Query 3

Using the 2010 Census population data and the 2015 Census household income data, calculate the following for each area of Los Angeles: The average annual income per person and the ratio of total crimes per person. The results should be summarized in a table.

Question 3

Implement Query 3 using DataFrame or SQL API. Use hint & explain methods to find out which join strategies the catalyst optimizer uses. Experiment by forcing Spark to use different strategies (between BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL) and comment on the results you observe.

Which of the available Spark join strategies is (are) the most appropriate and why?

```
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,
count, sum, avg, concat, lit, expr, to_timestamp, year
import time

# Create a Spark session and register Sedona
spark = SparkSession.builder \
    .appName("Query 3") \
    .getOrCreate()

# 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,
```

```python
inferSchema=True)

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2010
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")) == 2010)

# 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")

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(sum("population").alias("population
(2010)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2010)"))

# 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
(2010)").isNotNull()) & (col("population (2010)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.withColumn("crimes_per_person
(2010)", col("crimes (2010)") / col("population (2010)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
tulo_df = spark.read.csv(income_path, header=True, inferSchema=True)

# Clean the Estimated Median Income column and convert it to a
numeric format
tulo_df = tulo_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"))
```

```
start_time = time.time()

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

# Calculate the average income per person
combined_df = combined_df.withColumn("average_income_per_person",
col("estimated_median_income (2015)") / col("population (2010)"))

# Add a dollar sign to the average_income_per_person column without
rounding
final_df = combined_df.select(
    col("zip").alias("zip (LA)"),
    col("crimes (2010)"),
    col("population (2010)"),
    col("crimes_per_person (2010)"),
    concat(lit("$"), col("estimated_median_income
(2015)")).alias("estimated_income (2015)"),
    concat(lit("$"),
col("average_income_per_person")).alias("average_annual_per_person")
)

# Display the results
final_df.show(truncate=False)

end_time = time.time()
print(f"Time: {end_time - start_time} seconds")
```

```
+---------+-------------+-----------------+------------------------
+----------------------+------------------------+
|zip (LA)|crimes (2010)|population (2010)|crimes_per_person (2010)|
estimated_income (2015)|average_annual_per_person|
+---------+-------------+-----------------+------------------------
+----------------------+------------------------+
|90094    |136          |5464             |0.024890190336749635    |$104367.0
|$19.10084187408492       |
|90230    |452          |31766            |0.014229049927595543    |$74823.0
|$2.3554429263992946      |
|90293    |435          |12132            |0.03585558852621167     |$82055.0
|$6.763517969007583       |
|90292    |514          |21576            |0.023822766036336672    |$100507.0
|$4.658277715980719       |
|90291    |2457         |28341            |0.08669418863131152     |$80111.0
|$2.826682191877492       |
|90405    |47           |27186            |0.001728831015964099    |$77948.0
|$2.8672110645185023      |
|90045    |3299         |39480            |0.0835612968591692      |$75684.0
|$1.9170212765957446      |
|90066    |1755         |55277            |0.031749190440870524    |$68132.0
|$1.232556035964325       |
|90401    |19           |6722             |0.002826539720321333    |$62703.0
|$9.328027372805712       |
|90245    |7            |16654            |4.2031944277651017E-4   |$85727.0
|$5.147532124414555       |
|90266    |5            |35135            |1.4230823964707557E-4   |$143527.0
|$4.085014942365163       |
|90008    |3015         |32327            |0.09326569121786742     |$36564.0
|$1.1310669100133015      |
|90043    |2802         |44789            |0.0625600035723057      |$38180.0
|$0.8524414476768849      |
|90056    |27           |7827             |0.0034495975469528554   |$84099.0
|$10.74472978152549       |
|90301    |102          |36568            |0.002789323999124918    |$37424.0
