ΗΜΜΥ ΕΜΠ

Τομέας Τεχνολογίας, Πληροφορικής και Υπολογιστών

ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ

9^{Ο} ΕΞΑΜΗΝΟ

ΑΝΑΦΟΡΑ ΕΞΑΜΗΝΙΑΙΑΣ ΕΡΓΑΣΙΑΣ

Ομάδα 14

Δασκαλάκος Φώτης	***03117156***	***daskalakosfotis@yahoo.gr***
Παναγιωτόπουλος Νικήτας	***03119014***	***nikitas.panayiotopoulos4@gmail.com***

Github Link: https://github.com/Skatzhog/advanced-db-spark-project.git

Ζητούμενο 1:

Υλοποίηση του query 1 με 3 τρόπους, σε 4 executors με 1 core και 2GB memory:

- Dataframe με UDF, υλοποίηση στο query_1/query1_DF_UDF.ipynb
- Dataframe χωρίς UDF, υλοποίηση στο query_1/query1_DF.ipynb
- RRD , υλοποίηση στο query_1/query1_RDD.ipynb

Υλοποιήσεις:

Για τις υλοποιήσεις με dataframes ορίζουμε το crime_schema το οποίο χρησιμοποιείται από το read.csv του Spark για την εισαγωγή των δεδομένων του πρώτου και δεύτερου συνόλου δεδομένων σε dataframes.

```
# Functional Schema for the dataframe queries. The schema can alternatively be inferred
crime_schema = StructType([
        StructField("DR_NO", StringType()),
        StructField("Date Rptd", StringType()),
        StructField("DATE OCC", StringType()),
        StructField("TIME OCC", StringType()),
        StructField("AREA", StringType()),
        StructField("AREA NAME", StringType()),
        StructField("Rpt Dist No", StringType()),
        StructField("Part 1-2", StringType()),
        StructField("Crm Cd", StringType()),
        StructField("Crm Cd Desc", StringType()),
        StructField("Mocodes", StringType()),
        StructField("Vict Age", IntegerType()),
        StructField("Vict Sex", StringType()),
        StructField("Vict Descent", StringType()),
        StructField("Premis Cd", StringType()),
        StructField("Premis Desc", StringType()),
        StructField("Weapon Used Cd", StringType()),
        StructField("Weapon Desc", StringType()),
        StructField("Status", StringType()),
        StructField("Status Desc", StringType()),
        StructField("Crm Cd 1", StringType()),
        StructField("Crm Cd 2", StringType()),
        StructField("Crm Cd 3", StringType()),
        StructField("Crm Cd 4", StringType()),
        StructField("LOCATION", StringType()),
        StructField("Cross Street", StringType()),
        #StructField("LAT", DoubleType()),
        #StructField("LON", DoubleType()),
    ])
```

Στη συνέχεια στην περίπτωση της υλοποίησης με udf ορίζουμε βοηθητική συνάρτηση age group, φτιάχνουμε το age_group_udf και το χρησιμοποιούμε για να κατηγοριοποιήσουμε τα θύματα με βάση την ηλικία τους.

```python
def age_group(age):
    if age is None:
        return None
    if age < 18:
        return "Children"
    elif age <= 24:
        return "Young Adults"
    elif age <= 64:
        return "Adults"
    else:
        return "Elderly"


age_group_udf = udf(age_group, StringType())

result_udf = (
    agg_df
    .withColumn("age_group", age_group_udf(col("Vict Age")))
    .groupBy("age_group")
    .count()
    .orderBy(col("count").desc())
)
```

Αντίθετα στην υλοποίηση χωρίς udf χρησιμοποιούμε την συνάρτηση spark.when για να χωρίσουμε τα θύματα σε κατηγορίες ηλικίας.

```python
result_native = (
    agg_df
    .withColumn(
        "age_group",
        when(col("Vict Age") < 18, "Children")
        .when((col("Vict Age") >= 18) & (col("Vict Age") <= 24), "Young Adults")
        .when((col("Vict Age") >= 25) & (col("Vict Age") <= 64), "Adults")
        .when(col("Vict Age") > 64, "Elderly")
        .otherwise(None)
    )
    .groupBy("age_group")
    .count()
    .orderBy(col("count").desc())
)
```

Για την υλοποίηση με rdd χρησιμοποιούμε τη συνάρτηση csv.reader για να διαβάσουμε τα σύνολα δεδομένων 1 και 2.

```
def parse_csv(line):
    return next(csv.reader([line]))


sc = SparkSession \
    .builder \
    .appName("RDD query 1 execution") \
    .getOrCreate() \
    .sparkContext

rdd1 = sc.textFile("s3://initial-notebook-data-bucket-
            .map(parse_csv)

rdd2 = sc.textFile("s3://initial-notebook-data-bucket-
            .map(parse_csv)

crime_rdd_raw = rdd1.union(rdd2)
```

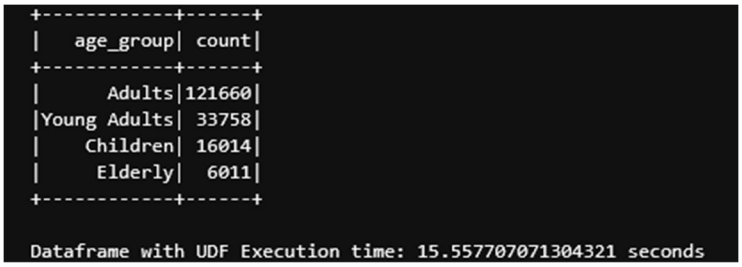
Στη συνέχεια ορίζουμε τις συναρτήσεις parse_age και age_bucket και χρησιμοποιούμε τις συναρτήσεις filter, map, reduceByKey και sortBy για να κατηγοριοποιήσουμε τα θύματα με βάση την ηλικία τους.

