Query 5

Calculate, for each police station, the number of crimes that took place closest to it, as well as its average distance from the locations where the specific incidents occurred. The results should be displayed sorted by number of incidents, in descending order

Question 5

Implement Query 5 using the DataFrame or SQL API. Execute your implementation using a total of 8 cores and 16GB of memory with the following configurations:

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

Comment on the results.

```
### 2 Executors × 4 Cores/8GB Memory ###

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, avg, expr, row_number,
round
from pyspark.sql.window import Window
from sedona.sql.types import GeometryType
from sedona.register.geo_registrator import SedonaRegistrator
import time

start_time = time.time()

# Paths to data
crime_data_2010_2019 = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_data_2020_present = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2020_to_Present_20241101.csv"
police_stations_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/LA_Police_Stations.csv"

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Query 5 - 2 Executors × 4 Cores/8GB Memory") \
    .config("spark.executor.instances", 2) \
    .config("spark.executor.cores", 4) \
    .config("spark.executor.memory", "8g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

# Load and clean crime data
```

```
crime_df_2010_2019 = spark.read.csv(crime_data_2010_2019,
header=True, inferSchema=True)
crime_df_2020_present = spark.read.csv(crime_data_2020_present,
header=True, inferSchema=True)
crime_data =
crime_df_2010_2019.union(crime_df_2020_present).dropDuplicates()
crime_data = crime_data.filter((col("LAT") != 0) & (col("LON") !=
0))
crime_data = crime_data.filter(col("LAT").isNotNull() &
col("LON").isNotNull())

# Load and clean police stations data
police_stations_df = spark.read.csv(police_stations_path,
header=True, inferSchema=True)
police_stations_df = police_stations_df.filter(col("X").isNotNull()
& col("Y").isNotNull())

# Create geometry columns
crime_data = crime_data.withColumn("crime_geometry",
expr("ST_Point(LON, LAT)"))
police_stations_df =
police_stations_df.withColumn("station_geometry", expr("ST_Point(X,
Y)"))

# Prepare police stations broadcast dataframe
police_stations_broadcast = police_stations_df.select(
    col("station_geometry"),
    col("DIVISION").alias("division")
)

# Perform spatial join
joined_df = crime_data.crossJoin(police_stations_broadcast) \
    .withColumn("distance", expr("ST_Distance(crime_geometry,
station_geometry)")) \
    .withColumn("distance_km", col("distance") * 111.32)

# Select the nearest police station for each crime
window_spec = Window.partitionBy("DR_NO").orderBy("distance_km")
nearest_station_df = joined_df.withColumn("rank",
row_number().over(window_spec)) \
                              .filter(col("rank") == 1)

# Aggregate results
result = nearest_station_df.groupBy("division") \
    .agg(
        round(avg("distance_km"), 3).alias("average_distance"),
        count("DR_NO").alias("#")
    ) \
    .orderBy(col("#").desc())

# Show results
result.show(truncate=False)
execution_time = time.time() - start_time
```

```
# Print execution time
print(f"Execution Time (2 Executors x 4 Cores/8GB Memory):
{execution_time:.2f} seconds")

# Stop SparkSession
spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3135	application_1732639283265_3091	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.
```

```
+----------------+----------------+------+
|division        |average_distance|#     |
+----------------+----------------+------+
|HOLLYWOOD       |2.275           |213080|
|VAN NUYS        |3.19            |211457|
|WILSHIRE        |2.929           |198150|
|SOUTHWEST       |2.402           |186742|
|OLYMPIC         |1.925           |180463|
|NORTH HOLLYWOOD |2.907           |171159|
|77TH STREET     |1.846           |167323|
|PACIFIC         |4.174           |157468|
|CENTRAL         |1.099           |154474|
|SOUTHEAST       |2.688           |151999|
|RAMPART         |1.64            |149675|
|TOPANGA         |3.611           |147167|
|WEST VALLEY     |3.225           |130933|
|HARBOR          |3.862           |126749|
|FOOTHILL        |4.593           |122515|
|WEST LOS ANGELES|3.322           |121074|
|HOLLENBECK      |2.94            |119329|
|NEWTON          |1.769           |109078|
|MISSION         |3.9             |109009|
|NORTHEAST       |4.35            |105687|
+----------------+----------------+------+
only showing top 20 rows

Execution Time (2 Executors x 4 Cores/8GB Memory): 37.16 seconds
