Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Κωνσταντίνος Πίκουλας (03120112) — konpik2002@gmail.com
Κωνσταντίνα Μαυρουκάκη-Καραγκούνη (03120063) – constantina.mavroukaki@gmail.com

Ιανουάριος 2025

Αποθετήριο GitHub.

Περιεχόμενα

1 Ζητούμενο 1

Στο ζητούμενο αυτό, καλούμαστε να χρησιμοποιήσουμε το DataFrame και το RDD API του PySpark, έτσι ώστε να ταξινομήσουμε σε φθίνουσα σειρά τα περιστατικά "Βαριάς Σωματικής Βλάβης", κατηγοριοποιημένα βάσει της ηλικιακής ομάδα στην οποία ανήκουν. Για την εκτέλεση του query, χρησιμοποιήθηκαν 4 `Spark Executors`.

Πρώτα απ' όλα, δημιουργήσαμε ένα `SparkSession`, και αντλήσαμε τα δεδομένα από τα δύο csv αρχεία που αφορούν τα εγκλήματα. Προκειμένου να είναι ενιαία προσπελάσιμα, εκτελέσαμε την πράξη `union`.

```
### QUERY 1
APP_NAME = "Crime Victime Age Analysis"
SPARK_EXECUTORS = 4
spark = SparkSession.builder.appName(APP_NAME).config("spark.executor.instances", SPARK_EXECUTORS).
    getOrCreate()

# crime data
d1 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2010_to_2019_20241101.csv",
    header=True, inferSchema=True)
d2 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2020_to_Present_20241101.csv",
    header=True, inferSchema=True)
crime_data = d1.union(d2)
```

Αρχικά, χρησιμοποιούμε το DataFrame API προκειμένου να εκτελέσουμε το Query 1. Απομονώνουμε τα δεδομένα βάσει της περιγραφής του εγκλήματος, έτσι ώστε να κρατήσουμε μόνο δεδομένα που αφορούν "Βαριά Σωματική Βλάβη". Έπειτα, προσθέτουμε μια στήλη στα δεδομένα που αφορά το `Age Group` του θύματος, ως εξής:

- Παιδιά: < 18
- Νεαροί ενήλικοι: 18-24
- Ενήλικοι: 25-64
- Ηλικιωμένοι: > 64

Ομαδοποιούμε τα δεδομένα βάσει του `Age Group` και τέλος επιστρέφουμε τον αριθμό των θυμάτων για κάθε ηλικιακή ομάδα σε φθίνουσα σειρά.

```
# DATAFRAME API BEGIN #
start = time.time()
filtered_df = crime_data.filter(crime_data["Crm Cd Desc"].contains("AGGRAVATED ASSAULT")) #get only
    aggravated assault
grouped_df = filtered_df.withColumn("AGE GROUP", when(filtered_df['Vict Age'] <= 0, "Unknown").when(
    filtered_df['Vict Age'] < 18, "Children").when(filtered_df["Vict Age"] <= 24, "Young adults")
                            .when(filtered_df['Vict Age'] <= 64, "Adults").when(filtered_df['Vict
    Age'] > 64, "Elderly"))
categories_df = grouped_df.groupBy("AGE GROUP").count().orderBy(desc('count')).collect() #group rows
     based on the age group assigned and count them
end = time.time()
dataframe_time = end-start
print("DATAFRAME performance:", end-start)
print("DATAFRAME results:", categories_df)
print("Dataframe Speedup:", rdd_time/dataframe_time)
# DATAFRAME API END #
```

Για το RDD API η διαδικασία είναι αρκετά παρόμοια. Δημιουργήσαμε μια συνάρτηση `get_category_by_age`, η οποία επιστρέφει την ηλικιακή ομάδα βάσει της ηλικίας. Αφού κρατήσαμε μόνο τα δεδομένα που χρειαζόμαστε, περάσαμε όλα τα δεδομένα από την συνάρτηση μας έτσι ώστε να κατηγοριοποιηθούν σωστά. Προκειμένου, όμως, να γίνει μετά η καταμέτρηση, φροντίσαμε τα δεδομένα να έχουν την μορφή `(<Age Group>, 1)`, όπου `<Age Group>` η ηλικιακή ομάδα του θύματος βάσει της ηλικίας του. Έτσι, στο `reduction` βήμα, προσθέτουμε τα values (τους άσσους στην προκειμένη περίπτωση), παίρνοντας σαν αποτέλεσμα το πλήθος των θυμάτων ανά ηλικιακή ομάδα.

```
def get_category_by_age(age):
    if age is None:
        return "NULL"
    age = int(age)
    category = None
    if age <= 0:
        category = 'Unknown'
    elif age < 18:
        category = "Children"
```

```
    elif age <= 24:
        category = "Young adults"
    elif age <= 64:
        category = "Adults"
    else:
        category = "Elderly"
    return category

# RDD API BEGIN #
start = time.time()
crime_rdd = crime_data.rdd # convert dataframe to rdd
filtered_rdd = crime_rdd.filter(lambda row: "AGGRAVATED ASSAULT" in row["Crm Cd Desc"])
grouped_rdd = filtered_rdd.map(lambda row: (get_category_by_age(row["Vict Age"]), 1))
categories_rdd = grouped_rdd.reduceByKey(lambda x,y: x+y).sortBy(lambda tup: -tup[1]).collect()
end = time.time()
rdd_time = end - start
print("RDD performance:", end - start)
print("RDD results:", categories_rdd)
# RDD API END #
```

Σημείωση: Η συνάρτηση `get_category_by_age` μπορούσε να χρησιμοποιηθεί και στο DataFrame API, ως `User Defined Function`. Εν τέλει, ο λόγος που δεν χρησιμοποιήθηκε, είναι επειδή πιθανότατα θα εμποδίσει το PySpark από το να εκτελέσει optimizations (περισσότερα μπορείτε να δείτε εδώ).

Παρατηρήσαμε πως το dataset περιέχει δεδομένα με θύματα που έχουν ηλικία ≤ 0. Επιλέξαμε να δημιουργήσουμε μια επιπλέον κατηγορία `Unknown` για αυτά τα θύματα. Τα αποτελέσματα και των δύο APIs είναι προφανώς ίδια. Παρακάτω φαίνεται η έξοδος από το DataFrame API.

```
+------------+------+
|   AGE GROUP| count|
+------------+------+
|      Adults|121093|
|Young adults| 33605|
|    Children| 10830|
|     Elderly|  5985|
|     Unknown|  5098|
+------------+------+
```

Το DataFrame API τελείωσε την εκτέλεση του σε `7.735968351364136 sec` , ενώ το RDD API τελείωσε σε `16.28496217727661 sec`. Παρατηρούμε δηλαδή ένα speedup ίσο με **2.1050967943017724** στην εκτέλεση του DataFrame API σε σχέση με το RDD API. Αυτή η διαφορά οφείλεται στο `Catalyst`, ένα optimization layer του Spark το οποίο χρησιμοποιείται στα DataFrames αλλά όχι στα RDDs. Έτσι, το Spark εκτελεί αρκετες βελτιστοποιήσεις κατα την εκτέλεση του query με το DataFrame API, με αποτέλεσμα την καλύτερη απόδοση του, σε σχέση με την εκτέλεση του query χρησιμοποιώντας το RDD API.

2 Ζητούμενο 2

2.1 Υποερώτημα α

Στο ερώτημα αυτό, καλούμαστε να βρούμε για κάθε έτος, τα τρία Αστυνομικά Τμήματα με το υψηλότερο ποσοστό κλεισμένων υποθέσεων και να παρουσιάσουμε τα αποτελέσματα με φθίνουσα σειρά ως προς το `closed_case_rate` και αύξουσα ως προς το έτος. Το query εκτελέστηκε τόσο με το DataFrame API, όσο και με το SQL API.

