time measurements for the four possible combinations of APIs and input file formats are presented in the table below.



```
| file_type | method    | elapsed_time       |
|-----------|-----------|--------------------|
| parquet   | sql       | 2.758932590484619  |
| parquet   | spark_sql | 5.527982711791992  |
| csv       | sql       | 10.812670707702637 |
| csv       | spark_sql | 11.355479001998901 |
|-----------|-----------|--------------------|
```

Figure 4: Execution time of Query 1 for the possible combinations of APIs and input file formats

Comments on performance

- **CSV vs. Parquet**:
  The execution time for the DataFrame API using csv files was 11.355 seconds, while using parquet files it was 5.528 seconds. Similarly, the execution time for the SQL API using csv files was 10.813 seconds, while using parquet files it was 2.759 seconds. This shows a significant performance improvement when using parquet files, regardless of the used API.

- **DataFrame API vs. SQL API:**
  Using csv files, the execution time for the DataFrame API was 11.355 seconds, slightly longer than the SQL API, which took 10.813 seconds. The difference is minor, indicating that both APIs perform similarly when using csv files. Using parquet format, the execution time for the DataFrame API was 5.528 seconds, whereas the SQL API was significantly faster at 2.759 seconds. This indicates a more noticeable difference in performance, favoring the SQL API when using parquet files.

In summary, the choice of file format and API has a significant impact on performance. These insights suggest that for large-scale data processing tasks, using **parquet files** in conjunction with the **SQL API** could provide the best performance.

Task 4

Query 2: To sort the parts of the day based on the number of crime incidents that occurred on the street ("STREET") in descending order. The parts of the day are defined as follows:

- Morning: 5:00 AM – 11:59 AM
- Midday: 12:00 PM – 4:59 PM
- Evening: 5:00 PM – 8:59 PM

- Night: 9:00 PM – 4:59 AM

We will use both DataFrame and RDD APIs while recording the execution time for each. First, we defined the "find_part_of_day" function, which determines the part of the day (NIGHT, MORNING, MIDDAY, EVENING) based on the time of the crime. Now, let's outline the steps for each API.

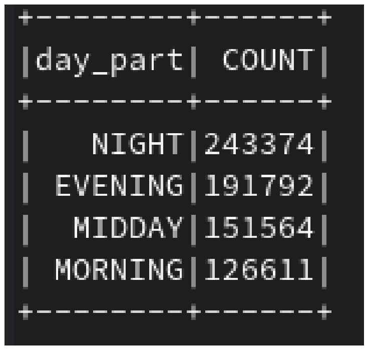
DataFrame API Implementation

- Filter the data to include only records where the premise description is "STREET".
- Define a udf to classify the time of occurrence into parts of the day, using the find_part_of_day function.
- Add a new column (day_part) to the DataFrame, group the data by day_part and count the number of incidents in each part of the day.
- Order the results by counts in descending order.

RDD API Implementation

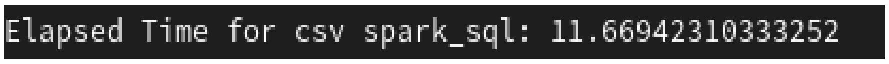
- Filter the RDD to include only records where the premise description is "STREET".
- Map each record to a tuple containing the part of the day and a count of 1.
- Reduce the mapped RDD by key (part of the day) to count the number of incidents in each part of the day
- Sort the results by count in descending order.
- Collect and print the result

Presented below are the results of the query execution, along with the corresponding execution times.

