Query 4

Find the racial profile of registered crime victims (Vict Descent) in Los Angeles for the year 2015 in the 3 areas with the highest per capita income. Do the same for the 3 areas with the lowest income. Use the mapping of the descent codes to the full description from the Race and Ethnicity codes dataset. The results should be printed in two separate tables from highest to lowest number of victims per racial group

Question 4

Implement Query 4 using the DataFrame or SQL API. Execute your implementation by scaling the total computational resources you will use: Specifically, you are asked to execute your implementation with 2 executors with the following configurations:

- 1 core/2 GB memory
- 2 cores/4GB memory
- 4 cores/8GB memory

Comment on the results.

```
### 1 Core, 2GB ###
from pyspark.sql import SparkSession
from sedona.register.geo_registrator import SedonaRegistrator
from sedona.sql.types import GeometryType
from pyspark.sql.functions import col, regexp_replace, explode,
to_timestamp, year, expr, sum as spark_sum, count
import time

# Initialize SparkSession with 1 core and 2GB memory
spark = SparkSession.builder \
    .appName("Query 4 - 1 Core, 2GB") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.cores", "1") \
    .config("spark.executor.memory", "2g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

start_time = time.time()

# Load census data from a GeoJSON file
census_blocks_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/2010_Census_Blocks.geojson"
census_raw_df = spark.read.format("geojson").option("multiline",
"true").load(census_blocks_path)

census_df =
census_raw_df.select(explode(col("features")).alias("feature")) \
    .select(
```

```
col("feature.geometry").cast(GeometryType()).alias("census_geometry"
),
        col("feature.properties.ZCTA10").alias("zip"),
        col("feature.properties.POP_2010").alias("population")
    ).filter(col("population").isNotNull())

# Load crime data from a CSV file
crime_data_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_df = spark.read.csv(crime_data_path, header=True,
inferSchema=True).dropDuplicates()

crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a")) \
    .filter(year(col("DATE_OCC_TIMESTAMP")) == 2015) \
    .withColumn("crime_geometry", expr("ST_Point(LON, LAT)"))

joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"),
"inner").dropDuplicates()

# Aggregate data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))

crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner") \
    .filter((col("population (2015)").isNotNull()) &
(col("population (2015)") > 0)) \
    .withColumn("crimes_per_person (2015)", col("crimes (2015)") /
col("population (2015)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
income_df = spark.read.csv(income_path, header=True,
inferSchema=True) \
    .withColumn("zip", col("Zip Code").cast("string")) \
    .withColumn("income_cleaned",
regexp_replace(regexp_replace(col("Estimated Median Income"), "\\$",
""), ",", "")) \
    .withColumn("estimated_median_income (2015)",
col("income_cleaned").cast("double"))

combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Get top and bottom 3 areas by income
```

```
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

top_zips = [row["zip"] for row in top_areas.collect()]
bottom_zips = [row["zip"] for row in bottom_areas.collect()]

# Filter and aggregate racial data
race_codes_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/RE_codes.csv"
race_codes_df = spark.read.csv(race_codes_path, header=True,
inferSchema=True)

final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

if top_zips:
    top_area_results =
final_joined_df.filter(col("zip").isin(top_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Top 3 High-Income Areas Racial Profile:")
    top_area_results.show(3, truncate=False)
else:
    print("No data available for top areas.")

if bottom_zips:
    bottom_area_results =
final_joined_df.filter(col("zip").isin(bottom_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Bottom 3 Low-Income Areas Racial Profile:")
    bottom_area_results.show(3, truncate=False)
else:
    print("No data available for bottom areas.")

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution Time with 1 Core, 2GB Memory:
{round(execution_time, 2)} seconds")

spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3731	application_1732639283265_3677	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.

Top 3 High-Income Areas Racial Profile:
+--------------------+---+
|Victim Descent      |#  |
+--------------------+---+
|White               |832|
|Other               |183|
|Hispanic/Latin/Mexican|86 |
+--------------------+---+
only showing top 3 rows

Bottom 3 Low-Income Areas Racial Profile:
+--------------------+----+
|Victim Descent      |#   |
+--------------------+----+
|Hispanic/Latin/Mexican|1687|
