Query 1

Sort, in descending order, the age groups of victims in incidents involving any form of “aggravated assault” (i.e., include this term in the relevant description). Consider the following age groups: Children: < 18, Young adults: 18 – 24, Adults: 25 – 64, Elderly: >64

Question 1

Implement Query 1 using the DataFrame and RDD APIs. Run both implementations with 4 Spark executors. Is there a performance difference between the two APIs? Justify your answer. (20%)

```
### DataFrame API ###
from pyspark.sql import SparkSession
from pyspark.sql.functions import when, col, count
import time

# Measure the full execution time
start_time = time.time()

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Query 1") \
    .getOrCreate()

# Read both CSV files into DataFrames
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"

# Read both CSV files into DataFrames
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)

# Combine the two DataFrames and remove duplicates
crime_data =
crime_df_2010_2019.union(crime_df_2020_present).dropDuplicates()

# Filter records for "AGGRAVATED ASSAULT" in the column "Crm Cd
Desc"
filtered_data = crime_data.filter(col("Crm Cd
Desc").contains("AGGRAVATED ASSAULT"))

# Add a new column for age groups based on "Vict Age"
filtered_data = filtered_data.withColumn(
    "Age Group",
    when(col("Vict Age") < 18, "Children (<18)") \
```

```
    .when((col("Vict Age") >= 18) & (col("Vict Age") <= 24), "Young
adults (18-24)") \
    .when((col("Vict Age") >= 25) & (col("Vict Age") <= 64), "Adults
(25-64)") \
    .when(col("Vict Age") > 64, "Elderly (>64)") \
    .otherwise("Not Known")
)

# Group by "Age Group" and count occurrences, then sort by count in
descending order
age_group_counts = filtered_data.groupBy("Age
Group").agg(count("*").alias("Count")).orderBy(col("Count").desc())

# Trigger Spark execution and display results
age_group_counts.show()

# Measure total execution time
end_time = time.time()
print(f"DataFrame API took: {end_time - start_time:.2f} seconds")
```

```
+--------------------+------+
|           Age Group| Count|
+--------------------+------+
|      Adults (25-64)|121093|
|Young adults (18-24)| 33605|
|      Children (<18)| 15928|
|       Elderly (>64)|  5985|
+--------------------+------+
DataFrame API took: 7.17 seconds
```

```
### RDD API ###
from pyspark.sql import SparkSession
import time

# Measure the full execution time
start_time = time.time()

# Create a SparkSession
spark = SparkSession.builder \
    .appName("Query 1 - RDD") \
    .getOrCreate()

# S3 paths for the CSV files
crime_data_2010_2019_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2010_to_2019_20241101.csv"
crime_data_2020_present_path = "s3://initial-notebook-data-bucket-
dblab-905418150721/CrimeData/
Crime_Data_from_2020_to_Present_20241101.csv"

# Read both CSV files into RDDs
crime_data_2010_2019_rdd = spark.read.csv(crime_data_2010_2019_path,
header=True, inferSchema=True).rdd
crime_data_2020_present_rdd =
```

```
spark.read.csv(crime_data_2020_present_path, header=True,
inferSchema=True).rdd

# Combine the two RDDs and remove duplicates
crime_data_rdd =
crime_data_2010_2019_rdd.union(crime_data_2020_present_rdd).distinct
()

# Filter records for "AGGRAVATED ASSAULT" in the column "Crm Cd
Desc"
filtered_rdd = crime_data_rdd.filter(lambda row: "AGGRAVATED
ASSAULT" in str(row["Crm Cd Desc"]))

# Define age group categorization
def categorize_age(row):
    age = row["Vict Age"]
    if age is None:
        return "Not Known"
    elif age < 18:
        return "Children (<18)"
    elif 18 <= age <= 24:
        return "Young adults (18-24)"
    elif 25 <= age <= 64:
        return "Adults (25-64)"
    elif age > 64:
        return "Elderly (>64)"
    else:
        return "Not Known"

# Map rows to age groups and count occurrences
age_groups_rdd = filtered_rdd.map(lambda row: (categorize_age(row),
1))
age_group_counts_rdd = age_groups_rdd.reduceByKey(lambda a, b: a +
b).sortBy(lambda x: x[1], ascending=False)

# Perform the action and measure time
for age_group, count in age_group_counts_rdd.collect():
    print(f"{age_group}: {count}")

# Calculate total execution time
end_time = time.time()
print(f"RDD API took: {end_time - start_time:.2f} seconds")
```

```
 Adults (25-64): 121093
 Young adults (18-24): 33605
 Children (<18): 15928
 Elderly (>64): 5985

 RDD API took: 19.73 seconds
```

Answer

There is a significant performance difference between the two APIs. In this case, the DataFrame API performed much faster (7.17 seconds)

compared to the RDD API (19.73 seconds). In most cases, the DataFrame API utilizes Spark's Catalyst optimizer and Tungsten execution engine to implement advanced query optimizations, generate efficient execution plans, and better memory management. All these work together to minimize unnecessary computations and lower execution times.

By contrast, RDD API is at an even lower level of abstraction and thus does not benefit from any of these optimizations. Directly processes data, but requires manually handling most operations, which tends to be inefficient for comparatively large datasets or complex transformations.

Therefore, DataFrame API generally performs better in terms of performance, scalability, and usability within Spark applications. The RDD API is usually very useful for low-level operations or in cases where there is a need for extremely fine granularity. However, this set of functions is lagging in terms of execution speed in general for data-processing tasks.