```

```
### 4 Executors × 2 Cores/4GB Memory ###

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, avg, expr, row_number,
round
from pyspark.sql.window import Window
from sedona.sql.types import GeometryType
from sedona.register.geo_registrator import SedonaRegistrator
```

```
import time

start_time = time.time()

# Paths to data
crime_data_2010_2019 = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_data_2020_present = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2020_to_Present_20241101.csv"
police_stations_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/LA_Police_Stations.csv"

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Query 5 - 4 Executors × 2 Cores/4GB Memory") \
    .config("spark.executor.instances", 4) \
    .config("spark.executor.cores", 2) \
    .config("spark.executor.memory", "4g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

# Load and clean crime data
crime_df_2010_2019 = spark.read.csv(crime_data_2010_2019,
header=True, inferSchema=True)
crime_df_2020_present = spark.read.csv(crime_data_2020_present,
header=True, inferSchema=True)
crime_data =
crime_df_2010_2019.union(crime_df_2020_present).dropDuplicates()
crime_data = crime_data.filter((col("LAT") != 0) & (col("LON") !=
0))
crime_data = crime_data.filter(col("LAT").isNotNull() &
col("LON").isNotNull())

# Load and clean police stations data
police_stations_df = spark.read.csv(police_stations_path,
header=True, inferSchema=True)
police_stations_df = police_stations_df.filter(col("X").isNotNull()
& col("Y").isNotNull())

# Create geometry columns
crime_data = crime_data.withColumn("crime_geometry",
expr("ST_Point(LON, LAT)"))
police_stations_df =
police_stations_df.withColumn("station_geometry", expr("ST_Point(X,
Y)"))

# Prepare police stations broadcast dataframe
police_stations_broadcast = police_stations_df.select(
    col("station_geometry"),
    col("DIVISION").alias("division")
)
```

```
# Perform spatial join
joined_df = crime_data.crossJoin(police_stations_broadcast) \
    .withColumn("distance", expr("ST_Distance(crime_geometry,
station_geometry)")) \
    .withColumn("distance_km", col("distance") * 111.32)

# Select the nearest police station for each crime
window_spec = Window.partitionBy("DR_NO").orderBy("distance_km")
nearest_station_df = joined_df.withColumn("rank",
row_number().over(window_spec)) \
                              .filter(col("rank") == 1)

# Aggregate results
result = nearest_station_df.groupBy("division") \
    .agg(
        round(avg("distance_km"), 3).alias("average_distance"),
        count("DR_NO").alias("#")
    ) \
    .orderBy(col("#").desc())

# Show results
result.show(truncate=False)
execution_time = time.time() - start_time

# Print execution time
print(f"Execution Time (4 Executors x 2 Cores/4GB Memory):
{execution_time:.2f} seconds")

# Stop SparkSession
spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3137	application_1732639283265_3093	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.
```

```
+----------------+----------------+------+
|division        |average_distance|#     |
+----------------+----------------+------+
|HOLLYWOOD       |2.275           |213080|
|VAN NUYS        |3.19            |211457|
|WILSHIRE        |2.929           |198150|
|SOUTHWEST       |2.402           |186742|
|OLYMPIC         |1.925           |180463|
|NORTH HOLLYWOOD |2.907           |171159|
|77TH STREET     |1.846           |167323|
|PACIFIC         |4.174           |157468|
|CENTRAL         |1.099           |154474|
|SOUTHEAST       |2.688           |151999|
|RAMPART         |1.64            |149675|
|TOPANGA         |3.611           |147167|
|WEST VALLEY     |3.225           |130933|
|HARBOR          |3.862           |126749|
|FOOTHILL        |4.593           |122515|
|WEST LOS ANGELES|3.322           |121074|
|HOLLENBECK      |2.94            |119329|
|NEWTON          |1.769           |109078|
|MISSION         |3.9             |109009|
|NORTHEAST       |4.35            |105687|
+----------------+----------------+------+
only showing top 20 rows

Execution Time (4 Executors x 2 Cores/4GB Memory): 40.30 seconds
```

```
### 8 Executors × 1 Cores/2GB Memory ###

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, avg, expr, row_number,
round
from pyspark.sql.window import Window
from sedona.sql.types import GeometryType
from sedona.register.geo_registrator import SedonaRegistrator
import time

start_time = time.time()

# Paths to data
crime_data_2010_2019 = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_data_2020_present = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2020_to_Present_20241101.csv"