|Black               |1567|
|White               |1028|
+--------------------+----+
only showing top 3 rows

Execution Time with 1 Core, 2GB Memory: 121.2 seconds
```

```
### 2 Cores, 4GB ###
from pyspark.sql import SparkSession
from sedona.register.geo_registrator import SedonaRegistrator
from sedona.sql.types import GeometryType
from pyspark.sql.functions import col, regexp_replace, explode,
to_timestamp, year, expr, sum as spark_sum, count
import time

# Initialize SparkSession with 2 cores and 4GB memory
spark = SparkSession.builder \
    .appName("Query 4 - 2 Cores, 4GB") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.cores", "2") \
    .config("spark.executor.memory", "4g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

start_time = time.time()

# Load census data from a GeoJSON file
census_blocks_path = "s3://initial-notebook-data-bucket-
```

```
dblab-905418150721/2010_Census_Blocks.geojson"
census_raw_df = spark.read.format("geojson").option("multiline",
"true").load(census_blocks_path)

census_df =
census_raw_df.select(explode(col("features")).alias("feature")) \
    .select(

col("feature.geometry").cast(GeometryType()).alias("census_geometry"
),
        col("feature.properties.ZCTA10").alias("zip"),
        col("feature.properties.POP_2010").alias("population")
    ).filter(col("population").isNotNull())

# Load crime data from a CSV file
crime_data_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_df = spark.read.csv(crime_data_path, header=True,
inferSchema=True).dropDuplicates()

crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a")) \
    .filter(year(col("DATE_OCC_TIMESTAMP")) == 2015) \
    .withColumn("crime_geometry", expr("ST_Point(LON, LAT)"))

joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"),
"inner").dropDuplicates()

# Aggregate data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))

crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner") \
    .filter((col("population (2015)").isNotNull()) &
(col("population (2015)") > 0)) \
    .withColumn("crimes_per_person (2015)", col("crimes (2015)") /
col("population (2015)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
income_df = spark.read.csv(income_path, header=True,
inferSchema=True) \
    .withColumn("zip", col("Zip Code").cast("string")) \
    .withColumn("income_cleaned",
regexp_replace(regexp_replace(col("Estimated Median Income"), "\\$",
""), ",", "")) \
```

```
    .withColumn("estimated_median_income (2015)",
col("income_cleaned").cast("double"))

combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Get top and bottom 3 areas by income
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

top_zips = [row["zip"] for row in top_areas.collect()]
bottom_zips = [row["zip"] for row in bottom_areas.collect()]

# Filter and aggregate racial data
race_codes_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/RE_codes.csv"
race_codes_df = spark.read.csv(race_codes_path, header=True,
inferSchema=True)

final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

if top_zips:
    top_area_results =
final_joined_df.filter(col("zip").isin(top_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Top 3 High-Income Areas Racial Profile:")
    top_area_results.show(3, truncate=False)
else:
    print("No data available for top areas.")

if bottom_zips:
    bottom_area_results =
final_joined_df.filter(col("zip").isin(bottom_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Bottom 3 Low-Income Areas Racial Profile:")
    bottom_area_results.show(3, truncate=False)
else:
    print("No data available for bottom areas.")

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution Time with 2 Cores, 4GB Memory:
{round(execution_time, 2)} seconds")

spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3747	application_1732639283265_3693	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.

Top 3 High-Income Areas Racial Profile:
+--------------------+---+
|Victim Descent      |#  |
+--------------------+---+
|White               |832|
|Other               |183|
|Hispanic/Latin/Mexican|86 |
+--------------------+---+
only showing top 3 rows

Bottom 3 Low-Income Areas Racial Profile:
+--------------------+----+
|Victim Descent      |#   |
+--------------------+----+
|Hispanic/Latin/Mexican|1687|
|Black               |1567|
|White               |1028|
+--------------------+----+
only showing top 3 rows

Execution Time with 2 Cores, 4GB Memory: 155.25 seconds
```

```python
### 4 Cores, 8GB ###
from pyspark.sql import SparkSession
from sedona.register.geo_registrator import SedonaRegistrator
from sedona.sql.types import GeometryType
from pyspark.sql.functions import col, regexp_replace, explode,
to_timestamp, year, expr, sum as spark_sum, count
import time

# Initialize SparkSession with 4 cores and 8GB memory
spark = SparkSession.builder \
    .appName("Query 4 - 4 Cores, 8GB") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.cores", "4") \