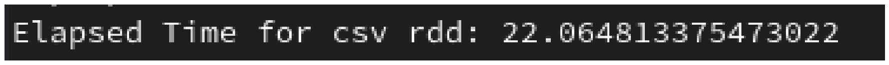


```
+--------+------+
|day_part| COUNT|
+--------+------+
|   NIGHT|243374|
| EVENING|191792|
|  MIDDAY|151564|
| MORNING|126611|
+--------+------+
```

Figure 5: Output of Query 2



```
Elapsed Time for csv spark_sql: 11.66942310333252
```



```
Elapsed Time for csv rdd: 22.064813375473022
```

Figure 6: Execution time of Query 2 using the DataFrame and the RDD APIs respectively

As we can see, parts of the day have been sorted in descending order. The highest number of crime incidents crime on the street occurs during the night, which aligns with the fact that the night hours, typically from 9 PM to 5 AM, are often associated with increased criminal activity. The lowest number of incidents (reported in the street) occurs during the morning hours.

The measurements of execution time reveal that **the DataFrame API is significantly more efficient than the RDD API** for processing CSV files and performing the required transformations and aggregations.

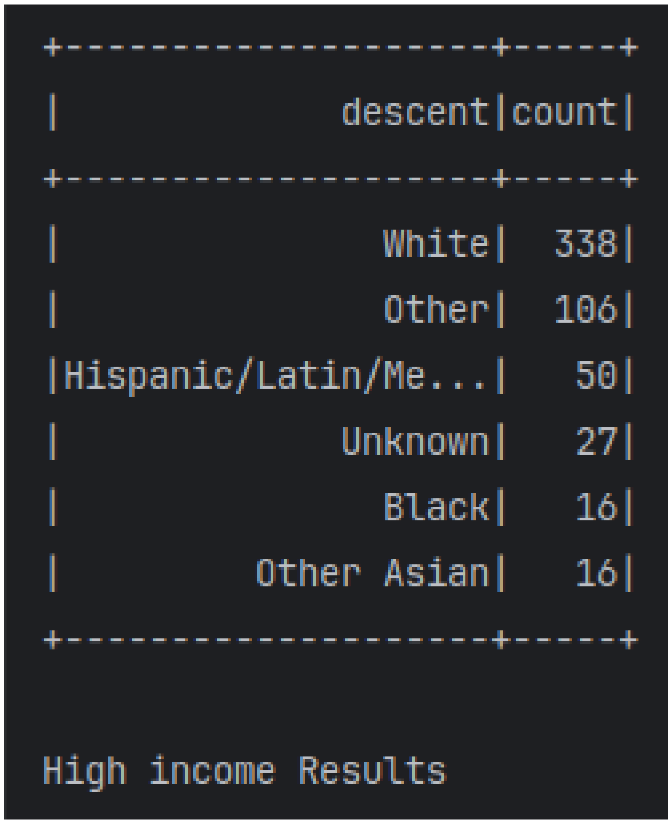
Task 5

Query 3: We need to determine the descent of recorded crime victims in Los Angeles for the year 2015 in the three areas (ZIP Codes) with the highest and the three areas with the lowest median household income. The results are presented below, in two separate tables, sorted from the highest to the lowest number of victims per descent.

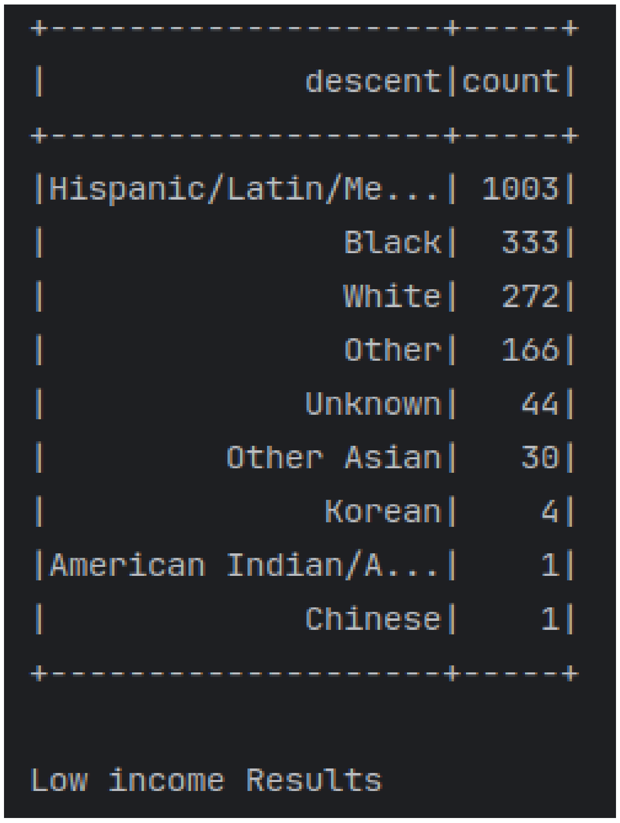
For the execution of this query, we will need three datasets: "Crime_Data_from_2010_to_2019.csv" (since we are focusing on the recorded crime victims for the year 2015, we don't need to combine the crime datasets for other years), "LA_income_2015.csv," and "revgecoding.csv."