Τα δεδομένα αντλήθηκαν από τα csv με τα καταγεγραμμένα εγκλήματα, όπως προηγουμένως και χρησιμοποιήθηκε ξανά η πράξη `union`, έτσι ώστε να αντιμετωπίζεται ενιαία το dataset.

```
### QUERY 2 (a)
APP_NAME = "Closed Cases"
spark = SparkSession.builder.appName(APP_NAME).getOrCreate() # reconstructing because we dont need 4
     spark executors now

# crime data
d1 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2010_to_2019_20241101.csv",
    header=True, inferSchema=True).select(col('DATE OCC'), col('AREA NAME'), col('Status'), col('LAT
    '), col('LON'))
d2 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2020_to_Present_20241101.csv",
    header=True, inferSchema=True).select(col('DATE OCC'), col('AREA NAME'), col('Status'), col('LAT
    '), col('LON'))
crime_data = d1.union(d2)
```

Παρακάτω φαίνεται η υλοποίηση του query χρησιμοποιώντας το DataFrame API. Γίνεται αρχικά εξαγωγή του έτους που καταγράφηκε το κάθε έγκλημα (σε νέα στήλη με το όνομα `YEAR`). Έπειτα, δημιουργείται μια νέα στήλη με το όνομα `closed` που συμβολίζει εάν η υπόθεση θεωρείται κλεισμένη ή όχι (χρησιμοποιώντας 1 ή 0 αντίστοιχα). Θεωρούμε μια υπόθεση κλεισμένη όταν η στήλη `Status` δεν έχει τις τιμές `IC` ή `UKN`. Στο επόμενο βήμα, κάνουμε aggregate τα rows σε ζευγάρια `(YEAR, AREA NAME)` και αθροίζουμε για το κάθε ζευγάρι την στήλη `closed` που δημιουργήσαμε προηγουμένως αλλά και το συνολικό πλήθος των rows. Έτσι, για κάθε Αστυνομικό Τμήμα και κάθε χρονιά έχουμε το πλήθος των κλεισμένων υποθέσεων αλλά και το πλήθος των υποθέσεων που ανέλαβε συνολικά. Υπολογίζουμε επομένως σε μια νέα στήλη `closed_case_rate`, το ποσοστό κλεισμένων υποθέσεων για κάθε ζευγάρι `(YEAR, AREA NAME)`. Ορίζουμε, έπειτα, ένα `Window function` το οποίο κάνει partition το DataFrame μας βάσει του `YEAR`. Τα partitioned δεδομένα ταξινομούνται βάσει του ποσοστού κλεισμένων υποθέσεων και κρατούνται μόνο τα 3 υψηλότερα ποσοστά ανά χρονιά. Έτσι, για κάθε έτος επιστρέφουμε τα 3 υψηλότερα ποσοστά κλεισμένων υποθέσεων μαζί με τα αντίστοιχα Αστυνομικά Τμήματα.

```
# DATAFRAME API BEGIN #
start = time.time()
crime_data = crime_data.withColumn('YEAR', year(to_timestamp(col('DATE OCC'), 'MM/dd/yyyy hh:mm:ss a
    '))).withColumn('closed', when(~col('Status').isin('IC', 'UKN'), 1).otherwise(0))
crime_data_agg = crime_data.groupBy('YEAR', 'AREA NAME').agg(
    count('*').alias("total_cases"),
    sum('closed').alias('closed_cases')
)
crime_cc_rates = crime_data_agg.withColumn('closed_case_rate', when(col('total_cases') > 0, col('
    closed_cases')/col('total_cases') * 100).otherwise(None)).drop('closed_cases').drop('total_cases
    ')
window_spec = Window.partitionBy('year').orderBy(desc('closed_case_rate'))
ranked_df = crime_cc_rates.withColumn("#", row_number().over(window_spec))
top3_df = ranked_df.filter(col("#") <= 3)
top3_df_collect = top3_df.collect()
end = time.time()
print("DATAFRAME performance:", end-start)
print("DATAFRAME results:")
top3_df.show(n=top3_df.count(), truncate=False)
# DATAFRAME API END #
```

Για την εκτέλεση του query με SQL API, η διαδικασία είναι παρόμοια. Πρώτα γράφουμε το query σε SQL, όπου για την παραγωγή των ενδιάμεσων αποτελεσμάτων χρησιμοποιήθηκαν τα ανάλογα Common Table Expressions.

```
WITH processed_data AS (
    SELECT
        YEAR(TO_TIMESTAMP(`DATE OCC`, 'MM/dd/yyyy hh:mm:ss a')) AS YEAR,
        `AREA NAME`,
        CASE WHEN `Status` NOT IN ('IC', 'UKN') THEN 1 ELSE 0 END AS closed
    FROM crime_data
),
```

```
aggregated_data AS (
    SELECT
        YEAR,
        `AREA NAME`,
        COUNT(*) AS total_cases,
        SUM(closed) AS closed_cases
    FROM processed_data
    GROUP BY YEAR, `AREA NAME`
),
crime_cc_rates AS (
    SELECT
        YEAR,
        `AREA NAME`,
        (CASE WHEN total_cases > 0 THEN (closed_cases / total_cases) * 100 ELSE NULL END) AS
    closed_case_rate
    FROM aggregated_data
),
ranked_data AS (
    SELECT
        YEAR,
        `AREA NAME`,
        closed_case_rate,
        ROW_NUMBER() OVER (PARTITION BY YEAR ORDER BY closed_case_rate DESC) AS rank
    FROM crime_cc_rates
)
SELECT
    YEAR,
    `AREA NAME`,
    closed_case_rate,
    rank AS `#`
FROM ranked_data
WHERE rank <= 3
```

Έπειτα εκτελούμε το SQL query μέσω του PySpark ως εξής:

```
# SQL API BEGIN #
start = time.time()
crime_data.createOrReplaceTempView("crime_data") # register the crime_data DataFrame as a temporary
    view
top3_sql = spark.sql(query)
top3_sql_collect = top3_sql.collect()
end = time.time()
print("SQL performance:", end-start)
print("SQL results:")
top3_sql.show(n=top3_df.count(), truncate=False)
# SQL API END #
```