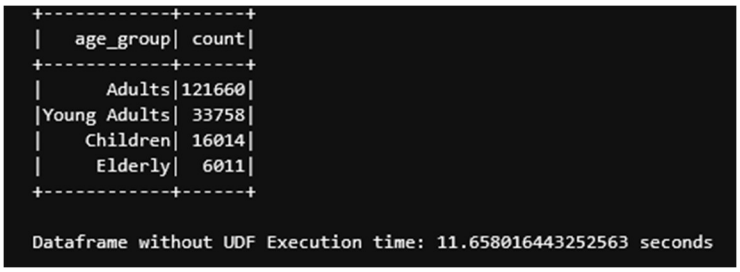
```
def parse_age(s):
    try:
        age = int(s)
        return age if age > 0 else None
    except:
        return None

def age_bucket(age):
    if age is None:
        return None
    if age < 18:
        return "Children"
    elif age <= 24:
        return "Young Adults"
    elif age <= 64:
        return "Adults"
    else:
        return "Elderly"

start = time.time()

agg_buckets = (
    crime_rdd_raw
    .filter(lambda r: len(r) > 11 and r[9] and "aggravated assault" in r[9].lower())
    # Convert age from string → int → bucket, then produce (bucket, 1)
    .map(lambda r: (age_bucket(parse_age(r[11])), 1))
    # drop rows with no valid bucket
    .filter(lambda x: x[0] is not None)
    .reduceByKey(lambda a, b: a + b)
    .sortBy(lambda x: x[1], ascending=False)
)
```

Στο τέλος τυπώνουμε τα αποτελέσματα χρησιμοποιώντας collect σε for loop(show λειτουργεί μόνο για dataframes).

```
# Collect to the driver to print
for group, cnt in agg_buckets.collect():
    print(group, cnt)
```

<u>Επίδοση των υλοποιήσεων, αποτελέσματα και σχολιασμός:</u>





Dataframe without UDF	11.66 seconds
Dataframe with UDF	15.56 seconds
RDD	17.90 seconds

Όπως βλέπουμε η υλοποίηση με Dataframes χωρίς udf είναι με διαφορά η πιο γρήγορη, και η υλοποίηση με RDDs είναι η πιο αργή.

Τα αποτελέσματα αυτά ταιριάζουν με τις προσδοκίες μας:

- Στην περίπτωση Dataframes χωρίς UDF, το Spark έχει στη διάθεσή του ολόκληρη τη λογική αλυσίδα υπολογισμών (Catalyst and Tungsten optimizations) και μπορεί να μεταβάλλει το σχέδιο των υπολογισμών (αναδιάταξη joins, filters, fusing multiple operators in 1 stage) και να μειώσει σημαντικά την πολυπλοκότητα.

- Αντίθετα, όταν χρησιμοποιούμε UDFs, ακόμα και απλά όπως στην περίπτωσή μας το Spark «βλέπει» απλά ένα μαύρο κουτί. Έτσι, οι βελτιστοποιήσεις είναι λιγότερες και φυσικά δεν μπορεί να επιταχυνθεί κώδικας γραμμένος στο udf, που στην περίπτωσή μας είναι και σε python (πολύ αργό).

- Όταν χρησιμοποιούμε RDD APIs τότε δεν υπάρχει κανένας optimizer και καθόλου codegen και το Spark δεν έχει καμία πληροφορία για το schema, άρα δεν μπορεί να κάνει optimizations. Γίνονται περισσότερα shuffles, object allocations, και αυξάνεται το python overhead (black box functions on JMV objects).

Ζητούμενο 2:

Στο δεύτερο ζητούμενο καλούμαστε να υλοποιήσουμε το Query 2 (σε configuration 4 executors, 1 core, 2GB memory) τόσο με DataFrame, όσο και με SQL api.

Αρχικά, στο αρχείο query2_DF.ipynb βρίσκεται το notebook με την υλοποίηση σε DataFrame api. Χρησιμοποιούμε τις συναρτήσεις .groupBy() και .join() για να ομαδοποιήσουμε τα καταγεγραμμένα εγκλήματα ανά χρόνο και φυλετική ομάδα του θύματος και στη συνέχεια να εξάγουμε τα ποσοστά εγκλημάτων:

```
#extract year from date
df_processed = df.withColumn("Year", year(to_timestamp(col("DATE OCC"), "yyyy MMM dd hh:mm:ss a")))

# Filter null values and group by year and descent
df_filtered = df_processed.filter(col("Year").isNotNull() & col("Vict Descent").isNotNull())
df_grouped = df_filtered.groupBy("Year", "Vict Descent") \
    .agg(count("*").alias("count"))

# Count total per Year and find percentage
df_yearly_totals = df_filtered.groupBy("Year").agg(count("*").alias("total_year"))
df_joined = df_grouped.join(df_yearly_totals, on="Year", how="inner")
df_with_percent = df_joined.withColumn("percentage", round((col("count") / col("total_year")) * 100, 1))
```