- First, we read the datasets into dataframes, we process/ filter the data. Specifically, we filter out crime data points without victim or descent information and extract the first ZIP code from the revgecoding data when more ZIP codes are present.
- The "get_bottom_3" and "get_higher_3" function retrieves the top 3 and bottom 3 ZIP codes based on income.
- The "filter_by_double_zipcodes" function performs an inner join between income and revgecoding dataframes to match ZIP codes. The resulting dataframe, named df_geocoordinates, is then joined with the crime data by the "convert_to_descent" function. The code that we used allows for specifying different join strategies using hints, since in the main execution we want to use different join strategies (broadcast, merge, shuffle_hash, shuffle_replicate_nl).
- Next, we convert descent codes (using the mapping of descent codes to their descriptions as referenced in the information accompanying the dataset), count the number of victims by descent and finally print results for both high and low-income areas.

Below are the outputs of the query execution, along with the execution times when no join strategy is defined, and when the join strategy is set to one of the following: broadcast, merge, shuffle_hash, and shuffle_replicate_nl.

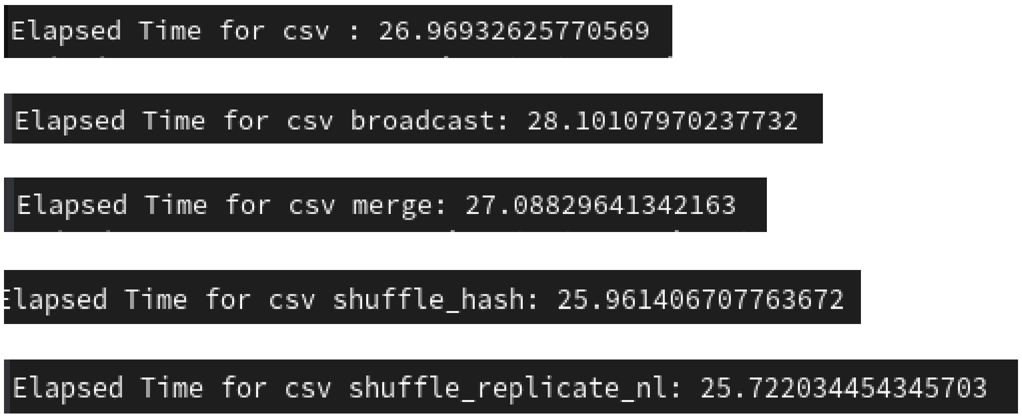


```
+--------------------+-----+
|             descent|count|
+--------------------+-----+
|               White|  338|
|               Other|  106|
|Hispanic/Latin/Me...|   50|
|             Unknown|   27|
|               Black|   16|
|         Other Asian|   16|
+--------------------+-----+


High income Results
```



```
+--------------------+-----+
|             descent|count|
+--------------------+-----+
|Hispanic/Latin/Me...| 1003|
|               Black|  333|
|               White|  272|
|               Other|  166|
|             Unknown|   44|
|         Other Asian|   30|
|              Korean|    4|
|American Indian/A...|    1|
|             Chinese|    1|
+--------------------+-----+


Low income Results
```