Τα αποτελέσματα και των δύο εκτελέσεων είναι προφανώς τα ίδια. Ενδεικτικά, παρακάτω φαίνεται η έξοδος από το DataFrame API.

```
+----+-----------+------------------+---+
|YEAR|AREA NAME  |closed_case_rate  |#  |
+----+-----------+------------------+---+
|2010|Rampart    |32.84713448949121 |1  |
|2010|Olympic    |31.515289821999087|2  |
|2010|Harbor     |29.36028339237341 |3  |
|2011|Olympic    |35.0400600901352061|1  |
|2011|Rampart    |32.4964471814306  |2  |
|2011|Harbor     |28.51336246316431 |3  |
|2012|Olympic    |34.29708533302119 |1  |
|2012|Rampart    |32.46000463714352 |2  |
|2012|Harbor     |29.509585848956675|3  |
|2013|Olympic    |33.58217940999398 |1  |
|2013|Rampart    |32.1060382916053  |2  |
|2013|Harbor     |29.723638951488557|3  |
|2014|Van Nuys   |32.0215235281705  |1  |
|2014|West Valley|31.49754809505847 |2  |
|2014|Mission    |31.2249398556535673|3  |
|2015|Van Nuys   |32.265140677157845|1  |
|2015|Mission    |30.4637626736763031|2  |
|2015|Foothill   |30.3530018036588523|3  |
|2016|Van Nuys   |32.1945184621240941|1  |
|2016|West Valley|31.40146437042384 |2  |
|2016|Foothill   |29.9086472281316453|3  |
|2017|Van Nuys   |32.0554272517321  |1  |
|2017|Mission    |31.0553871589969683|2  |
```

```
|2017|Foothill   |30.469700657094183|3  |
|2018|Foothill   |30.731346958877126|1  |
|2018|Mission    |30.727023319615913|2  |
|2018|Van Nuys   |28.905206942590123|3  |
|2019|Mission    |30.727411112319235|1  |
|2019|West Valley|30.57974335472044 |2  |
|2019|N Hollywood|29.23808669119627 |3  |
|2020|West Valley|30.771131982204647|1  |
|2020|Mission    |30.14974649215894 |2  |
|2020|Harbor     |29.693486590038315|3  |
|2021|Mission    |30.318115590092276|1  |
|2021|West Valley|28.971087440009363|2  |
|2021|Foothill   |27.993757094211126|3  |
|2022|West Valley|26.536367172306498|1  |
|2022|Harbor     |26.337538060026098|2  |
|2022|Topanga    |26.2340133178310963|3  |
|2023|Foothill   |26.76076020122974 |1  |
|2023|Topanga    |26.538022616453986|2  |
|2023|Mission    |25.662731120516817|3  |
|2024|N Hollywood|19.598528961078763|1  |
|2024|Foothill   |18.620882188721385|2  |
|2024|77th Street|17.586318167150694|3  |
+----+-----------+------------------+---+
```

Η έκδοση του query με το DataFrame API, εκτελείται σε `13.183066368103027 sec`, ενώ η έκδοση με το SQL API, σε `8.888805389404297 sec`. Παρατηρούμε, δηλαδή, πως η εκτέλεση της έκδοσης με το SQL API παρουσιάζει speedup **1.4831088982798082** σε σχέση με το DataFrame API. Ωστόσο χρονικά οι δύο εκδόσεις δεν απέχουν ιδιαίτερα, η διαφορά τους ίσως οφείλεται στο γεγονός ότι ενώ και τα δύο APIs βασίζονται στο lazy evaluation, το SQL API έχει καλύτερη διαχείριση καθώς εισάγει ολόκληρο το σχέδιο στο Catalyst Optimizer με μία εντολή. Στο DataFrame API, κάθε μετασχηματισμός εισάγεται σταδιακά, κάτι που ενδεχομένως δυσχεραίνει την εφαρμογή βελτιστοποιήσεων.

2.2 Υποερώτημα β

Στο ερώτημα αυτό, αποθηκεύσαμε το αρχικό dataset ως αρχείο τύπου parquet. Το Apache Parquet είναι ένα format αρχείου, το οποίο έχει σχεδιαστεί για να υποστηρίζει ταχύτερη επεξεργασία δεδομένων. Έπειτα, το χρησιμοποιήσαμε για να εκτελέσουμε ξανά το query 2 με το DataFrame API.

```
### QUERY 2 (b)
crime_data.write.mode("overwrite").parquet("s3://groups-bucket-dblab-905418150721/group33/CrimeData.
    parquet")
```

Παρακάτω, φαίνεται ο τρόπος με τον οποίο διαβάσαμε τα δεδομένα από το αρχείο parquet και εκτελέσαμε ξανά το προηγούμενο query.

```
crime_data_parquet = spark.read.parquet("s3://groups-bucket-dblab-905418150721/group33/CrimeData.
    parquet")

def process_data_dataframe(crime_data):
    crime_data = crime_data.withColumn('YEAR', year(to_timestamp(col('DATE OCC'), 'MM/dd/yyyy hh:mm:
    ss a'))) \
                           .withColumn('closed', when(~col('Status').isin('IC', 'UKN'), 1).otherwise
    (0))
    crime_data_agg = crime_data.groupBy('YEAR', 'AREA NAME').agg(
        count('*').alias("total_cases"),
        sum('closed').alias('closed_cases')
    )
    crime_cc_rates = crime_data_agg.withColumn(
        'closed_case_rate',
        when(col('total_cases') > 0, col('closed_cases') / col('total_cases') * 100).otherwise(None)
    ).drop('closed_cases').drop('total_cases')
    window_spec = Window.partitionBy('YEAR').orderBy(desc('closed_case_rate'))
    ranked_df = crime_cc_rates.withColumn("#", row_number().over(window_spec))
    top3_df = ranked_df.filter(col("#") <= 3)
    return top3_df

# DATAFRAME API BEGIN #
start = time.time()
top3_df_parquet = process_data_dataframe(crime_data_parquet)
top3_df_parquet_collect = top3_df_parquet.collect()
end = time.time()
print("DATAFRAME performance:", end-start)
print("DATAFRAME results:")
```

```
top3_df_parquet.show(n=top3_df_parquet.count(), truncate=False)
# DATAFRAME API END #
```

Η εκτέλεση του query 2 χρησιμοποιώντας τα δεδομένα από το parquet αρχείο και το DataFrame API, εκτελείται σε μόλις 4.5808703899383545 sec. Παρουσιάζει, δηλαδή, speedup **2.8778518591268054** σε σύγκριση με την εκτέλεση του query 2, χρησιμοποιώντας τα δεδομένα από τα csv αρχεία και το DataFrame API. Γεγονός που ήταν αναμενόμενο αφού η χρήση δεδομένων από parquet αρχεία είναι πολύ πιο αποδοτική λόγω της στήλη-προσανατολισμένης φύσης, της ενσωματωμένης υποστήριξης σχήματος, των βελτιστοποιήσεων φίλτρου και της αποφυγής parsing. Αντίθετα, τα csv αρχεία απαιτούν περισσότερη επεξεργασία για parsing, ανάγνωση και μετατροπή.

3 Ζητούμενο 3

Στο συγκεκριμένο ερώτημα θα χρησιμοποιήσουμε ως αναφορά τα δεδομένα απογραφής 2010 για τον πληθυσμό και εκείνα της απογραφής 2015 για το εισόδημα ανά νοικοκυριό προκειμένου να να υπολογίσουμε για κάθε περιοχή του Los Angeles το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία του συνολικού αριθμού εγκλημάτων ανά άτομο. Το query εκτελέστηκε με DataFrame API.