|$1.0234084445416758      |
|90250    |2            |93193            |2.146083933342633E-5    |$46172.0
|$0.4954449368514803      |
|90278    |1            |40071            |2.4955703626063736E-5   |$107010.0
|$2.6705098450250806      |
|90304    |11           |28210            |3.899326479971641E-4    |$36412.0
|$1.2907479617157036      |
|90302    |31           |29415            |0.0010538840727519973   |$41426.0
|$1.4083290838007818      |
|90254    |2            |19506            |1.0253255408592229E-4   |$111187.0
|$5.70014354557572        |
+---------+-------------+-----------------+------------------------
+----------------------+------------------------+
only showing top 20 rows

Time: 15.450810194015503 seconds
```

```
BROADCAST Strategy
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,
count, sum, concat, lit, expr, to_timestamp, year, broadcast
import time

# Create a Spark session and register Sedona
spark = SparkSession.builder \
```

```python
    .appName("BROADCAST") \
    .getOrCreate()

# Register Sedona
SedonaRegistrator.registerAll(spark)

# 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)

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2010
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")) == 2010)

# 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")

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(sum("population").alias("population
(2010)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2010)"))
```

```python
# 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
(2010)").isNotNull()) & (col("population (2010)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.withColumn("crimes_per_person
(2010)", col("crimes (2010)") / col("population (2010)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
tulo_df = spark.read.csv(income_path, header=True, inferSchema=True)

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

# Measure the performance of the BROADCAST join
start_time = time.time()

# Use BROADCAST join to combine crime data and income data
combined_df = crime_ratio_df.join(
    broadcast(tulo_df.select("zip", "estimated_median_income
(2015)")),
    "zip",
    "inner"
)

# Calculate the average income per person
combined_df = combined_df.withColumn("average_income_per_person",
col("estimated_median_income (2015)") / col("population (2010)"))

# Add a dollar sign to the average_income_per_person column without
rounding
final_df = combined_df.select(
    col("zip").alias("zip (LA)"),
    col("crimes (2010)"),
    col("population (2010)"),
    col("crimes_per_person (2010)"),
    concat(lit("$"), col("estimated_median_income
(2015)")).alias("estimated_income (2015)"),
    concat(lit("$"),
col("average_income_per_person")).alias("average_annual_per_person")
```

```python
)

# Display the results
final_df.show(truncate=False)

end_time = time.time()
print(f"BROADCAST Join Time: {end_time - start_time} seconds")
```

```
+---------+-------------+-----------------+------------------------
+-----------------------+-------------------------+
|zip (LA)|crimes (2010)|population (2010)|crimes_per_person (2010)|
estimated_income (2015)|average_annual_per_person|
+---------+-------------+-----------------+------------------------
+-----------------------+-------------------------+
|90094    |136          |5464             |0.024890190336749635    |$104367.0
|$19.10084187408492       |
|90266    |5            |35135            |1.4230823964707557E-4   |$143527.0
|$4.085014942365163       |
|90230    |452          |31766            |0.014229049927595543    |$74823.0
|$2.3554429263992946      |
|90293    |435          |12132            |0.03585558852621167     |$82055.0
|$6.763517969007583       |
|90292    |514          |21576            |0.023822766036336672    |$100507.0
|$4.658277715980719       |
|90291    |2457         |28341            |0.08669418863131152     |$80111.0
|$2.826682191877492       |
|90405    |47           |27186            |0.001728831015964099    |$77948.0
|$2.8672110645185023      |
|90034    |2234         |57964            |0.038541163480781175    |$58004.0
|$1.0006900835001036      |
|90045    |3299         |39480            |0.0835612968591692      |$75684.0
|$1.9170212765957446      |
|90066    |1755         |55277            |0.031749190440870524    |$68132.0
|$1.232556035964325       |
|90401    |19           |6722             |0.002826539720321333    |$62703.0
|$9.328027372805712       |
|90245    |7            |16654            |4.2031944277651017E-4   |$85727.0