Figure 7: Output of Query 3



```
Elapsed Time for csv : 26.96932625770569
```

```
Elapsed Time for csv broadcast: 28.10107970237732
```

```
Elapsed Time for csv merge: 27.08829641342163
```

```
lapsed Time for csv shuffle_hash: 25.961406707763672
```

```
Elapsed Time for csv shuffle_replicate_nl: 25.722034454345703
```

Figure 8: Execution time of Query 3 using different join strategies

The Catalyst Optimizer chooses the **broadcast** join strategy. Generally, the Broadcast Join indicates that one of the datasets is small enough to be broadcast to all nodes. This avoids shuffling the larger dataset, making the join operation faster. So, considering that the "LA_income_2015" dataset is very small compared to the "Crime_Data_from_2010_to_2019" dataset, the choice of the broadcast join can be justified.

Task 6 & 7

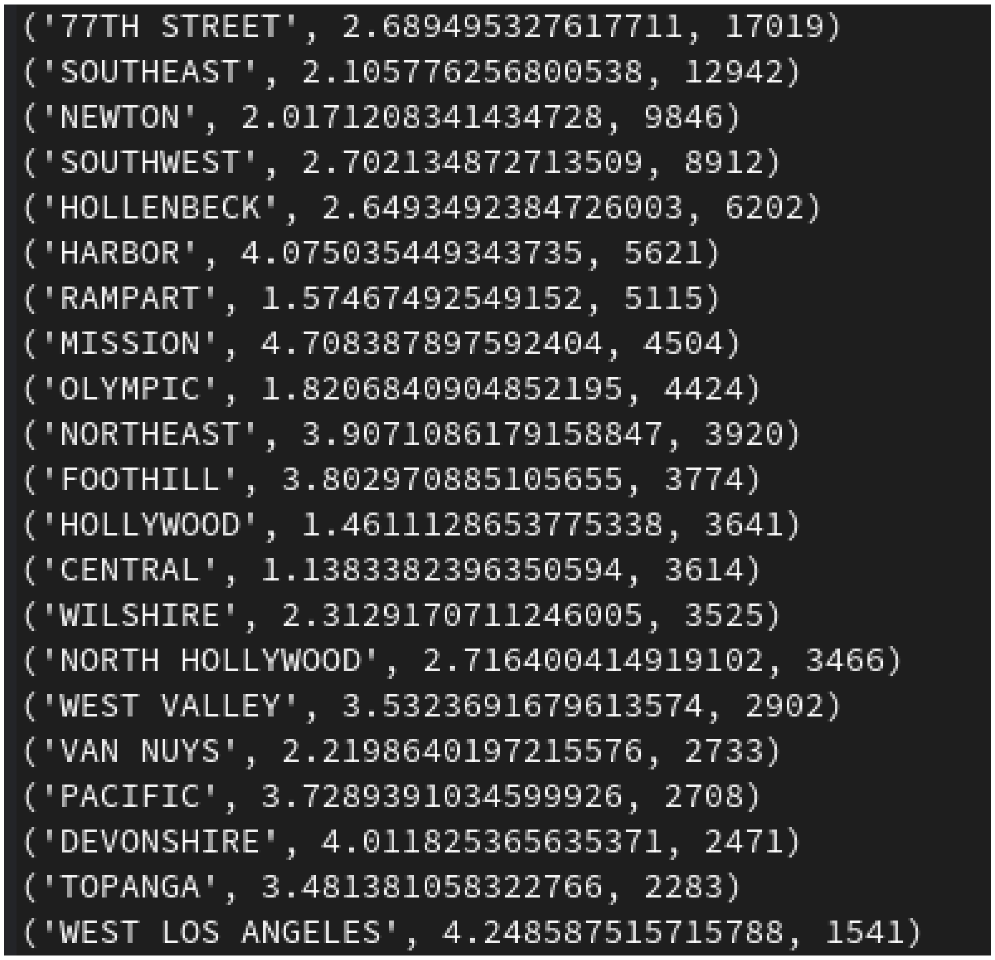
Query 4: For the final query, we need to calculate the number of crime incidents involving the use of firearms of any kind that each police division handled, as well as the average distance of each incident from the respective police station. The results will be sorted by the number of incidents in descending order.