Πρώτα απ' όλα απαιτείται να αρχικοποιήσουμε το Sedona Context, το οποίο είναι απαραίτητο για την εκτέλεση geospatial analytics χρησιμοποιώντας τη βιβλιοθήκη Apache Sedona. Τα δεδομένα εγκλημάτων φορτώνονται από δύο αρχεία csv που περιέχουν εγγραφές από το 2010 έως το παρόν και ενοποιούνται με τη μέθοδο `union`. Για να αποφευχθούν σφάλματα, αφαιρούνται εγγραφές με συντεταγμένες στο `Null Island (0,0)`. Στη συνέχεια, προστίθεται μια νέα στήλη `geometry` που περιέχει τα γεωμετρικά σημεία των εγκλημάτων, χρησιμοποιώντας τις στήλες `LAT` και `LON`. Τα δεδομένα απογραφής πληθυσμού φορτώνονται από ένα αρχείο GeoJSON. Αφαιρούνται εγγραφές με αρνητικές τιμές πληθυσμού ή κατοικιών. Επίσης, φιλτράρονται μόνο οι εγγραφές που ανήκουν στην πόλη του Los Angeles, με βάση τη στήλη `CITY`. Τέλος τα δεδομένα εισοδήματος φορτώνονται από ένα αρχείο csv. Η στήλη που περιέχει το μέσο εισόδημα ανά νοικοκυριό καθαρίζεται αφαιρώντας τα σύμβολα $ και ,. Η καθαρισμένη τιμή μετατρέπεται σε αριθμητικό τύπο δεδομένων (`DecimalType`).

```
### QUERY 3
APP_NAME = "(Income/person and crimes/person) / region"
spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

sedona = SedonaContext.create(spark)

# crime data
d1 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2010_to_2019_20241101.csv",
    header=True, inferSchema=True)
d2 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2020_to_Present_20241101.csv",
    header=True, inferSchema=True)
crime_data = d1.union(d2)

# remove NULL ISLAND (0,0)
crime_data = crime_data.filter(~((col('LAT') == 0) & (col('LON') == 0))) \
        .select('DR_NO', 'LAT', 'LON').withColumn('geometry', ST_Point('LON', 'LAT')).drop('LON').
    drop('LAT')

# census block data
geojson_path = "s3://initial-notebook-data-bucket-dblab-905418150721/2010_Census_Blocks.geojson"
census_blocks = sedona.read.format("geojson") \
            .option("multiLine", "true").load(geojson_path) \
            .selectExpr("explode(features) as features") \
            .select("features.*")
census_blocks = census_blocks.select(
            [col(f"properties.{col_name}").alias(col_name) for col_name in
            census_blocks.schema["properties"].dataType.fieldNames()] + ["geometry"],).drop("
    properties").drop("type")
census_blocks = census_blocks.filter((census_blocks['HOUSING10'] >= 0) & (census_blocks['POP_2010']
    >= 0)).filter(F.col('CITY') == 'Los Angeles')

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

income_data = income_data.withColumn(
    "median_hincome",
    regexp_replace(col("Estimated Median Income"), "[$,]", "").cast(DecimalType())
).drop('Estimated Median Income', 'Community')
```

Για την κατάλληλη επεξεργασία των δεδομένων χρησιμοποιείται η μέθοδος `ST_Within` της Sedona για να συσχετιστούν τα δεδομένα εγκλημάτων με τα δεδομένα απογραφής. Συγκεκριμένα, κάθε περιστατικό εγκλήματος αντιστοιχίζεται στην περιοχή απογραφής (`COMM`) στην οποία ανήκει. Στη συνέχεια, υπολογίζεται ο συνολικός αριθμός εγκλημάτων για κάθε περιοχή. Οι περιοχές ομαδοποιούνται με βάση τον ταχυδρομικό κώδικα (`ZCTA10`) και την περιοχή (`COMM`). Υπολογίζονται το συνολικό πλήθος κατοίκων και κατοικιών για κάθε περιοχή. Έπειτα, γίνεται σύνδεση με τα δεδομένα εισοδήματος, χρησιμοποιώντας τον ταχυδρομικό κώδικα ως κοινό πεδίο. Υπολογίζεται το συνολικό εισόδημα για κάθε περιοχή, πολλαπλασιάζοντας τον αριθμό κατοικιών με το μέσο εισόδημα ανά νοικοκυριό. Οι περιοχές ομαδοποιούνται με βάση τη στήλη `COMM`, και υπολογίζεται το συνολικό εισόδημα και ο πληθυσμός

για κάθε περιοχή. Στη συνέχεια, συνδέονται τα δεδομένα εγκλημάτων με αυτά τα σύνολα, δημιουργώντας έναν ενιαίο πίνακα που περιλαμβάνει τα δεδομένα εγκλημάτων, πληθυσμού και εισοδήματος για κάθε περιοχή (έγινε χρήση `RIGHT JOIN`, έτσι ώστε να συμπεριλάβουμε περιοχές χωρίς κανένα έγκλημα). Τέλος για κάθε περιοχή, υπολογίζεται το μέσο εισόδημα ανά άτομο και η αναλογία εγκλημάτων ανά άτομο, σε περιπτώσεις όπου ο πληθυσμός είναι μηδενικός, ορίζεται η τιμή σε μηδέν.

```
# DATAFRAME API BEGIN #
start = time.time()
# crime data
comm_join_crimes = crime_data.join(census_blocks, ST_Within(crime_data['geometry'], census_blocks['
    geometry']))
count_comm_crimes = comm_join_crimes.groupby('COMM').agg(F.count('*').alias('crimes_count'))
#income data
zip_comm_houses_pop = census_blocks.groupBy('ZCTA10', 'COMM').agg(
    sum(col('POP_2010')).alias('population'),
    sum(col('HOUSING10')).alias('houses'),
).select('ZCTA10', 'COMM', 'population', 'houses')
# joined with median household income
zip_comm_houses_pop_hincome = zip_comm_houses_pop.join(
    income_data,
    income_data['Zip Code'] == zip_comm_houses_pop['ZCTA10']
).drop('Zip Code')
# zip_total_income = houses * median_hincome per (zip, comm)
zip_comm_total_income = zip_comm_houses_pop_hincome.withColumn(
    "zip_total_income",
    col('median_hincome') * col('houses')
)
# total_population per COMM
comm_total_population_total_income = zip_comm_total_income.groupBy('COMM').agg(
    sum('population').alias('total_population'),
    sum('zip_total_income').alias('total_income')
)
# join crime and income data
comm_crime_income = comm_total_population_total_income.join(
    count_comm_crimes,
    on='COMM',
    how='right'
)
comm_mincome_person = comm_crime_income.withColumn(
    'average_income_per_person',
    when(col('total_population') > 0, col('total_income') / col('total_population'))
    .otherwise(0)).withColumn(
    'crime_rate_per_person',
    when(col('total_population') > 0, col('crimes_count') / col('total_population'))
    .otherwise(0)
).drop('total_income').drop('crimes_count').drop('total_population')
comm_mincome_person_collect = comm_mincome_person.collect()
end = time.time()
print("DATAFRAME performance:", end-start)
comm_mincome_person.show(n=comm_mincome_person.count(), truncate=False)
# DATAFRAME API END #
```

Παρακάτω φαίνεται η έξοδος του ερωτήματος (οι πρώτες 20 σειρές) το οποίο εκτελείται σε `29.66372036933899 sec`.

```
+----------------------+-------------------------+---------------------+
|COMM                  |average_income_per_person|crime_rate_per_person|
+----------------------+-------------------------+---------------------+
|San Pedro             |24244.561240             |0.7751021460858706   |
|South Park            |6943.255564              |0.8862840357021805   |
|Vernon Central        |6624.199012              |0.7288347583605012   |
|Florence-Firestone    |8079.274829              |1.1133874146386178   |
|Studio City           |44049.852630             |0.9295754238516157   |
|Valley Village        |28191.368349             |0.6809068681929318   |
|Sherman Oaks          |37767.445674             |0.7903687241383545   |
|Valley Glen           |18271.771022             |0.5546786523216308   |
|Park La Brea          |36619.900017             |1.025207944321847    |
|Miracle Mile          |38834.846111             |0.8820452139253908   |
|Hollywood             |25648.050759             |1.5511440107671601   |
|Hancock Park          |21538.793405             |0.8983094426946069   |
|Melrose               |21688.494261             |0.909787014117455    |
|North Hills           |16224.486386             |0.655059866962306    |
|Panorama City         |10221.126106             |0.6498237536867851   |
|Mission Hills         |17158.531061             |0.6565328178543186   |
|Arleta                |12110.779170             |0.4423287504562599   |
+----------------------+-------------------------+---------------------+
```