|$5.147532124414555       |
|90008    |3015         |32327            |0.09326569121786742     |$36564.0
|$1.1310669100133015      |
|90043    |2802         |44789            |0.0625600035723057      |$38180.0
|$0.8524414476768849      |
|90056    |27           |7827             |0.0034495975469528554   |$84099.0
|$10.74472978152549       |
|90047    |3114         |48606            |0.06406616467102827     |$39269.0
|$0.8079043739456034      |
|90301    |102          |36568            |0.002789323999124918    |$37424.0
|$1.0234084445416758      |
|90250    |2            |93193            |2.146083933342633E-5    |$46172.0
|$0.4954449368514803      |
|90304    |11           |28210            |3.899326479971641E-4    |$36412.0
|$1.2907479617157036      |
|90303    |2            |26176            |7.640586797066015E-5    |$39671.0
|$1.5155485941320292      |
+---------+-------------+-----------------+------------------------
+-----------------------+-------------------------+
only showing top 20 rows

BROADCAST Join Time: 14.191836595535278 seconds
```

```
MERGE Strategy
```

```python
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,
count, sum, concat, lit, expr, to_timestamp, year
import time

# Create a Spark session and register Sedona
spark = SparkSession.builder \
    .appName("MERGE") \
    .getOrCreate()

# Register Sedona for spatial operations
SedonaRegistrator.registerAll(spark)

# 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)

# Convert "DATE OCC" column to timestamp and filter crimes from 2010
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")) == 2010)

# Create a geometry column for crimes based on LAT and LON columns
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")

# Aggregate census data by ZIP code
aggregated_population =
```

```
census_df.groupBy("zip").agg(sum("population").alias("population
(2010)"))

# Count the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2010)"))

# Measure performance for the MERGE join
start_time = time.time()

# Use a MERGE join to combine crime data and census population data
crime_ratio_df = crime_counts.join(
    aggregated_population.hint("merge"),
    "zip",
    "inner"
)

# Remove rows where the population is NULL or zero
crime_ratio_df = crime_ratio_df.filter((col("population
(2010)").isNotNull()) & (col("population (2010)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.withColumn("crimes_per_person
(2010)", col("crimes (2010)") / col("population (2010)"))

# Load income data from a CSV file
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
tulo_df = spark.read.csv(income_path, header=True, inferSchema=True)

# Clean the Estimated Median Income column and convert it to numeric
tulo_df = tulo_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 crime, population, and income data
combined_df = crime_ratio_df.join(
    tulo_df.select("zip", "estimated_median_income (2015)"),
    "zip",
    "inner"
)

# Calculate the average income per person
combined_df = combined_df.withColumn("average_income_per_person",
col("estimated_median_income (2015)") / col("population (2010)"))

# Add a dollar sign to the average income and estimated income
columns
final_df = combined_df.select(
```

```
    col("zip").alias("zip (LA)"),
    col("crimes (2010)"),
    col("population (2010)"),
    col("crimes_per_person (2010)"),
    concat(lit("$"), col("estimated_median_income
(2015)")).alias("estimated_income (2015)"),
    concat(lit("$"),
col("average_income_per_person")).alias("average_annual_per_person")
)

# Display the results
final_df.show(truncate=False)

end_time = time.time()
print(f"MERGE Join Time: {end_time - start_time} seconds")
```

```
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
|zip (LA)|crimes (2010)|population (2010)|crimes_per_person (2010)|
estimated_income (2015)|average_annual_per_person|
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
|90001   |794          |57110            |0.013902994221677465    |$33887.0
|$0.593363684118368      |
|90002   |2492         |51223            |0.04865002049860414     |$30413.0
|$0.5937371883724109     |
|90003   |6132         |66266            |0.0925361422147104      |$30805.0
|$0.46486886185977727    |
|90004   |2991         |62180            |0.04810228369250563     |$40612.0
|$0.6531360566098424     |
|90005   |1700         |37681            |0.04511557548897322     |$31142.0
|$0.8264642658103554     |
|90006   |2891         |59185            |0.0488468361916026      |$31521.0
|$0.5325842696629214     |