police_stations_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/LA_Police_Stations.csv"

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Query 5 - 8 Executors × 1 Cores/2GB Memory") \
    .config("spark.executor.instances", 8) \
```

```
    .config("spark.executor.cores", 1) \
    .config("spark.executor.memory", "2g") \
    .getOrCreate()

SedonaRegistrator.registerAll(spark)

# Load and clean crime data
crime_df_2010_2019 = spark.read.csv(crime_data_2010_2019,
header=True, inferSchema=True)
crime_df_2020_present = spark.read.csv(crime_data_2020_present,
header=True, inferSchema=True)
crime_data =
crime_df_2010_2019.union(crime_df_2020_present).dropDuplicates()
crime_data = crime_data.filter((col("LAT") != 0) & (col("LON") !=
0))
crime_data = crime_data.filter(col("LAT").isNotNull() &
col("LON").isNotNull())

# Load and clean police stations data
police_stations_df = spark.read.csv(police_stations_path,
header=True, inferSchema=True)
police_stations_df = police_stations_df.filter(col("X").isNotNull()
& col("Y").isNotNull())

# Create geometry columns
crime_data = crime_data.withColumn("crime_geometry",
expr("ST_Point(LON, LAT)"))
police_stations_df =
police_stations_df.withColumn("station_geometry", expr("ST_Point(X,
Y)"))

# Prepare police stations broadcast dataframe
police_stations_broadcast = police_stations_df.select(
    col("station_geometry"),
    col("DIVISION").alias("division")
)

# Perform spatial join
joined_df = crime_data.crossJoin(police_stations_broadcast) \
    .withColumn("distance", expr("ST_Distance(crime_geometry,
station_geometry)")) \
    .withColumn("distance_km", col("distance") * 111.32)

# Select the nearest police station for each crime
window_spec = Window.partitionBy("DR_NO").orderBy("distance_km")
nearest_station_df = joined_df.withColumn("rank",
row_number().over(window_spec)) \
                              .filter(col("rank") == 1)

# Aggregate results
result = nearest_station_df.groupBy("division") \
    .agg(
        round(avg("distance_km"), 3).alias("average_distance"),
        count("DR_NO").alias("#")
```

```
    ) \
    .orderBy(col("#").desc())

# Show results
result.show(truncate=False)
execution_time = time.time() - start_time

# Print execution time
print(f"Execution Time (8 Executors x 1 Cores/2GB Memory):
{execution_time:.2f} seconds")

# Stop SparkSession
spark.stop()
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3138	application_1732639283265_3094	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.

+----------------+----------------+------+
|division        |average_distance|#     |
+----------------+----------------+------+
|HOLLYWOOD       |2.275           |213080|
|VAN NUYS        |3.19            |211457|
|WILSHIRE        |2.929           |198150|
|SOUTHWEST       |2.402           |186742|
|OLYMPIC         |1.925           |180463|
|NORTH HOLLYWOOD |2.907           |171159|
|77TH STREET     |1.846           |167323|
|PACIFIC         |4.174           |157468|
|CENTRAL         |1.099           |154474|
|SOUTHEAST       |2.688           |151999|
|RAMPART         |1.64            |149675|
|TOPANGA         |3.611           |147167|
|WEST VALLEY     |3.225           |130933|
|HARBOR          |3.862           |126749|
|FOOTHILL        |4.593           |122515|
|WEST LOS ANGELES|3.322           |121074|
|HOLLENBECK      |2.94            |119329|
|NEWTON          |1.769           |109078|
|MISSION         |3.9             |109009|
|NORTHEAST       |4.35            |105687|
+----------------+----------------+------+
only showing top 20 rows

Execution Time (8 Executors x 1 Cores/2GB Memory): 29.73 seconds
```

Answer

The best run time was 29.73 seconds when set to 8 executors of 1 core and 2GB's memory. In this setup, increased parallelism was efficient in spreading the work across a higher number of executors despite limited resources per executor.

The configuration of two executors with four cores and 8GB of memory delivered an average execution time of 37.16 seconds. Although much resource was given to each executor, it could not beat the result achieved by the 8-executor combination, possibly owing to lesser-efficient parallelism.

It was the 4-executor system that actually exhibited the slowest performance at 40.30 seconds. It had 2 cores, and each of the 4 executors had 4GB of memory.Although there was a balance between the parallelism and resource allocation , this was not as efficient in terms of resource allocation compared to the other setups.

So it appears that, under this workload and dataset, increasing the number of smaller executors maximises performance.