Για να δοκιμάσουμε όλους τους δυνατούς συνδυασμούς στις στρατηγικές join μεταξύ των BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL για τα τρία join του query αλλάζουμε τον κώδικα ως εξής:

```
### QUERY 3 (join strategies)
def execute_query_with_join_strategy(strategy1, strategy2, strategy3):
    start = time.time()

    # Ρυθμίσεις για το Spark SQL για κάθε join
    spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)  # Απενεργοποίηση default broadcast
    spark.conf.set("spark.sql.join.preferSortMergeJoin", "true" if strategy1 == "MERGE" else "false"
    )
    spark.conf.set("spark.sql.join.preferBroadcastJoin", "true" if strategy1 == "BROADCAST" else "
    false")

    # crime_data join census_blocks (strategy1)
    comm_join_crimes = crime_data.join(
        census_blocks.hint(strategy1),
        ST_Within(crime_data['geometry'], census_blocks['geometry'])
    )
    count_comm_crimes = comm_join_crimes.groupby('COMM').agg(F.count('*').alias('crimes_count'))

    # zip_comm_houses_pop_hincome join income_data (strategy2)
    zip_comm_houses_pop = census_blocks.groupBy('ZCTA10', 'COMM').agg(
        sum(col('POP_2010')).alias('population'),
        sum(col('HOUSING10')).alias('houses'),
    ).select('ZCTA10', 'COMM', 'population', 'houses')

    zip_comm_houses_pop_hincome = zip_comm_houses_pop.join(
        income_data.hint(strategy2),
        income_data['Zip Code'] == zip_comm_houses_pop['ZCTA10']
    ).drop('Zip Code')

    # zip_comm_total_income (strategy3)
    zip_comm_total_income = zip_comm_houses_pop_hincome.withColumn(
        "zip_total_income",
        col('median_hincome') * col('houses')
    )

    comm_total_population_total_income = zip_comm_total_income.groupBy('COMM').agg(
        sum('population').alias('total_population'),
        sum('zip_total_income').alias('total_income')
    )

    comm_crime_income = comm_total_population_total_income.join(
        count_comm_crimes.hint(strategy3),
        on='COMM',
        how='outer'
    )

    comm_mincome_person = comm_crime_income.withColumn(
        'average_income_per_person',
        when(col('total_population') > 0, col('total_income') / col('total_population'))
        .otherwise(0)
    ).withColumn(
        'crime_rate_per_person',
        when(col('total_population') > 0, col('crimes_count') / col('total_population'))
        .otherwise(0)
    ).drop('total_income').drop('crimes_count').drop('total_population')
    comm_mincome_person_collect = comm_mincome_person.collect()

    end = time.time()
    execution_time = end - start

    return execution_time

join_strategies = ["BROADCAST", "MERGE", "SHUFFLE_HASH", "SHUFFLE_REPLICATE_NL"]
strategy_combinations = list(itertools.product(join_strategies, repeat=3))

results = []

for strat1, strat2, strat3 in strategy_combinations:
    exec_time = execute_query_with_join_strategy(strat1, strat2, strat3)
    results.append((strat1, strat2, strat3, exec_time))

results_df = spark.createDataFrame(results, ["Strategy1", "Strategy2", "Strategy3", "ExecutionTime"
    ])
results_df_sorted = results_df.orderBy("ExecutionTime", ascending=True)
```

```
results_df_sorted.show(n=results_df_sorted.count(), truncate=False)
```

Παρακάτω φαίνεται ο χρόνος εκτέλεσης για κάθε συνδυασμό join strategies.

```
+--------------------+--------------------+--------------------+------------------+
|Strategy1           |Strategy2           |Strategy3           |ExecutionTime     |
+--------------------+--------------------+--------------------+------------------+
|SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |BROADCAST           |13.472944498062134|
|BROADCAST           |SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|13.713311672210693|
|SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|MERGE               |13.72116470336914 |
|SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|13.74719762802124 |
|SHUFFLE_HASH        |SHUFFLE_HASH        |BROADCAST           |13.748756647109985|
|SHUFFLE_REPLICATE_NL|MERGE               |BROADCAST           |14.009728908538818|
|SHUFFLE_REPLICATE_NL|BROADCAST           |BROADCAST           |14.104211568832397|
|SHUFFLE_HASH        |SHUFFLE_HASH        |SHUFFLE_HASH        |14.124082088470459|
|SHUFFLE_HASH        |SHUFFLE_HASH        |MERGE               |14.143094301223755|
|SHUFFLE_HASH        |MERGE               |MERGE               |14.165607452392578|
|BROADCAST           |SHUFFLE_HASH        |MERGE               |14.181203842163086|
|SHUFFLE_REPLICATE_NL|BROADCAST           |SHUFFLE_HASH        |14.28413438796997 |
|SHUFFLE_REPLICATE_NL|MERGE               |SHUFFLE_HASH        |14.296715497970581|
|SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |14.303778886795044|
|MERGE               |MERGE               |BROADCAST           |14.306119918823242|
|BROADCAST           |SHUFFLE_REPLICATE_NL|MERGE               |14.30896806716919 |
|SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|14.312310218811035|
|SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|BROADCAST           |14.338404417037964|
|SHUFFLE_HASH        |BROADCAST           |SHUFFLE_REPLICATE_NL|14.344301700592041|
|SHUFFLE_REPLICATE_NL|MERGE               |MERGE               |14.345511674880981|
|MERGE               |SHUFFLE_HASH        |BROADCAST           |14.434895992279053|
|SHUFFLE_REPLICATE_NL|MERGE               |SHUFFLE_REPLICATE_NL|14.442206621170044|
|SHUFFLE_REPLICATE_NL|BROADCAST           |SHUFFLE_REPLICATE_NL|14.490171670913696|
|MERGE               |SHUFFLE_HASH        |SHUFFLE_HASH        |14.524138689041138|
|MERGE               |SHUFFLE_REPLICATE_NL|MERGE               |14.545830965042114|
|SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|MERGE               |14.566548347473145|
|MERGE               |SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |14.573344707489014|
|MERGE               |SHUFFLE_HASH        |MERGE               |14.653209447860718|
|BROADCAST           |SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|14.756784200668335|
|SHUFFLE_HASH        |SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|14.820102214813232|
|MERGE               |SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|14.938286542892456|
|BROADCAST           |SHUFFLE_HASH        |BROADCAST           |14.974244832992554|
|SHUFFLE_HASH        |BROADCAST           |BROADCAST           |14.982024669647217|
|SHUFFLE_HASH        |MERGE               |BROADCAST           |15.060087203979492|
|SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |SHUFFLE_HASH        |15.076187133789062|
|SHUFFLE_HASH        |MERGE               |SHUFFLE_HASH        |15.079508543014526|
|SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|BROADCAST           |15.351330280303955|
|SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |SHUFFLE_REPLICATE_NL|15.469627141952515|
|SHUFFLE_HASH        |BROADCAST           |SHUFFLE_HASH        |15.482222080230713|
|BROADCAST           |BROADCAST           |SHUFFLE_HASH        |15.497687578201294|
|MERGE               |MERGE               |MERGE               |15.682572364807129|
|BROADCAST           |BROADCAST           |MERGE               |15.800673723220825|
|SHUFFLE_HASH        |MERGE               |SHUFFLE_REPLICATE_NL|15.830183982849121|
|MERGE               |MERGE               |SHUFFLE_HASH        |15.844552516937256|
|MERGE               |BROADCAST           |BROADCAST           |15.85817003250122 |
|MERGE               |BROADCAST           |SHUFFLE_HASH        |15.874963998794556|
|MERGE               |SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|15.993769884109497|
|SHUFFLE_REPLICATE_NL|SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |16.042428493499756|
|BROADCAST           |BROADCAST           |SHUFFLE_REPLICATE_NL|16.231744050979614|
|BROADCAST           |MERGE               |MERGE               |16.40001654624939 |
|BROADCAST           |MERGE               |SHUFFLE_REPLICATE_NL|16.669507026672363|
|SHUFFLE_REPLICATE_NL|BROADCAST           |MERGE               |17.152887105941772|
|SHUFFLE_HASH        |BROADCAST           |MERGE               |17.63453984260559 |
|BROADCAST           |BROADCAST           |BROADCAST           |17.746803045272827|
|MERGE               |BROADCAST           |MERGE               |17.746988534927368|
|BROADCAST           |MERGE               |BROADCAST           |18.210407495498657|
|BROADCAST           |SHUFFLE_HASH        |SHUFFLE_HASH        |19.552934885025024|
|MERGE               |BROADCAST           |SHUFFLE_REPLICATE_NL|19.877957344055176|
|MERGE               |SHUFFLE_REPLICATE_NL|BROADCAST           |20.444053649902344|
|MERGE               |MERGE               |SHUFFLE_REPLICATE_NL|20.667855501174927|
|BROADCAST           |SHUFFLE_REPLICATE_NL|BROADCAST           |20.7849862575531  |
|BROADCAST           |MERGE               |SHUFFLE_HASH        |21.747425079345703|
|BROADCAST           |SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |23.430171728134155|
|SHUFFLE_REPLICATE_NL|SHUFFLE_HASH        |MERGE               |24.037864923477173|
+--------------------+--------------------+--------------------+------------------+
```