    .config("spark.executor.memory", "8g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

start_time = time.time()
```

```
# Load census data from a GeoJSON file
census_blocks_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/2010_Census_Blocks.geojson"
census_raw_df = spark.read.format("geojson").option("multiline",
"true").load(census_blocks_path)

census_df =
census_raw_df.select(explode(col("features")).alias("feature")) \
    .select(

col("feature.geometry").cast(GeometryType()).alias("census_geometry"
),
        col("feature.properties.ZCTA10").alias("zip"),
        col("feature.properties.POP_2010").alias("population")
    ).filter(col("population").isNotNull())

# Load crime data from a CSV file
crime_data_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_df = spark.read.csv(crime_data_path, header=True,
inferSchema=True).dropDuplicates()

crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a")) \
    .filter(year(col("DATE_OCC_TIMESTAMP")) == 2015) \
    .withColumn("crime_geometry", expr("ST_Point(LON, LAT)"))

joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"),
"inner").dropDuplicates()

# Aggregate data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))

crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner") \
    .filter((col("population (2015)").isNotNull()) &
(col("population (2015)") > 0)) \
    .withColumn("crimes_per_person (2015)", col("crimes (2015)") /
col("population (2015)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
income_df = spark.read.csv(income_path, header=True,
inferSchema=True) \
    .withColumn("zip", col("Zip Code").cast("string")) \
    .withColumn("income_cleaned",
```

```
regexp_replace(regexp_replace(col("Estimated Median Income"), "\\$",
""), ",", "")) \
    .withColumn("estimated_median_income (2015)",
col("income_cleaned").cast("double"))

combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Get top and bottom 3 areas by income
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

top_zips = [row["zip"] for row in top_areas.collect()]
bottom_zips = [row["zip"] for row in bottom_areas.collect()]

# Filter and aggregate racial data
race_codes_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/RE_codes.csv"
race_codes_df = spark.read.csv(race_codes_path, header=True,
inferSchema=True)

final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

if top_zips:
    top_area_results =
final_joined_df.filter(col("zip").isin(top_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Top 3 High-Income Areas Racial Profile:")
    top_area_results.show(3, truncate=False)
else:
    print("No data available for top areas.")

if bottom_zips:
    bottom_area_results =
final_joined_df.filter(col("zip").isin(bottom_zips)) \
        .groupBy("Vict Descent
Full").agg(count("LOCATION").alias("#")) \
        .orderBy(col("#").desc()) \
        .withColumnRenamed("Vict Descent Full", "Victim Descent")
    print("Bottom 3 Low-Income Areas Racial Profile:")
    bottom_area_results.show(3, truncate=False)
else:
    print("No data available for bottom areas.")

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution Time with 4 Cores, 8GB Memory:
{round(execution_time, 2)} seconds")
```

```
spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3758	application_1732639283265_3704	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.

Top 3 High-Income Areas Racial Profile:
+---------------------+---+
|Victim Descent       |#  |
+---------------------+---+
|White                |832|
|Other                |183|
|Hispanic/Latin/Mexican|86 |
+---------------------+---+
only showing top 3 rows

Bottom 3 Low-Income Areas Racial Profile:
+---------------------+----+
|Victim Descent       |#   |
+---------------------+----+
|Hispanic/Latin/Mexican|1687|
|Black                |1567|
|White                |1028|
+---------------------+----+
only showing top 3 rows

Execution Time with 4 Cores, 8GB Memory: 152.32 seconds
```

Answer

Efficiency was proven by the fact that the configuration of a single-core machine with 2GB of memory had the fastest execution time of 121.2 seconds, after which the increased execution time, when measured with additional resources of 2 cores and 4GB, ran to 155.25 seconds. Apparently, the overhead created by any additional processes involves dealing with coordination of tasks or task utilization. Using even 4 cores with 8GB further reduces execution time to only 152.32 seconds but never approaches the efficiency of the first configuration.

This indicates that the workload does not scale as well with more resources. Some, or all of these reasons, might include small input data, insufficient parallelizable tasks, or serious bottlenecks in I/O such as reading from the S3. Thus, the configuration of 1 core and 2GB of memory has been found most useful for this workload.