|90007   |2798         |40920            |0.06837732160312805     |$22304.0
|$0.5450635386119257     |
|90008   |3015         |32327            |0.09326569121786742     |$36564.0
|$1.1310669100133015     |
|90010   |733          |3800             |0.19289473684210526     |$45786.0
|$12.048947368421052     |
|90011   |5288         |103892           |0.05089901051091518     |$30251.0
|$0.291177376506372      |
|90012   |1680         |31103            |0.054014082242870465    |$31576.0
|$1.0152075362505224     |
|90013   |2059         |11772            |0.17490655793408086     |$19887.0
|$1.6893476044852191     |
|90014   |858          |7005             |0.12248394004282655     |$23642.0
|$3.3750178443968593     |
|90015   |2607         |18986            |0.1373117033603708      |$29684.0
|$1.5634678183924997     |
|90016   |2912         |47596            |0.06118161190015968     |$38330.0
|$0.8053197747709891     |
|90017   |1984         |23768            |0.0834735779198923      |$22754.0
|$0.9573375967687647     |
|90018   |2649         |49310            |0.05372135469478807     |$33864.0
|$0.6867572500506997     |
|90019   |3100         |64458            |0.048093332092215085    |$46571.0
|$0.722501473827919      |
|90020   |1216         |38967            |0.031205892165165398    |$38849.0
|$0.9969717966484462     |
|90021   |1306         |3951             |0.33054922804353326     |$12813.0
|$3.242976461655277      |
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
only showing top 20 rows

MERGE Join Time: 14.373335599899292 seconds
```

```
SHUFFLE_HASH Strategy
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,
count, sum, concat, lit, expr, to_timestamp, year
import time

# Create a Spark session and register Sedona
spark = SparkSession.builder \
```

```
    .appName("SHUFFLE_HASH") \
    .getOrCreate()

# Register Sedona
SedonaRegistrator.registerAll(spark)

# 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)

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2010
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")) == 2010)

# 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")

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(sum("population").alias("population
(2010)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2010)"))
```

```python
# Measure the performance of the SHUFFLE_HASH join
start_time = time.time()

# Use SHUFFLE_HASH join to combine crime and population data
crime_ratio_df = crime_counts.join(
    aggregated_population.hint("shuffle_hash"),
    "zip",
    "inner"
)

# Filter out rows where population is NULL or 0
crime_ratio_df = crime_ratio_df.filter((col("population
(2010)").isNotNull()) & (col("population (2010)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.withColumn("crimes_per_person
(2010)", col("crimes (2010)") / col("population (2010)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
tulo_df = spark.read.csv(income_path, header=True, inferSchema=True)

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

# Combine crime, population, and income data
combined_df = crime_ratio_df.join(
    tulo_df.select("zip", "estimated_median_income (2015)"),
    "zip",
    "inner"
)

# Calculate the average income per person
combined_df = combined_df.withColumn("average_income_per_person",
col("estimated_median_income (2015)") / col("population (2010)"))

# Add a dollar sign to the average_income_per_person column without
rounding
final_df = combined_df.select(
    col("zip").alias("zip (LA)"),
    col("crimes (2010)"),
    col("population (2010)"),
    col("crimes_per_person (2010)"),
    concat(lit("$"), col("estimated_median_income
(2015)")).alias("estimated_income (2015)"),
```

```
    concat(lit("$"),
col("average_income_per_person")).alias("average_annual_per_person")
)

# Display the results
final_df.show(truncate=False)

end_time = time.time()
print(f"SHUFFLE_HASH Join Time: {end_time - start_time} seconds")
```

```
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
|zip (LA)|crimes (2010)|population (2010)|crimes_per_person (2010)|
estimated_income (2015)|average_annual_per_person|
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
|90008    |3015         |32327            |0.09326569121786742     |$36564.0
|$1.1310669100133015      |
|90064    |1426         |25403            |0.056135102153288985    |$87283.0
|$3.4359327638467896      |
|90062    |2395         |32821            |0.07297157307821212     |$34588.0
|$1.053837482099875       |
|90011    |5288         |103892           |0.05089901051091518     |$30251.0