Γενικότερα οι οι στρατηγικές join καθορίζουν πώς δύο σύνολα δεδομένων συνδυάζονται. Οι κύριες στρατηγικές είναι:

- **Broadcast Join:** Ένα μικρότερο dataset αντιγράφεται και διαμοιράζεται σε όλους τους κόμβους. Είναι ιδανικό για μικρά datasets, καθώς μειώνει την ανάγκη για data shuffling.
- **Merge Join (Sort-Merge Join):** Τα datasets ταξινομούνται πρώτα και στη συνέχεια συνδυάζονται. Αυτή η στρατηγική είναι αποτελεσματική για μεγάλα datasets που είναι ήδη ταξινομημένα.
- **Shuffle Hash Join:** Τα δεδομένα χωρίζονται σε partitions βάσει του hash key και στη συνέχεια γίνεται το join. Είναι κατάλληλο για μεγάλου μεγέθους datasets αλλά απαιτεί shuffling.
- **Shuffle Replicate NL (Nested Loop Join):** Το Spark χρησιμοποιεί shuffling στους πίνακες, ενώ μετά αντιγράφεται ένας από τους δύο προκύπτοντες πίνακες και το αποτέλεσμα είναι ένα καρτεσιανό γινόμενο των δύο datasets.

Από τα αποτελέσματα που παρουσιάζονται στον παραπάνω πίνακα έχουμε τις εξής παρατηρήσεις:

- Ο συνδυασμός `SHUFFLE_REPLICATE_NL`, `SHUFFLE_HASH`, και `BROADCAST` είναι ο γρηγορότερος με χρόνο εκτέλεσης 13.47 δευτερόλεπτα. Αυτό οφείλεται στη χρήση του BROADCAST που ελαχιστοποιεί το shuffling για το τρίτο join.
- Ο συνδυασμός `BROADCAST`, `SHUFFLE_HASH`, και `SHUFFLE_REPLICATE_NL` έχει τον μεγαλύτερο χρόνο εκτέλεσης (23.43 δευτερόλεπτα), λόγω της χρήσης του S `SHUFFLE_REPLICATE_NL`, που είναι αργό για μεγάλα δεδομένα.
- Η χρήση του `BROADCAST` γενικά μειώνει το χρόνο εκτέλεσης όταν χρησιμοποιείται σε κρίσιμα σημεία.
- Οι στρατηγικές `MERGE` και `SHUFFLE_HASH` δείχνουν καλή ισορροπία απόδοσης.
- Ο `SHUFFLE_REPLICATE_NL` αυξάνει σημαντικά το χρόνο εκτέλεσης όταν χρησιμοποιείται επαναλαμβανόμενα.
- Συνδυασμοί που περιλαμβάνουν το `BROADCAST` στην τελευταία ή πρώτη φάση τείνουν να είναι γρηγορότεροι.
- Όταν το `MERGE` χρησιμοποιείται σε τουλάχιστον μία φάση, μειώνεται το κόστος ταξινόμησης και shuffling.
- Η επιλογή των στρατηγικών join πρέπει να γίνεται βάσει του μεγέθους των δεδομένων και της σχέσης μεταξύ τους.
- Το `BROADCAST` είναι ιδανικό για μικρά datasets, ενώ το `MERGE` ή το `SHUFFLE_HASH` για μεγάλα.

4 Ζητούμενο 4

Σε αυτό το ερώτημα καλούμαστε να βρούμε το φυλετικό προφίλ των καταγεγραμμένων θυμάτων εγκλημάτων στο Los Angeles για το έτος 2015 τόσο στις τρεις περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα όσο και στις τρεις περιοχές με το χαμηλότερο κατά κεφαλήν εισόδημα. Τα αποτελέσματα ζητούνται σε φθίνουσα σειρά πλήθους θυμάτων ανά φυλετική κατηγορία. Το query εκτελέστηκε με DataFrame API.

Αρχικά όπως και στο προηγούμενο ερώτημα γίνεται φόρτωση των δεδομένων εγκλημάτων από ένα αρχείο csv (χρησιμοποιούμε μόνο αυτό που περιέχει δεδομένα για τα έτη 2010-2019 αφού τα αποτελέσματα μας αφορούν το έτος 2015). Τα δεδομένα φιλτράρονται για να απομακρυνθούν οι εγγραφές με συντεταγμένες (0,0). Επιπλέον, για κάθε εγγραφή δημιουργείται μια γεωμετρική αναπαράσταση (geometry) των συντεταγμένων με τη βοήθεια της βιβλιοθήκης Apache Sedona. Έπειτα τα δεδομένα φιλτράρονται ώστε να διατηρηθούν μόνο τα εγκλήματα που συνέβησαν το 2015, και έχουν συμπληρωμένη τη στήλη `Vict Descent` που περιγράφει τη φυλετική καταγωγή του θύματος. Στην συνέχεια ακολουθούμε ακριβώς την ίδια διαδικασία με το ερώτημα τρία για να εξάγουμε το μέσο εισόδημα ανά άτομο σε κάθε περιοχή.

```
### Query 4
APP_NAME = "Crime Victime Race Analysis"
SPARK_EXECUTORS = 2
spark = SparkSession.builder.appName(APP_NAME).config("spark.executor.instances", SPARK_EXECUTORS).
    getOrCreate()
sedona = SedonaContext.create(spark)

# crime data
crime_data = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2010_to_2019_20241101.csv",
    header=True, inferSchema=True
)

# remove NULL ISLAND (0,0)
crime_data = crime_data.filter(~((F.col('LAT') == 0) & (F.col('LON') == 0))) \
    .select('DR_NO', 'LAT', 'LON', 'Vict Descent', 'Date Rptd') \
    .withColumn('geometry', ST_Point('LON', 'LAT')) \
    .drop('LON').drop('LAT')

# filter crimes for the year 2015
crime_data = crime_data.withColumn("Date Rptd", to_timestamp(F.col("Date Rptd"), "MM/dd/yyyy hh:mm:
    ss a"))
crimes_2015 = crime_data.filter(
    (F.year(F.col("Date Rptd")) == 2015) & (F.col("Vict Descent").isNotNull())
)

# median household income data
income_data = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721//LA_income_2015.csv",
    header=True, inferSchema=True
)
income_data = income_data.withColumn(
    "median_income",
    F.regexp_replace(F.col("Estimated Median Income"), "[$,]", "").cast(DecimalType())
).drop('Estimated Median Income', 'Community')

# census block data
geojson_path = "s3://initial-notebook-data-bucket-dblab-905418150721/2010_Census_Blocks.geojson"
census_blocks = sedona.read.format("geojson") \
            .option("multiLine", "true").load(geojson_path) \
            .selectExpr("explode(features) as features") \
            .select("features.*")
census_blocks = census_blocks.select(
            [col(f"properties.{col_name}").alias(col_name) for col_name in
            census_blocks.schema["properties"].dataType.fieldNames()] + ["geometry"],).drop("
    properties").drop("type")
census_blocks = census_blocks.filter(F.col('CITY') == 'Los Angeles')

# race and ethnicity legend
descent_legend = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/RE_codes.csv",
    header=True, inferSchema=True
)

### Query 3
comm_join_crimes = crime_data.join(census_blocks, ST_Within(crime_data['geometry'], census_blocks['
    geometry']))
```

```
count_comm_crimes = comm_join_crimes.groupby('COMM').agg(F.count('*').alias('crimes_count'))

zip_comm_houses_pop = census_blocks.groupBy('ZCTA10', 'COMM').agg(
    sum(col('POP_2010')).alias('population'),
    sum(col('HOUSING10')).alias('houses'),
).select('ZCTA10', 'COMM', 'population', 'houses')

zip_comm_houses_pop_hincome = zip_comm_houses_pop.join(
    income_data,
    income_data['Zip Code'] == zip_comm_houses_pop['ZCTA10']
).drop('Zip Code')

zip_comm_total_income = zip_comm_houses_pop_hincome.withColumn(
    "zip_total_income",
    col('median_income') * col('houses')
)

comm_total_population_total_income = zip_comm_total_income.groupBy('COMM').agg(
    sum('population').alias('total_population'),
    sum('zip_total_income').alias('total_income')
)

comm_crime_income = comm_total_population_total_income.join(
    count_comm_crimes,
    on='COMM',
    how='right'
)
comm_mincome_person = comm_crime_income.withColumn(
    'average_income_per_person',
    when(col('total_population') > 0, col('total_income') / col('total_population'))
    .otherwise(0)).withColumn(
    'crime_rate_per_person',
    when(col('total_population') > 0, col('crimes_count') / col('total_population'))
    .otherwise(0)
).drop('total_income').drop('crimes_count').drop('total_population')
```

