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, lit
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") \
    .config("spark.driver.allowMultipleContexts", "true") \
    .config("spark.yarn.am.attemptFailuresValidityInterval", "1s") \
    .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)
```

```
# Extract the features field and select geometry and attributes
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)
crime_df = crime_df.dropDuplicates()

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2015
crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crime_df = crime_df.filter(year(col("DATE_OCC_TIMESTAMP")) == 2015)

# Create a geometry column from the LAT and LON fields in the crime
data
crime_df = crime_df.withColumn("crime_geometry", expr("ST_Point(LON,
LAT)"))

# Perform a spatial join between crime data and census data
joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"), "inner")
joined_df = joined_df.dropDuplicates()

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))

# Join the aggregated population data with the crime counts
crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner")

# Filter out rows where population is NULL or 0
crime_ratio_df = crime_ratio_df.filter((col("population
(2015)").isNotNull()) & (col("population (2015)") > 0))

# Calculate the crime-to-population ratio
```

```
crime_ratio_df = crime_ratio_df.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)

# Clean the Estimated Median Income column and convert it to a
numeric format
income_df = income_df.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"))

# Join census, crime, and income datasets
combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Analyze racial distribution by performing another join with race
codes
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)

# Join with racial codes for mapping racial descriptions
final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

# Aggregate racial data for high-income and low-income areas
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

# Filter data for top and bottom areas
top_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in top_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
)

bottom_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in bottom_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
)

# Sort results by number of victims
```

```
top_area_results = top_area_results.orderBy(col("#").desc())
bottom_area_results = bottom_area_results.orderBy(col("#").desc())

# Rename column "Vict Descent Full" to "Victim Descent"
top_area_results = top_area_results.withColumnRenamed("Vict Descent
Full", "Victim Descent")
bottom_area_results = bottom_area_results.withColumnRenamed("Vict
Descent Full", "Victim Descent")

# Display results
print("Top 3 High-Income Areas Racial Profile:")
top_area_results.show(truncate=False)

print("Bottom 3 Low-Income Areas Racial Profile:")
bottom_area_results.show(truncate=False)

end_time = time.time()

# Display results and execution 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?
3096	application_1732639283265_3052	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 |
|Unknown                     |55 |
|Black                       |45 |
|Other Asian                 |29 |
|Chinese                     |1  |
|American Indian/Alaskan Native|1  |
+----------------------------+---+

Bottom 3 Low-Income Areas Racial Profile:
+----------------------------+----+
|Victim Descent              |#   |
+----------------------------+----+
|Hispanic/Latin/Mexican      |1687|
|Black                       |1567|
|White                       |1028|
|Other                       |523 |
|Other Asian                 |127 |
|Unknown                     |27  |
|Korean                      |8   |
|American Indian/Alaskan Native|5   |
|Filipino                    |3   |
|Chinese                     |3   |
|Japanese                    |2   |
|Pacific Islander            |1   |
+----------------------------+----+

Execution Time with 1 Core, 2GB Memory: 123.7 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, lit
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") \
    .config("spark.driver.allowMultipleContexts", "true") \
    .config("spark.yarn.am.attemptFailuresValidityInterval", "1s") \
    .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)

# Extract the features field and select geometry and attributes
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)
crime_df = crime_df.dropDuplicates()

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2015
crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crime_df = crime_df.filter(year(col("DATE_OCC_TIMESTAMP")) == 2015)

# Create a geometry column from the LAT and LON fields in the crime
data
crime_df = crime_df.withColumn("crime_geometry", expr("ST_Point(LON,
LAT)"))

# Perform a spatial join between crime data and census data
joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"), "inner")
joined_df = joined_df.dropDuplicates()

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))
```

```
# Join the aggregated population data with the crime counts
crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner")

# Filter out rows where population is NULL or 0
crime_ratio_df = crime_ratio_df.filter((col("population
(2015)").isNotNull()) & (col("population (2015)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.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)

# Clean the Estimated Median Income column and convert it to a
numeric format
income_df = income_df.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"))

# Join census, crime, and income datasets
combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Analyze racial distribution by performing another join with race
codes
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)

# Join with racial codes for mapping racial descriptions
final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

# Aggregate racial data for high-income and low-income areas
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

# Filter data for top and bottom areas
top_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in top_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
```

```
)

bottom_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in bottom_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
)

# Sort results by number of victims
top_area_results = top_area_results.orderBy(col("#").desc())
bottom_area_results = bottom_area_results.orderBy(col("#").desc())

# Rename column "Vict Descent Full" to "Victim Descent"
top_area_results = top_area_results.withColumnRenamed("Vict Descent
Full", "Victim Descent")
bottom_area_results = bottom_area_results.withColumnRenamed("Vict
Descent Full", "Victim Descent")

# Display results
print("Top 3 High-Income Areas Racial Profile:")
top_area_results.show(truncate=False)

print("Bottom 3 Low-Income Areas Racial Profile:")
bottom_area_results.show(truncate=False)

end_time = time.time()

# Display results and execution time
execution_time = end_time - start_time
print(f"Execution Time with 2 Core, 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?
3098	application_1732639283265_3054	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 |
|Unknown                     |55 |
|Black                       |45 |
|Other Asian                 |29 |
|American Indian/Alaskan Native|1  |
|Chinese                     |1  |
+----------------------------+---+