Χρησιμοποιώντας το window.partitionBy() ταξινομούμε τα δεδομένα κάθε χρονιάς ξεχωριστά στο ίδιο DataFrame και με την .filter() κρατάμε τις πρώτες 3 φυλετικές ομάδες σε εγκλήματα:

```
# window configuration: Per Year (partitionBy), order descending
windowSpec = Window.partitionBy("Year").orderBy(col("count").desc())

# keep top 3 groups per year
final_result = df_with_percent \
    .withColumn("rank", row_number().over(windowSpec)) \
    .filter(col("rank") <= 3) \
    .select(
        col("Year"),
        col("Vict Descent"),
        col("count").alias("#"),
        col("percentage").alias("%")
    ) \
    .orderBy(col("Year").desc(), col("#").desc())
```

Τέλος αλλάζουμε τα ονόματα των φυλετικών ομάδων εφόσον στο input csv ήταν διαχωρισμένα μόνο με τα αρχικά γράμματα:

```
#Format the Victim descent names
final_result_named = final_result.withColumn("Vict Descent",
    when(col("Vict Descent") == 'W', "White")
    .when(col("Vict Descent") == 'B', "Black")
    .when(col("Vict Descent") == 'H', "Hispanic/Latin/Mexican")
    .when(col("Vict Descent") == 'X', "Unknown"))
```

Στο αρχείο query2_SQL.ipynb βρίσκεται η υλοποίηση με SQL api. Χρησιμοποιώντας την .createOrReplaceTempView() μπορούμε να δημιουργήσουμε ένα πλήρες SQL query πάνω στο ίδιο dataset.

```
# To utilize as SQL tables in the SQL query
df.createOrReplaceTempView("crime_data")
```

Με διαδοχικές χρήσεις της SELECT εξάγουμε τον χρόνο και την φυλετική ομάδα, τα στατιστικά ανά χρόνο και ομάδα καθώς και τις 3 κορυφαίες «επιδόσεις» ανά έτος, προσαρμόζοντας πάλι τα ονόματα των ομάδων κατάλληλα:

```
query = """
WITH BaseData AS (
    -- Year extraction
    SELECT
        year(to_timestamp(`DATE OCC`, 'yyyy MMM dd hh:mm:ss a')) as Year,
        `Vict Descent`
    FROM crime_data
    WHERE `DATE OCC` IS NOT NULL AND `Vict Descent` IS NOT NULL
),
YearlyStats AS (
    -- Crime per Victim Descent group and year
    SELECT
        Year,
        `Vict Descent`,
        count(*) as count,
        sum(count(*)) OVER (PARTITION BY Year) as total_year,
        row_number() OVER (PARTITION BY Year ORDER BY count(*) DESC) as rank
    FROM BaseData
    GROUP BY Year, `Vict Descent`
)
-- Top 3 and Formatting
SELECT
    Year,
    CASE
        WHEN `Vict Descent` = 'W' THEN 'White'
        WHEN `Vict Descent` = 'B' THEN 'Black'
        WHEN `Vict Descent` = 'H' THEN 'Hispanic/Latin/Mexican'
        WHEN `Vict Descent` = 'X' THEN 'Unknown'
        ELSE `Vict Descent`
    END as Vict_Descent,
    count as `#`,
    ROUND((count / total_year) * 100, 1) as `%`
FROM YearlyStats
WHERE rank <= 3
ORDER BY Year DESC, count DESC
"""
```

Το κοινό αποτέλεσμα των 2 υλοποιήσεων για το query είναι ο εξής πίνακας με τις στήλες “Year”, “Vict_Descent”, “#”(απόλυτος αριθμός), “%”(ποσοστό):

Year	Vict_Descent	#	%
2025	Hispanic/Latin/Mexican	34	40.5
2025	Unknown	24	28.6
2025	White	13	15.5
2024	Hispanic/Latin/Mexican	28576	29.1
2024	White	22958	23.3
2024	Unknown	19984	20.3
2023	Hispanic/Latin/Mexican	69401	34.6
2023	White	44615	22.2
2023	Black	30504	15.2
2022	Hispanic/Latin/Mexican	73111	35.6
2022	White	46695	22.8
2022	Black	34634	16.9
2021	Hispanic/Latin/Mexican	63676	35.1
2021	White	44523	24.5
2021	Black	30173	16.6
2020	Hispanic/Latin/Mexican	61606	35.3
2020	White	42638	24.5
2020	Black	28785	16.5
2019	Hispanic/Latin/Mexican	72458	36.4

Ενώ και τα 2 δίνουν σωστά κοινό αποτέλεσμα τις κορυφαίες 3 ομάδες σε αριθμό και ποσοστό εγκλημάτων ανά έτος, αρχίζοντας από τα πιο πρόσφατα, παρατηρούμε πως η διαφορά τους στον χρόνο εκτέλεσης είναι επίσης μικρή.

- DataFrame:

```
Execution time: 19.89593768119812 seconds
```

- SQL:

```
Execution time: 14.321094989776611 seconds
```

Αρχικά είχαμε χρησιμοποιήσει και την .count() για να υπολογίσουμε και τον αριθμό των γραμμών στο τελικό DataFrame αλλά λόγω του Lazy Evaluation της Spark όλοι οι χρονοβόροι υπολογισμοί γίνονται με την εκτέλεση των actions και έχοντας τόσο την .count() όσο και την απαραίτητη .show() ο χρόνος έβγαινε διπλάσιος από τον αντίστοιχο της SQL. Χωρίς αυτόν βλέπουμε μικρότερη διαφορά που πιθανότατα δεν οφείλεται στο API μιας και τα δύο χρησιμοποιούν τον Spark Catalyst για να παράξουν το ίδιο Physical Plan που είναι και καταλύτης για τις καλές επιδόσεις και των 2. Χαρακτηριστικά στην υλοποίηση με DataFrame έχουμε χρησιμοποιήσει .join() ενώ στην SQL ποτέ κάτι αντίστοιχο. Συνεπώς, αλγοριθμικές επιλογές σαν και αυτή είναι που δημιουργούν αυτές τις μικρές διαφορές.