Με τη χρήση ενός παραθύρου (window function), υπολογίζεται ένας μοναδικός αριθμός σειράς για κάθε κοινότητα με βάση το μέσο εισόδημα ανά άτομο. Έτσι, εντοπίζονται οι τρεις περιοχές με το υψηλότερο και χαμηλότερο κατά κεφαλήν εισόδημα. Τα δεδομένα εγκλημάτων συνδέονται με τα **census blocks**, χρησιμοποιώντας τη γεωχωρική συνάρτηση **ST_Within** της Apache Sedona, ώστε να αποδοθεί κάθε έγκλημα στην αντίστοιχη περιοχή. Στη συνέχεια, πραγματοποιείται αντιστοίχιση με τον πίνακα **descent_legend**, ώστε να μετατραπούν οι κωδικοί φυλετικής καταγωγής σε πλήρη περιγραφή. Τα εγκλήματα για τις τρεις περιοχές με το υψηλότερο και χαμηλότερο εισόδημα ομαδοποιούνται ανά φυλετική καταγωγή (**descent_description**), υπολογίζεται ο αριθμός των θυμάτων για κάθε φυλετική ομάδα και τα αποτελέσματα εμφανίζονται ταξινομημένα σε φθίνουσα σειρά σε δύο πίνακες.

```
def query4(crimes_2015, income_data, census_blocks, descent_legend, comm_mincome_person):
    window_spec = Window.orderBy("average_income_per_person")
    comm_mincome_person = comm_mincome_person.withColumn("row_num", row_number().over(window_spec))
    # with row number
    comm_bottom_3 = comm_mincome_person.filter(comm_mincome_person.row_num <= 3).select('COMM')
    comm_top_3 = comm_mincome_person.filter(comm_mincome_person.row_num > comm_mincome_person.count
    () - 3).select("COMM")

    # match crimes to zip codes
    crimes_with_zipcodes = crimes_2015.join(
        census_blocks,
        ST_Within(crime_data['geometry'], census_blocks['geometry'])
    )

    # join with descent legend for descriptions
    crimes_with_descriptions = crimes_with_zipcodes.join(
        descent_legend,
        crimes_with_zipcodes["Vict Descent"] == descent_legend["Vict Descent"],
        how="left"
    ).select(
        crimes_with_zipcodes["*"],
        descent_legend["Vict Descent Full"].alias("descent_description")
    )

    top_crimes = crimes_with_descriptions.join(comm_top_3, "COMM", "inner")
    bottom_crimes = crimes_with_descriptions.join(comm_bottom_3, "COMM", "inner")

    # group and count victims by descent description for each group
    top_victim_count = top_crimes.groupBy("descent_description") \
        .agg(F.count("*").alias("victim_count")) \
```

```
        .orderBy(F.desc("victim_count"))

    bottom_victim_count = bottom_crimes.groupBy("descent_description") \
        .agg(F.count("*").alias("victim_count")) \
        .orderBy(F.desc("victim_count"))

    # display results
    print("Top COMM Victim Count:")
    top_victim_count.show()

    print("Bottom COMM Victim Count:")
    bottom_victim_count.show()
```

Σημείωση: Θα μπορούσαμε να εκτελέσουμε την πράξη `UNION` μεταξύ των 3 περιοχών με χαμηλότερο εισόδημα και των 3 με μεγαλύτερο, ώστε να γίνει ένα `JOIN` με το dataset εγκλημάτων αντί για δύο. Θέλαμε, ωστόσο, να τα έχουμε σε διαφορετικά DataFrames.

Τα αποτελέσματα του ερωτήματος φαίνονται παρακάτω.

```
Top COMM Victim Count:
+--------------------+------------+
| descent_description|victim_count|
+--------------------+------------+
|               White|         682|
|               Other|          81|
|Hispanic/Latin/Me...|          75|
|             Unknown|          50|
|               Black|          46|
|         Other Asian|          22|
|             Chinese|           1|
|American Indian/A...|           1|
+--------------------+------------+

Bottom COMM Victim Count:
+--------------------+------------+
| descent_description|victim_count|
+--------------------+------------+
|Hispanic/Latin/Me...|        3314|
|               Black|        1152|
|               White|         435|
|               Other|         250|
|         Other Asian|         136|
|             Unknown|          31|
|American Indian/A...|          22|
|             Chinese|           4|
|              Korean|           4|
|            Filipino|           3|
|         AsianIndian|           1|
|           Guamanian|           1|
+--------------------+------------+
```

Τέλος μας ζητείται να εκτελέσουμε το ερώτημα με διαφορετικά configurations (διατηρώντας σταθερό και ίσο με 2 το πλήθος των executors) έτσι ώστε να δούμε πως αυτή ανταποκρίνεται στην κλιμάκωση του συνόλου των υπολογιστικών πόρων χρήσης. Τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

Executor Cores	Executor Memory	Processing Time (s)
1	2GB	121.39
2	4GB	106.00
4	8GB	100.25

Τα αποτελέσματα δείχνουν ότι η αύξηση των διαθέσιμων υπολογιστικών πόρων (πυρήνες και μνήμη ανά executor) βελτιώνει την απόδοση της επεξεργασίας του ερωτήματος. Συγκεκριμένα, παρατηρείται ότι καθώς αυξάνεται ο αριθμός των πυρήνων ανά executor και η διαθέσιμη μνήμη, ο χρόνος εκτέλεσης μειώνεται. Η διαφορά μεταξύ του πρώτου και του τελευταίου configuration (με 1 πυρήνα και 2 GB μνήμης έναντι 4 πυρήνων και 8 GB μνήμης) είναι σημαντική, μειώνοντας τον χρόνο εκτέλεσης από 121.39 σε 100.25 δευτερόλεπτα. Αυτό υποδεικνύει ότι η εφαρμογή είναι σε θέση να αξιοποιήσει την αύξηση πόρων, αν και η βελτίωση από το δεύτερο στο τρίτο configuration είναι μικρότερη, υποδηλώνοντας πιθανό φαινόμενο κορεσμού ή μειωμένη απόδοση κλιμάκωσης.