|$0.291177376506372       |
|90021    |1306         |3951             |0.33054922804353326     |$12813.0
|$3.242976461655277       |
|90405    |47           |27186            |0.001728831015964099    |$77948.0
|$2.8672110645185023      |
|90007    |2798         |40920            |0.06837732160312805     |$22304.0
|$0.5450635386119257      |
|90034    |2234         |57964            |0.038541163480781175    |$58004.0
|$1.0006900835001036      |
|90037    |4458         |62276            |0.07158455905966986     |$27179.0
|$0.43642815852013617     |
|90066    |1755         |55277            |0.031749190440870524    |$68132.0
|$1.232556035964325       |
|90401    |19           |6722             |0.002826539720321333    |$62703.0
|$9.328027372805712       |
|90403    |1            |24525            |4.077471967380224E-5    |$78151.0
|$3.186585117227319       |
|90016    |2912         |47596            |0.06118161190015968     |$38330.0
|$0.8053197747709891      |
|90232    |78           |15149            |0.00514885470988184     |$77004.0
|$5.083107795894119       |
|90404    |15           |21360            |7.022471910112359E-4    |$66623.0
|$3.1190543071161048      |
|90018    |2649         |49310            |0.05372135469478807     |$33864.0
|$0.6867572500506997      |
|90248    |334          |9947             |0.033577963204986426    |$53306.0
|$5.359002714386247       |
|90044    |4563         |89779            |0.050824803127680195    |$29206.0
|$0.3253099277113802      |
|90745    |25           |57251            |4.3667359522104416E-4   |$71443.0
|$1.2478908665350823      |
|90061    |1775         |26872            |0.06605388508484668     |$33731.0
|$1.255247097350402       |
+---------+-------------+-----------------+-----------------------
+----------------------+------------------------+
only showing top 20 rows

SHUFFLE_HASH Join Time: 13.013461351394653 seconds
```

```
SHUFFLE_REPLICATE_NL Strategy
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,
count, sum, concat, lit, expr, to_timestamp, year
import time

# Create a Spark session and register Sedona
spark = SparkSession.builder \
```

```
    .appName("SHUFFLE_REPLICATE_NL") \
    .getOrCreate()

# Register Sedona
SedonaRegistrator.registerAll(spark)

# 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)

# Convert "DATE OCC" to a timestamp and filter for crimes that
occurred in 2010
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")) == 2010)

# 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")

# Aggregate census data by ZIP code
aggregated_population =
census_df.groupBy("zip").agg(sum("population").alias("population
(2010)"))

# Calculate the total number of crimes by ZIP code
crime_counts =
joined_df.groupBy("zip").agg(count("LOCATION").alias("crimes
(2010)"))
```

```
# Measure the performance of the SHUFFLE_REPLICATE_NL join
start_time = time.time()

# Use SHUFFLE_REPLICATE_NL join to combine crime and population data
crime_ratio_df = crime_counts.join(
    aggregated_population.hint("shuffle_replicate_nl"),
    "zip",
    "inner"
)

# Filter out rows where population is NULL or 0
crime_ratio_df = crime_ratio_df.filter((col("population
(2010)").isNotNull()) & (col("population (2010)") > 0))

# Calculate the crime-to-population ratio
crime_ratio_df = crime_ratio_df.withColumn("crimes_per_person
(2010)", col("crimes (2010)") / col("population (2010)"))

# Load income data
income_path = "s3://initial-notebook-data-bucket-dblab-905418150721/
LA_income_2015.csv"
tulo_df = spark.read.csv(income_path, header=True, inferSchema=True)

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

# Combine crime, population, and income data
combined_df = crime_ratio_df.join(
    tulo_df.select("zip", "estimated_median_income (2015)"),
    "zip",
    "inner"
)

# Calculate the average income per person
combined_df = combined_df.withColumn("average_income_per_person",
col("estimated_median_income (2015)") / col("population (2010)"))

# Add a dollar sign to the average_income_per_person column without
rounding
final_df = combined_df.select(
    col("zip").alias("zip (LA)"),
    col("crimes (2010)"),
    col("population (2010)"),
    col("crimes_per_person (2010)"),
    concat(lit("$"), col("estimated_median_income
(2015)")).alias("estimated_income (2015)"),
```

```
    concat(lit("$"),
col("average_income_per_person")).alias("average_annual_per_person")
)

# Display the results
final_df.show(truncate=False)

end_time = time.time()
print(f"SHUFFLE_REPLICATE_NL Join Time: {end_time - start_time}
seconds")
```