Ζητούμενο 3:

Υλοποίηση του query 3 με διάφορους τρόπους, σε 4 executors με 1 core και 2GB memory:

- Dataframes με χρήση των μεθόδων hint και explain, για να βρούμε τη στρατηγική join που χρησιμοποιεί ο catalyst optimizer και να δοκιμάσουμε επίσης τις στρατηγικές BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL στο query_3/query3_DF.ipyn2b
- RDD APIs στο query_3/query3_RDD.ipynb

Υλοποιήση Dataframes:

Στα dataframes ακολουθούμε την ακόλουθη στρατηγική για να ταξινομήσουμε και να εμφανίσουμε με φθίνουσα σειρά συχνότητας εμφάνισης τις μεθόδους διάπραξης εγκλημάτων με τους αντίστοιχους κωδικούς τους:

Αφού εισάγουμε τα δεδομένα από τα data-sets Los Angeles Crime Data (2010-2019), Los Angeles Crime Data (2020-) και MO Codes και μετατρέψουμε τα δεδομένα των MO Codes από αρχείο txt σε πιο εύχρηστη μορφή, φτιάχνουμε το dataframe mocodes_exploded το οποίο περιέχει μία γραμμή για κάθε MO Code που εμφανίζεται σε κάποιο έγκλημα. Στη συνέχεια φτιάχνουμε το mo_counts dataframe το οποίο περιέχει τον αριθμό που εμφανίζεται κάθε distinct MO Code στα datasets LA Crime data.

```
# Keep rows with non-null Mocodes
mocodes_exploded = (
    crime_df
    .filter(col("Mocodes").isNotNull())
    .select(
        explode(
            split(trim(col("Mocodes")), r"\s+")  # split on 1+ spaces
        ).alias("MO_CODE")
    )
    .filter(col("MO_CODE") != "")  # drop empty pieces
)

mo_counts = (
    mocodes_exploded
    .groupBy("MO_CODE")
    .count()
)
```

Κάνουμε τους υπολογισμούς και παίρνουμε τα αποτελέσματα:

```
+-------+-----+
|MO_CODE|count|
+-------+-----+
|0543   |221  |
|1512   |41   |
|0401   |13049|
|1280   |29   |
|0201   |608  |
|1008   |1071 |
|0371   |15199|
|0385   |41926|
|0908   |1837 |
|0535   |82   |
+-------+-----+
only showing top 10 rows
```

```
+-------+------------------+
|MO_CODE|MO_DESC           |
+-------+------------------+
|0100   |Suspect Impersonate|
|0101   |Aid victim        |
|0102   |Blind             |
|0103   |Physically disabled|
|0104   |Customer          |
|0105   |Delivery          |
|0106   |Doctor            |
|0107   |God               |
|0108   |Infirm            |
|0109   |Inspector         |
+-------+------------------+
only showing top 10 rows

Execution time: 2.4645142555236816 seconds
```

Στη συνέχεια υπολογίζουμε το query με ένα πολύ απλό join:

```
result = (
    mo_counts
    .join(mo_dict_df, on="MO_CODE", how="left")
    .orderBy(desc("count"))
)
```

Σε ακόλουθα cells γίνεται το ίδιο query με μόνη διαφορά ότι δίνονται σαν hint οι στρατηγικές join BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL:

```
result_broadcast = (
    mo_counts
    .join(
        mo_dict_df.hint("broadcast"),
        on="MO_CODE",
        how="left"
    )
    .orderBy(desc("count"))
)
```

<u>Επίδοση Joins και physical plans:</u>

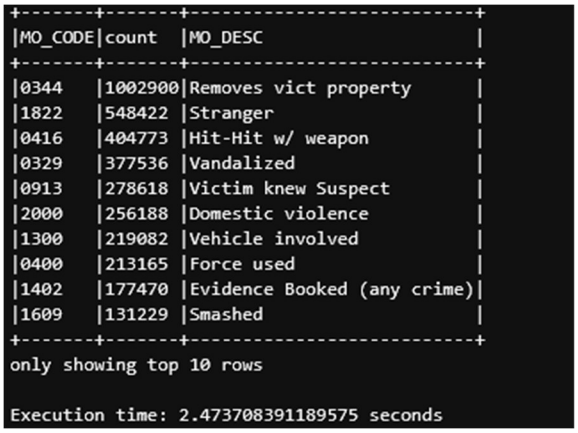
Join χωρίς hint:

Το physical plan που δίνει η μέδοδος explain είναι το εξής:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [count#153L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(count#153L DESC NULLS LAST, 1000), ENSURE_REQUIREMENTS, [plan_id=769]
      +- Project [MO_CODE#149, count#153L, MO_DESC#144]
         +- BroadcastHashJoin [MO_CODE#149], [MO_CODE#143], LeftOuter, BuildRight, false
            :- HashAggregate(keys=[MO_CODE#149], functions=[count(1)], schema specialized)
            :  +- Exchange hashpartitioning(MO_CODE#149, 1000), ENSURE_REQUIREMENTS, [plan_id=762]
            :     +- HashAggregate(keys=[MO_CODE#149], functions=[partial_count(1)], schema specialized)
            :        +- Filter NOT (MO_CODE#149 = )
            :           +- Generate explode(split(trim(Mocodes#10, None), \s+, -1)), false, [MO_CODE#149]
            :              +- Union
            :                 :- Filter isnotnull(Mocodes#10)
            :                 :  +- FileScan csv [Mocodes#10] Batched: false, DataFilters: [isnotnull(Mocodes#10)], Format: CSV, Loca
            :                 +- Filter isnotnull(Mocodes#73)
            :                    +- FileScan csv [Mocodes#73] Batched: false, DataFilters: [isnotnull(Mocodes#73)], Format: CSV, Loca
            +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]),false), [plan_id=765]
               +- Project [trim(split(value#141, \s+, 2)[0], None) AS MO_CODE#143, trim(split(value#141, \s+, 2)[1], None) AS MO_DESC
                  +- Filter (NOT (trim(split(value#141, \s+, 2)[0], None) = ) AND isnotnull(trim(split(value#141, \s+, 2)[0], None)))
                     +- FileScan text [value#141] Batched: false, DataFilters: [NOT (trim(split(value#141, \s+, 2)[0], None) = ), isn
```

Βλέπουμε ότι το Spark κάνει broadcast hash join.

Αποτελέσματα και χρόνος εκτέλεσης:



```
+-------+-------+--------------------------+
|MO_CODE|count  |MO_DESC                   |
+-------+-------+--------------------------+
|0344   |1002900|Removes vict property     |
|1822   |548422 |Stranger                  |
|0416   |404773 |Hit-Hit w/ weapon         |
|0329   |377536 |Vandalized                |
|0913   |278618 |Victim knew Suspect       |
|2000   |256188 |Domestic violence         |
|1300   |219082 |Vehicle involved          |
|0400   |213165 |Force used                |
|1402   |177470 |Evidence Booked (any crime)|
|1609   |131229 |Smashed                   |
+-------+-------+--------------------------+
only showing top 10 rows

Execution time: 2.473708391189575 seconds
```

BROADCAST join hint: 3.11 seconds

MERGE join hint: 3.14 seconds

SHUFFLE_HASH join hint: 3.22 seconds

SHUFFLE_REPLICATE_NL join hint: 2.53 seconds

Τα Physical plans παραμένουν ίδια με μόνη διαφορά τη χρήση της στρατηγικής join που έχουμε δώσει ως hint.

Παρατηρούμε ότι οι χρόνοι εκτέλεσης κυμαίνονται στα 2 με 3.5 δευτερόλεπτα.

Και τα δύο tables mo_dict_df και mo_counts αποτελούνται από δύο μόνο columns και μερικές εκατοντάδες γραμμές. Επομένως, περιμένουμε ότι το χρονικό κόστος broadcast, shuffle, sort etc είναι σχεδόν αμελητέο και το μεγαλύτερο κόστος εκτέλεσης βρίσκεται στο overhead job setup και προεπεξεργασία των tables. Είναι δηλαδή λογικό ότι όλοι οι χρόνοι εκτέλεσης είναι σχετικά κοντά μεταξύ τους, και ανά εκτέλεση κυμαίνονται σε αυτό το μικρό διάστημα.

<u>Υλοποίηση RDD APIs:</u>

Όπως και στην υλοποίηση Dataframes, πρώτα ορίζουμε και υπολογίζουμε τον πίνακα mo_counts_rdd καθαρίζοντας το crime table, κάνοντας flatmap κάθε γραμμή σε distinct MO codes και μετρώντας occurrences με reduceByKey.

Στη συνέχεια, κάνουμε broadcast το mo_dict (γνωρίζουμε ότι είναι μικρό) σε όλα τα executors και υπολογίζουμε το τελικό αποτέλεσμα κάνοντας map στο mo_counts_rdd τους κωδικούς MO του mo_dict. Τέλος, κάνουμε sort.

Η υλοποίηση αυτή είναι κοντά σε λογική με το broadcast hash join του Spark.

Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε join ως εξής:
mo_counts_rdd.join(sc.parallelize(mo_dict.items()))

Ωστόσο, κάτι τέτοιο δεν θα ήταν αποδοτικό γιατί θα κάναμε shuffle τόσο για τα mo_counts_rdd όσο και για το mo_dict (Στην υλοποίηση που χρησιμοποιήσαμε, το lookup για κάθε record σε κάθε executor γίνεται σε O(1), ενώ με join θα στέλναμε τμήματα του mo_dict επανειλημμένα)

```python
MOCODES_IDX = 10


# drop header rows and any malformed rows
crime_rdd = crime_rdd_raw.filter(
    lambda row: len(row) > MOCODES_IDX and row[0] != "DR_NO"
)

# explode mocodes: one record per single code
mocodes_rdd = crime_rdd.flatMap(
    lambda row: [
        code for code in (row[MOCODES_IDX] or "").split()
        if code
    ]
)

mo_counts_rdd = mocodes_rdd.map(lambda code: (code, 1)) \
                           .reduceByKey(lambda a, b: a + b)
# (MO_CODE, count)

start = time.time()

bc_mo_dict = sc.broadcast(mo_dict)

# (code, count, description)
result_rdd = mo_counts_rdd.map(
    lambda kv: (kv[0], kv[1], bc_mo_dict.value.get(kv[0], "UNKNOWN"))
)