5 Ζητούμενο 5

Σε αυτό το ερώτημα, καλούμαστε να υπολογίσουμε ανά αστυνομικό τμήμα, τον αριθμό εγκλημάτων που έλαβαν χώρα πλησιέστερα σε αυτό, καθώς και την μέση απόσταση του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά.

Αρχικά, αντλούμε τα δεδομένα για τα εγκλήματα από το `crime dataset`, όπως και προηγουμένως φροντίζουμε να αφαιρέσουμε εγγραφές που αναφέρονται στο `Null Island (0,0)`. Τα δεδομένα για τις τοποθεσίες των αστυνομικών τμημάτων, αντλούνται από το αντίστοιχο dataset.

```
# crime data
d1 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2010_to_2019_20241101.csv",
    header=True, inferSchema=True)
d2 = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/
    Crime_Data_from_2020_to_Present_20241101.csv",
    header=True, inferSchema=True)
crime_data = d1.union(d2)

# remove NULL ISLAND (0,0)
crime_data = crime_data.filter(~((col('LAT') == 0) & (col('LON') == 0)))
crime_data = crime_data.withColumn("crime_geometry", ST_Point(F.col("LON"), F.col("LAT")))
crime_data = crime_data.select(col('crime_geometry'), col('AREA '), col('DR_NO'))

# police stations data
police_stations = spark.read.csv(
    "s3://initial-notebook-data-bucket-dblab-905418150721/LA_Police_Stations.csv",
    header=True, inferSchema=True).withColumnRenamed("X", "PS_LON").withColumnRenamed("Y", "PS_LAT")
police_stations = police_stations.withColumn("ps_geometry", ST_Point(F.col("PS_LON"), F.col("PS_LAT"
    ))).select(col('ps_geometry'), col('DIVISION'))
```

Για την παραγωγή των αποτελεσμάτων, αρχικά υπολογίζονται με `crossjoin` οι αποστάσεις όλων των εγκλημάτων από όλα τα αστυνομικά τμήματα. Για τον υπολογισμό των αποστάσεων, έγινε χρήση της συνάρτησης `ST_DistanceSphere`, η οποία υπολογίζει την απόσταση harversine μεταξύ δύο σημείων. Ωστόσο, επειδή τα σημεία είναι σχετικά κοντά μεταξύ τους, θα μπορούσαμε να χρησιμοποιήσουμε και την συνάρτηση `ST_Distance`, η οποία υπολογίζει την ευκλείδεια απόσταση μεταξύ 2 σημείων. Στο βήμα αυτό, θέλουμε για κάθε έγκλημα, να βρούμε το κοντινότερο αστυνομικό τμήμα. Ορίζουμε ένα `Window Function`, το οποίο κάνει partition τα δεδομένα που παρήχθησαν στο προηγούμενο βήμα με βάση την στήλη `DR_NO`.Το `DR_NO` οποίο είναι μοναδικό για κάθε σημείο στο dataset μας. Για κάθε ένα από αυτά τα partitions, κρατάμε το row με την ελάχιστη απόσταση μεταξύ αστυνομικού τμήματος και σημείου εγκλήματος. Τελικά, ομαδοποιούμε τα δεδομένα, που παρήχθησαν κρατώντας τις ελάχιστες αποστάσεις, με βάση το `DIVISION` (δηλαδή το αστυνομικό τμήμα) και βρίσκουμε τον μέσο όρο απόστασης ανά αστυνομικό τμήμα.

```
def query5(crime_data, police_stations):
    joined_df = crime_data.crossJoin(police_stations).withColumn(
        "distance_km",
        ST_DistanceSphere(F.col("crime_geometry"), F.col("ps_geometry")) / 1000
    )

    window_spec = Window.partitionBy("DR_NO").orderBy(asc("distance_km"))
    joined_df_with_rank = joined_df.withColumn("rank", F.row_number().over(window_spec))
    nearest_station_df = joined_df_with_rank.filter(F.col("rank") == 1).select(
        "DR_NO", "DIVISION", "distance_km"
    )

    result_df = nearest_station_df.groupBy("DIVISION").agg(
        F.count("*").alias("#"),
        F.mean("distance_km").alias("average_distance")
    ).orderBy(F.col("#").desc())

    result_df.show(n=result_df.count(), truncate=False)
```

Παρακάτω φαίνονται τα αποτελέσματα του query 5.

```
+----------------+------+------------------+
|DIVISION        |#     |average_distance  |
+----------------+------+------------------+
|HOLLYWOOD       |224340|2.076263960178719 |
|VAN NUYS        |210134|2.9533697428197816|
|SOUTHWEST       |188901|2.191398805780884 |
|WILSHIRE        |185996|2.5926655329787645|
```

```
|77TH STREET     |171827|1.716544971970096 |
|OLYMPIC         |170897|1.7236036971780955|
|NORTH HOLLYWOOD |167854|2.643006094141565 |
|PACIFIC         |161359|3.8500706553079165|
|CENTRAL         |153871|0.9924764374568908|
|RAMPART         |152736|1.5345341879190117|
|SOUTHEAST       |152176|2.4218662158881807|
|WEST VALLEY     |138643|3.035671216314069 |
|TOPANGA         |138217|3.2969548417555528|
|FOOTHILL        |134896|4.250921708424983 |
|HARBOR          |126747|3.7025615993565206|
|HOLLENBECK      |115837|2.680181237706819 |
|WEST LOS ANGELES|115781|2.7924572890341217|
|NEWTON          |111110|1.6346357397097429|
|NORTHEAST       |108109|3.6236655246040868|
|MISSION         |103355|3.6909426142785966|
|DEVONSHIRE      |77094 |2.824765412800822 |
+----------------+------+------------------+
```

Τέλος μας ζητείται να εκτελέσουμε το ερώτημα με διαφορετικά configurations χρησιμοποιώντας σταθερούς συνολικούς πόρους 8 cores και 16GB μνήμης. Τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

Executors	**Executor Cores**	**Executor Memory**	**Processing Time (s)**
2	4	8GB	19.10
4	2	4GB	37.56
8	1	2GB	41.06

Τα αποτελέσματα δείχνουν, πως ο χρόνος εκτέλεσης μειώνεται δραστικά όταν έχουμε περισσότερη μνήμη σε κάθε executor. Αυτό οφείλεται πιθανότατα στο γεγονός πως υπάρχει αρκετή μνήμη ώστε οι executors, να αποθηκεύουν ενδιάμεσα αποτελέσματα στην κύρια μνήμη. Σε περίπτωση που δεν επαρκεί η κύρια μνήμη, για τα ενδιάμεσα αποτελέσματα θα πρέπει να γίνουν αποθηκεύσεις στον δίσκο. Το πρώτο configuration παρουσιάζει την καλύτερη επίδοση, γιατί έχει αρκετή μνήμη για να αποφύγουμε I/O operations με τον δίσκο, αλλά και επειδή έχει αρκετό αριθμό cores ώστε να υπάρχει ένας ικανοποιητικός βαθμός παραλληλίας.