We are going to execute this query using the RDD API (task 6), as well as the DataFrame API (task7). Also, we need to perform a join between the main dataset (Los Angeles Crime Data) and the LA Police Stations dataset. For this purpose we will implement broadcast join and repartition join (we will also implement the join without specifying its type).

Next, we will outline the most important part of the code and present the results:

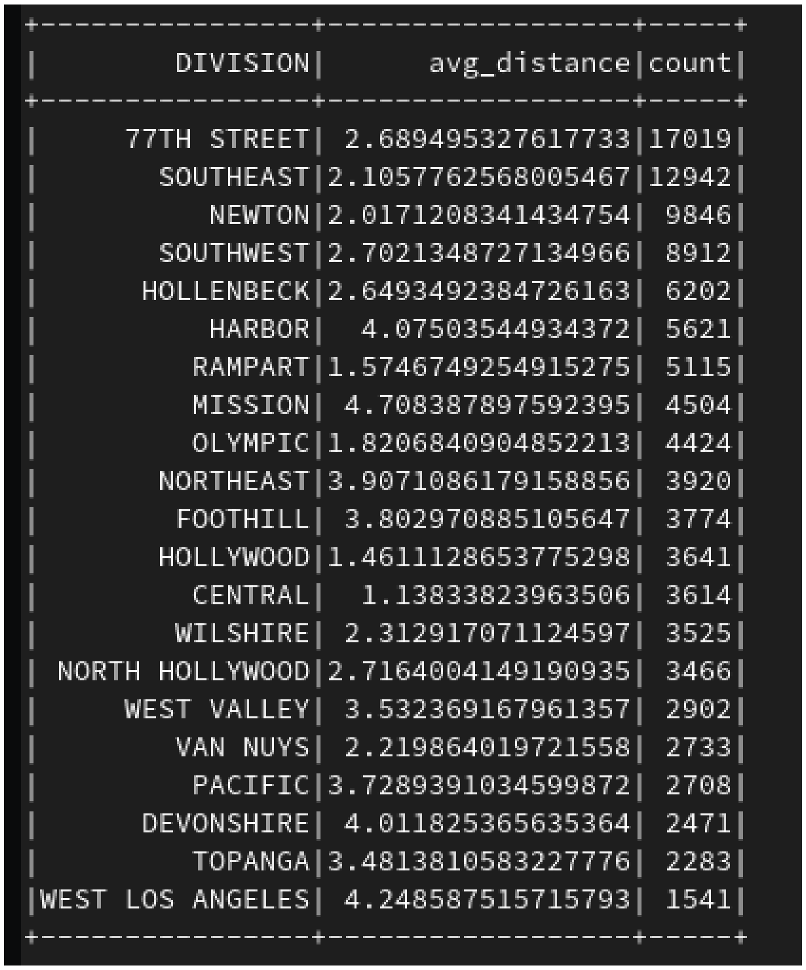
- We filter the incidents that involve the use of firearms using the function "select_1xx".
- We define the functions "**get_distance_rdd" and** "**get_distance"** that calculates distance for RDD and DataFrame respectively using the geopy library.
- We define the functions "**read_datasets_rdd" and** "**read_datasets_csv"** that read and process data using the RDD and DataFrame API respectively. (They also filter out Null Island entries.)
- Finally we execute query 4 with different join strategies using the RDD and the DataFrame API and print the output and the execution time of each combination. (Of course, the output of the query was the same regardless of the type of join so we present it once for each API).