```
+---------+-------------+-----------------+-----------------------
+-----------------------+-------------------------+
|zip (LA)|crimes (2010)|population (2010)|crimes_per_person (2010)|
estimated_income (2015)|average_annual_per_person|
+---------+-------------+-----------------+-----------------------
+-----------------------+-------------------------+
|90094   |136          |5464             |0.024890190336749635    |$104367.0
|$19.10084187408492        |
|90293   |435          |12132            |0.03585558852621167     |$82055.0
|$6.763517969007583        |
|90292   |514          |21576            |0.023822766036336672    |$100507.0
|$4.658277715980719        |
|90291   |2457         |28341            |0.08669418863131152     |$80111.0
|$2.826682191877492        |
|90405   |47           |27186            |0.001728831015964099    |$77948.0
|$2.8672110645185023       |
|90045   |3299         |39480            |0.0835612968591692      |$75684.0
|$1.9170212765957446       |
|90066   |1755         |55277            |0.031749190440870524    |$68132.0
|$1.232556035964325        |
|90401   |19           |6722             |0.002826539720321333    |$62703.0
|$9.328027372805712        |
|90266   |5            |35135            |1.4230823964707557E-4   |$143527.0
|$4.085014942365163        |
|90008   |3015         |32327            |0.09326569121786742     |$36564.0
|$1.1310669100133015       |
|90043   |2802         |44789            |0.0625600035723057      |$38180.0
|$0.8524414476768849       |
|90056   |27           |7827             |0.0034495975469528554   |$84099.0
|$10.74472978152549        |
|90230   |452          |31766            |0.014229049927595543    |$74823.0
|$2.3554429263992946       |
|90301   |102          |36568            |0.002789323999124918    |$37424.0
|$1.0234084445416758       |
|90250   |2            |93193            |2.146083933342633E-5    |$46172.0
|$0.4954449368514803       |
|90278   |1            |40071            |2.4955703626063736E-5   |$107010.0
|$2.6705098450250806       |
|90304   |11           |28210            |3.899326479971641E-4    |$36412.0
|$1.2907479617157036       |
|90302   |31           |29415            |0.0010538840727519973   |$41426.0
|$1.4083290838007818       |
|90254   |2            |19506            |1.0253255408592229E-4   |$111187.0
|$5.70014354557572         |
|90245   |7            |16654            |4.2031944277651017E-4   |$85727.0
|$5.147532124414555        |
+---------+-------------+-----------------+-----------------------
+-----------------------+-------------------------+
only showing top 20 rows

SHUFFLE_REPLICATE_NL Join Time: 12.394477605819702 seconds
```

Answer

Based on the time of execution for joining these datasets, SHUFFLE_REPLICATE_NL and SHUFFLE_HASH are probably the two best joining methods for that dataset and workload. The time of 12.39 seconds for SHUFFLE_REPLICATE_NL demonstrates that replication of smaller datasets across partitions becomes highly effective in this case. It is particularly effective when the one dataset is relatively small compared to the other, enabling replication without imposing heavy overheads on memory space or through the network. For example, SHUFFLE_HASH also performed exceptionally well in 13.01 seconds, taking advantage of hash-based partitioning to evenly distribute the data across the cluster. The approach is most needed when the join keys are evenly distributed, thus minimizing shuffle cost and skew.

Broadcast Join's time is 14.19 seconds, which also reflects a very good performance with the small-to-fit memory broadcast dataset, thus making it feasible for a dramatically reduced dataset against another during in-memory joins. MERGE Join completed in 14.37 seconds and was still worth considering as long as the datasets were sorted on the join keys since it avoided most of the shuffle and memory overhead. Finally, the Last but not least: Standard Join took 15.45 seconds, the slowest one among all joins, stressing the absence of optimizations such as broadcasting, shuffling, or merging, thus making it unfit for bigger datasets.

In sum, the time parameter of 12.39 seconds makes SHUFFLE_REPLICATE_NL the most appropriate strategy with superior performance to efficiently handling the workload. SHUFFLE_HASH, at 13.01 seconds, is a very strong secondary choice, especially with those dataset characteristics that match well with hash-based partitioning methods. Broadcast Join at 14.1 can be considered where there are smaller datasets or memory constraint situations.