Bottom 3 Low-Income Areas Racial Profile:
+----------------------------+----+
|Victim Descent              |#   |
+----------------------------+----+
|Hispanic/Latin/Mexican      |1687|
|Black                       |1567|
|White                       |1028|
|Other                       |523 |
|Other Asian                 |127 |
|Unknown                     |27  |
|Korean                      |8   |
|American Indian/Alaskan Native|5   |
|Chinese                     |3   |
|Filipino                    |3   |
|Japanese                    |2   |
|Pacific Islander            |1   |
+----------------------------+----+

Execution Time with 2 Core, 4GB Memory: 102.2 seconds
```

```
### 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, lit
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") \
    .config("spark.driver.allowMultipleContexts", "true") \
    .config("spark.yarn.am.attemptFailuresValidityInterval", "1s") \
    .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)

# Extract the features field and select geometry and attributes
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)
crime_df = crime_df.dropDuplicates()

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2015
crime_df = crime_df.withColumn("DATE_OCC_TIMESTAMP",
to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crime_df = crime_df.filter(year(col("DATE_OCC_TIMESTAMP")) == 2015)

# Create a geometry column from the LAT and LON fields in the crime
data
crime_df = crime_df.withColumn("crime_geometry", expr("ST_Point(LON,
LAT)"))

# Perform a spatial join between crime data and census data
joined_df = crime_df.join(census_df,
expr("ST_Intersects(crime_geometry, census_geometry)"), "inner")
joined_df = joined_df.dropDuplicates()

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(spark_sum("population").alias("populati
on (2015)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2015)"))
```

```
# Join the aggregated population data with the crime counts
crime_ratio_df = crime_counts.join(aggregated_population, "zip",
"inner")

# Filter out rows where population is NULL or 0
crime_ratio_df = crime_ratio_df.filter((col("population
(2015)").isNotNull()) & (col("population (2015)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.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)

# Clean the Estimated Median Income column and convert it to a
numeric format
income_df = income_df.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"))

# Join census, crime, and income datasets
combined_df = crime_ratio_df.join(income_df.select("zip",
"estimated_median_income (2015)"), "zip", "inner")

# Analyze racial distribution by performing another join with race
codes
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)

# Join with racial codes for mapping racial descriptions
final_joined_df = joined_df.join(race_codes_df, joined_df["Vict
Descent"] == race_codes_df["Vict Descent"], "inner")

# Aggregate racial data for high-income and low-income areas
top_areas = combined_df.orderBy(col("estimated_median_income
(2015)").desc()).limit(3)
bottom_areas = combined_df.orderBy("estimated_median_income
(2015)").limit(3)

# Filter data for top and bottom areas
top_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in top_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
```

```
)

bottom_area_results = final_joined_df.filter(
    col("zip").isin([row["zip"] for row in bottom_areas.collect()])
).groupBy("Vict Descent Full").agg(
    count("LOCATION").alias("#")
)

# Sort results by number of victims
top_area_results = top_area_results.orderBy(col("#").desc())
bottom_area_results = bottom_area_results.orderBy(col("#").desc())

# Rename column "Vict Descent Full" to "Victim Descent"
top_area_results = top_area_results.withColumnRenamed("Vict Descent
Full", "Victim Descent")
bottom_area_results = bottom_area_results.withColumnRenamed("Vict
Descent Full", "Victim Descent")

# Display results
print("Top 3 High-Income Areas Racial Profile:")
top_area_results.show(truncate=False)

print("Bottom 3 Low-Income Areas Racial Profile:")
bottom_area_results.show(truncate=False)

end_time = time.time()

# Display results and execution time
execution_time = end_time - start_time
print(f"Execution Time with 4 Core, 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?
3100	application_1732639283265_3056	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 |
|Unknown                      |55 |
|Black                        |45 |
|Other Asian                  |29 |
|American Indian/Alaskan Native|1  |
|Chinese                      |1  |
+-----------------------------+---+

Bottom 3 Low-Income Areas Racial Profile:
+-----------------------------+----+
|Victim Descent               |#   |
+-----------------------------+----+
|Hispanic/Latin/Mexican       |1687|
|Black                        |1567|
|White                        |1028|
|Other                        |523 |
|Other Asian                  |127 |
|Unknown                      |27  |
|Korean                       |8   |
|American Indian/Alaskan Native|5   |
|Filipino                     |3   |
|Chinese                      |3   |
|Japanese                     |2   |
|Pacific Islander             |1   |
+-----------------------------+----+

Execution Time with 4 Core, 8GB Memory: 116.25 seconds
```

Answer

It took 123.7 seconds with 1 core and 2GB of memory, while under 2 cores and 4GB memory, it was reduced to 102.2 as execution times. On the contrary, under 4 cores and 8GB memory, the execution time recorded was 116.25 seconds.

The performance increase from 1 to 2 cores is to be expected because extra computational resources facilitate better parallelization and speed up computing. The slowdown at 4 cores, however, signals diminishing returns or inefficient resource imposition spooling some workload's task synchronization overhead suboptimal Spark configurations, or prompted by IO bottlenecks due to reading large datasets from S3 and spatial operations performed on them.

2 cores and 4GB memory were the best resource, achieving the highest performance. Therefore, that is the configuration in which the demand is optimally aligned with resource availability. The use of higher resource configurations would then necessitate more optimizations, which may involve tuning data partitions, caching

intermediate results, or optimizing Sedona spatial operations.