Finally, the results of task 6 and 7 are presented below:



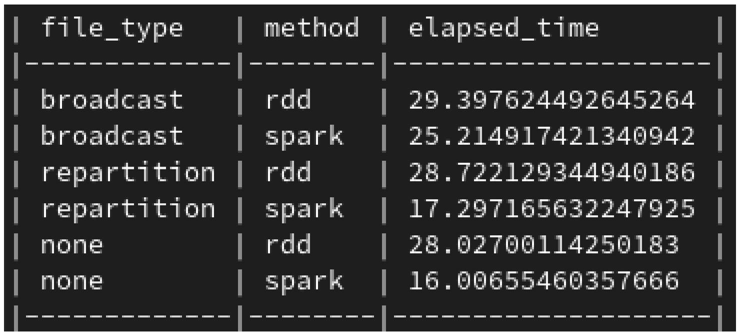
```
('77TH STREET', 2.6894953276177711, 17019)
('SOUTHEAST', 2.105776256800538, 12942)
('NEWTON', 2.0171208341434728, 9846)
('SOUTHWEST', 2.702134872713509, 8912)
('HOLLENBECK', 2.6493492384726003, 6202)
('HARBOR', 4.075035449343735, 5621)
('RAMPART', 1.57467492549152, 5115)
('MISSION', 4.708387897592404, 4504)
('OLYMPIC', 1.8206840904852195, 4424)
('NORTHEAST', 3.9071086179158847, 3920)
('FOOTHILL', 3.802970885105655, 3774)
('HOLLYWOOD', 1.4611128653775338, 3641)
('CENTRAL', 1.1383382396350594, 3614)
('WILSHIRE', 2.3129170711246005, 3525)
('NORTH HOLLYWOOD', 2.716400414919102, 3466)
('WEST VALLEY', 3.5323691679613574, 2902)
('VAN NUYS', 2.2198640197215576, 2733)
('PACIFIC', 3.7289391034599926, 2708)
('DEVONSHIRE', 4.011825365635371, 2471)
('TOPANGA', 3.481381058322766, 2283)
('WEST LOS ANGELES', 4.248587515715788, 1541)
```

Figure 9: Output of Query 4 using RDD API



```
+----------------+------------------+-----+
|        DIVISION|      avg_distance|count|
+----------------+------------------+-----+
|     77TH STREET| 2.689495327617733|17019|
|       SOUTHEAST|2.1057762568005467|12942|
|          NEWTON|2.0171208341434754| 9846|
|       SOUTHWEST|2.7021348727134966| 8912|
|      HOLLENBECK|2.6493492384726163| 6202|
|          HARBOR|  4.07503544934372| 5621|
|         RAMPART|1.5746749254915275| 5115|
|         MISSION| 4.708387897592395| 4504|
|         OLYMPIC|1.8206840904852213| 4424|
|       NORTHEAST|3.9071086179158856| 3920|
|        FOOTHILL| 3.802970885105647| 3774|
|       HOLLYWOOD|1.4611128653775298| 3641|
|         CENTRAL|  1.13833823963506| 3614|
|        WILSHIRE| 2.312917071124597| 3525|
| NORTH HOLLYWOOD|2.7164004149190935| 3466|
|     WEST VALLEY| 3.532369167961357| 2902|
|        VAN NUYS| 2.219864019721558| 2733|
|         PACIFIC|3.7289391034599872| 2708|
|      DEVONSHIRE| 4.011825365635364| 2471|
|         TOPANGA|3.4813810583227776| 2283|
|WEST LOS ANGELES| 4.248587515715793| 1541|
+----------------+------------------+-----+
```

Figure 8: Figure 9: Output of Query 4 using DataFrame API



```
| file_type   | method | elapsed_time       |
|-------------|--------|--------------------|
| broadcast   | rdd    | 29.397624492645264 |
| broadcast   | spark  | 25.214917421340942 |
| repartition | rdd    | 28.722129344940186 |
| repartition | spark  | 17.297165632247925 |
| none        | rdd    | 28.02700114250183  |
| none        | spark  | 16.00655460357666  |
|-------------|--------|--------------------|
```

Figure 7: Execution time of Query 4 for different combinations of APIs and join strategies

Link to github repository that contains all used implementatios:
https://github.com/thodpap/BigDataNTUA2024