# sort by count descending
result_sorted = result_rdd.sortBy(lambda x: -x[1])

# e.g. show top 10
for code, cnt, desc in result_sorted.take(10):
    print(code, cnt, desc)

end = time.time()
print("Execution time:", end - start, "seconds")
```

```
0344 1002900 Removes vict property
1822 548422 Stranger
0416 404773 Hit-Hit w/ weapon
0329 377536 Vandalized
0913 278618 Victim knew Suspect
2000 256188 Domestic violence
1300 219082 Vehicle involved
0400 213165 Force used
1402 177470 Evidence Booked (any crime)
1609 131229 Smashed
Execution time: 9.286213874816895 seconds
```

Χρονικά, η εκτέλεση γίνεται σε 9.29 δευτερόλεπτα.

DF Default	2.47 seconds
DF Broadcast Hash join	3.11 seconds
DF Merge join	3.14 seconds
DF Shuffle Hash join	2.53 seconds
RDD join	9.29 seconds

Συμπέρασματα:

Παρατηρούμε ότι η εκτέλεση με Dataframes είναι συστηματικά πιο γρήγορη από την εκτέλεση με RDD APIs, ανεξάρτητα από τη μέθοδο του join. Επίσης, στα table sizes στα οποία εκτελούμε τα join δεν υπάρχει μεγάλη διαφορά μεταξύ των διάφορων στρατηγικών.

- Στην περίπτωση που ένα από τα δύο tables είναι μεγάλα θα περιμέναμε το Broadcast Hash join να είναι το πιο γρήγορο(δεν κάνει shuffle το μεγάλο table), και μετά τα Sort-Merge και Shuffle Hash joins(και τα δύο κάνουν shuffle το μεγάλο table). Θα περιμέναμε το Shuffle Replicate NL να είναι το πιο αργό.
- Στην περίπτωση που και τα δύο tables μας ήταν μεγάλα θα περιμέναμε το Sort-Merge join να είναι το πιο γρήγορο ή το Shuffle Hash, ανάλογα με τη μορφή των δεδομένων. Στη δικιά μας περίπτωση που η μνήμη είναι περιορισμένη μάλλον το Sort-Merge θα ήταν πιο γρήγορο. Το Shuffle Replicate NL και το broadcast join είναι και τα δύο μη πρακτικά σε αυτήν την περίπτωση.

Ζητούμενο 4

Στο τέταρτο ζητούμενο καλούμαστε να υλοποιήσουμε το Query 4, είτε με DataFrame είτε με SQL api, σε 3 διαφορετικά configurations:

i. 2 executors, 1 core, 2GB memory
ii. 2 executors, 2 core, 4GB memory
iii. 2 executors, 4 core, 8GB memory.

Κάθε αρχείο "query_4" που βρίσκεται στο αποθετήριο περιέχει τον ίδιο κώδικα με μόνη διαφορά τα παραπάνω configurations πόρων του spark.

Επιλέγουμε DataFrame API μιας και το Spark υλοποιεί και τα 2 χρησιμοποιώντας τους ίδιους optimizers και έχουν παρόμοια καλή απόδοση.

Σε αυτό το query θα χρειαστεί να υπολογίσουμε την απόσταση του κάθε καταγεγραμμένου εγκλήματος από κάθε αστυνομικό τμήμα πριν βρούμε το κοντινότερο και τα αθροίσουμε για να υπολογίσουμε τα στατιστικά που χρειαζόμαστε. Συνεπώς, θα χρησιμοποιήσουμε .crossjoin() συμπεριλαμβάνοντας την παρακάτω συνάρτηση για τον υπολογισμό της απόστασης:

```
# define function for counting distance with sedona
def get_distance(lat1_col, lon1_col, lat2_col, lon2_col):
    p1 = ST_Point(lon1_col.cast("double"), lat1_col.cast("double"))
    p2 = ST_Point(lon2_col.cast("double"), lat2_col.cast("double"))
    return ST_DistanceSphere(p1, p2) / 1000.0  # km
```

Με τις συναρτήσεις ST_Point() και ST_distanceSphere() του sedona, μετατρέπουμε τα γεωγραφικά μήκη και πλάτη σε σημεία και στη συνέχεια υπολογίζουμε την απόστασή τους λαμβάνοντας υπόψη την καμπυλότητα της Γης. Θα το εφαρμώσουμε σε κάθε συνδυασμό γεωγραφικών συντεταγμένων από το LA_Police_Stations.csv και τα "LA_Crime_Data" αρχεία:

```
# Cross join with stations, compute distance to each station seperately
crimes_stations_dist_df = (
    filtered_crimes_df
    .crossJoin(
        stations_df.select(
            col("division"),
            col("Y"), # Lat
            col("X") # Lon
        )
    )
    .withColumn(
        "distance",
        get_distance(col("LAT"), col("LON"), col("Y"), col("X"))
    )
)
```

Το αποτέλεσμα των περεταίρω υπολογισμών είναι ο εξής πίνακας με τη μέση απόσταση των εγκλημάτων που συνέβησαν κοντινότερα στο εκάστοτε αστυνομικό τμήμα καθώς και το σύνολό τους στην στήλη "#".

division	average_distance_km	#
HOLLYWOOD	2.077	225515
VAN NUYS	2.953	211130
SOUTHWEST	2.191	189565
WILSHIRE	2.593	187061
77TH STREET	1.717	172558
OLYMPIC	1.725	172353
NORTH HOLLYWOOD	2.643	168655
PACIFIC	3.853	162514
CENTRAL	0.993	154952
SOUTHEAST	2.422	153746
RAMPART	1.535	153690
TOPANGA	3.298	141070
WEST VALLEY	3.039	139820
FOOTHILL	4.251	135381
HARBOR	3.702	127370
HOLLENBECK	2.677	116558
WEST LOS ANGELES	2.79	116308
NEWTON	1.635	111628
NORTHEAST	3.623	108549
MISSION	3.685	105331
DEVONSHIRE	2.824	81226

Για την εκτέλεση του Query 4 χρησιμοποιούμε 2 executors με τη διαφορά υπολογιστικών πόρων να εντοπίζεται μόνο στους πυρήνες και την μνήμη κάθε φορά, τα οποία διαδοχικά διπλασιάζονται. Παρατηρούμε πως αναμενόμενα οδηγούμαστε σε καλύτερους χρόνους εκτέλεσης για κάθε αύξηση πόρων. Συγκεκριμένα, αυξάνοντας τους πυρήνες υπάρχει μεγαλύτερη δυνατότητα για παραλληλοποίηση, ενώ η ανάλογη αύξηση της μνήμης βοηθά σε αυτό καθώς περισσότερα δεδομένα είναι διαθέσιμα στον κάθε επεξεργαστή για ταχύτατη προσπέλαση όταν τα χρειαστεί.

i. (2,1,2)

```
Execution time: 38.64295053482056 seconds
```

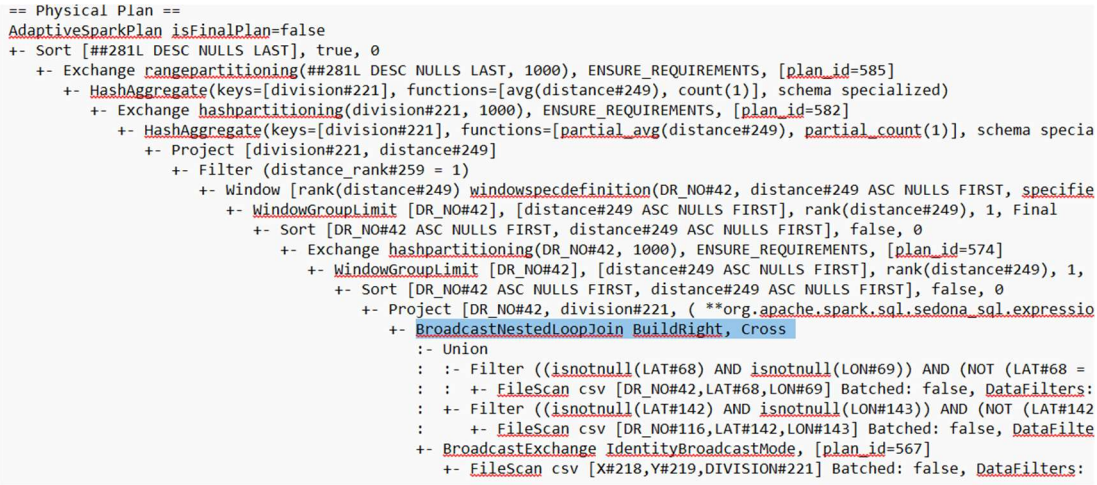
ii. (2,2,4)

```
Execution time: 27.015095949172974 seconds
```

iii. (2,4,8)

```
Execution time: 20.58036732673645 seconds
```

Βλέπουμε παρόλα αυτά πως η χρήση περισσότερων πόρων δεν αυξάνει δραματικά την απόδοση για πάντα. Αυτό μπορεί να οφείλεται στις μεγάλες καθυστερήσεις επικοινωνίας(Overhead) των επεξεργαστών λόγω μεγαλύτερης παραλληλοποίησης, όπως και σε περιορισμένες ανάγκες σε μνήμη που είχαν ήδη καλυφθεί από την μικρότερη(4GB) σε πολύ μεγάλο βαθμό. Επομένως η χρυσή τομή ανάμεσα στην ταχύτητα εκτέλεσης και τον περιορισμό των πόρων για το Query 4, είναι ο συνδυασμός 2 cores και 4 GB memory.



Τέλος, με χρήση της μεθόδου explain παρατηρούμε ότι όλες οι εκτελέσεις χρησιμοποιούν BroadcastNestedLoopJoin για να φτιάξουμε το καρτεσιανό γινόμενο των filtered_crimes_df και stations_df.

Η άλλη εναλλακτική του Spark για cross join είναι η ShuffleReplicateNL, η οποία δεν είναι απαραίτητη γιατί στα μεγέθη πίνακα που έχουμε μπορεί να γίνει broadcast.

Ζητούμενο 5

Στο πέμπτο ζητούμενο καλούμαστε να υλοποιήσουμε το Query 5, είτε με DataFrame είτε με SQL api, σε 3 διαφορετικά configurations:

iv. 2 executors, 4 core, 8GB memory
v. 4 executors, 2 core, 4GB memory
vi. 8 executors, 1 core, 2GB memory.

Κάθε αρχείο “query_4” που βρίσκεται στο αποθετήριο περιέχει τον ίδιο κώδικα με μόνη διαφορά τα παραπάνω configurations πόρων του spark.

Στο Query 5 θα χρησιμοποιήσουμε επιπλέον τα αρχεία LA_Census_Blocks_2020.geojson και LA_income_2021.csv που περιέχουν τον αριθμό εγκλημάτων ανά block στην πόλη του Los Angeles καθώς και το κατά κεφαλήν εισόδημα ανά περιοχή. Η αρχική επεξεργασία του geojson αρχείου έγινε με τον παρακάτω κώδικά που είχε δοθεί στο εργαστήριο:

```
geojson_path = "s3://initial-notebook-data-bucket-dblab-905418150721/project_data/LA_Census_Blocks_2020.geojson"
blocks_df = sedona.read.format("geojson") \
            .option("multiLine", "true").load(geojson_path) \
            .selectExpr("explode(features) as features") \
            .select("features.*")
# Formatting magic
flattened_df = blocks_df.select( \
               [col(f"properties.{col_name}").alias(col_name) for col_name in \
               blocks_df.schema["properties"].dataType.fieldNames()] + ["geometry"]) \
            .drop("properties") \
            .drop("type")
```

Στη συνέχεια χρησιμοποιούμε τη sum() για να υπολογίσουμε τον συνολικό αριθμό σπιτιών και τον πληθυσμό ανά μπλοκ κρατώντας παράλληλα τις γεωγραφικές συντεταγμένες σε ξεχωριστή στήλη:

```
from pyspark.sql.functions import sum as _sum

#Filter data only from LA, do summations for Housing and Population
group_flattened = (
    flattened_df
    .select("COMM", "POP20", "ZCTA20", "HOUSING20", "geometry")
    .filter(
        (col("ZCTA20") > 0) &
        (col("HOUSING20") > 0) &
        (col("POP20") > 0) &
        (col("COMM") != "")
    )
    .groupBy("COMM", "ZCTA20")
    .agg(
        _sum("POP20").alias("Total_POP"),
        _sum("HOUSING20").alias("Total_Housing"),
        ST_Union_Aggr("geometry").alias("geometry")
    )
)
```

Πραγματοποιούμε .join() με το dataframe των δεδομένων εισοδήματος, με βάση τον ταχυδρομικό κώδικα. Υπολογίζουμε το κατά κεφαλήν εισόδημα ανά περιοχή της πόλης, χρησιμοποιώντας το συνολικό εισόδημα και τον συνολικό πληθυσμό:

```
# Join Census population/housing data (group_flattened) with median income data (median_df)
# Join key: ZCTA20 (from Census) == Zip Code (from income dataset)
joined = (
    group_flattened
    .join(median_df, group_flattened["ZCTA20"] == median_df["Zip Code"]) #Broadcast Hash Join
    .withColumn(
        "Estimated Median Income",
        regexp_replace(col("Estimated Median Income"), "[^0-9]", "") # Remove non-digits: "$68,000"
    )
    .withColumn(
        "ZIP_Total_Income",
        (col("Estimated Median Income") * col("Total_Housing"))
    )
    .groupBy("COMM") # Aggregate at the community ("COMM") level
    .agg(
        _sum("Total_POP").alias("Total_COMM_Pop"),
        _sum("ZIP_Total_Income").alias("COMM_Total_Income"),
        ST_Union_Aggr("geometry").alias("geometry") # merges ZIP shapes into COMM shape
    )
    # Compute GDP per capita per community
    .withColumn(
        "GDP_Per_Capita",
        (col("COMM_Total_Income")/col("Total_COMM_Pop"))
    )
    .select("COMM", "GDP_Per_Capita", "geometry")
)

joined.explain()
```

Έπειτα αφού φιλτράρουμε το dataframe κρατώντας μόνο τα έτη 2020 και 2021 πραγματοποιούμε 2ο join() με βάση τις συντεταγμένες, για τις οποίες ελέγχουμε με το .ST_Within() εάν βρίσκονται εντός της εκάστοτε περιοχής που περιγράφεται από τη στήλη “COMM”:

```
final_crimes_df = (
    joined
    .join(crime_data_20_21_df, ST_Within(crime_data_20_21_df.geom, joined.geometry), "inner") # Range Join
    .groupBy("COMM")
    .agg(
        count("*").alias("#")
    )
)
```

Κάνουμε join() με το αρχικό flattened_df και δημιουργούμε την στήλη “Crimes_Per_Capita” υπολογίζοντας τον αριθμό εγκλημάτων που πραγματοποιήθηκαν ανά περιοχή προς τον πληθυσμό της περιοχής εκείνης:

```
#join Cencus with crimes per COMM. Join key is Zip Code. Result is crimes per capita in every area.
crime_joined = (
    group_flattened
    .join(final_crimes_df, on="COMM", how="inner") # Broadcast Hash Join
    .withColumn(
        "Crimes_Per_Capita",
        (col("#")/col("Total_Pop"))
    )
    .select("COMM", "Crimes_Per_Capita")
)
```

Τέλος, κάνουμε join() με τα δεδομένα για το κατά κεφαλήν εισόδημα και υπολογίζουμε την συσχέτιση με τη συνάρτηση corr():

```python
#Corelation while using all areas
start1 = time.time()
corr_joined = (
    joined
    .join(crime_joined, on="COMM", how="inner") # Sort Merge Join
    .select("COMM", "GDP_Per_Capita", "Crimes_Per_Capita")
)

full_corr = corr_joined.select(
    corr("GDP_Per_Capita", "Crimes_Per_Capita")
).collect()[0][0]
```

Επαναλαμβάνουμε το τελευταίο βήμα αφού φιλτράρουμε τόσο για τις κορυφαίες, όσο και για τις τελευταίες περιοχές ως προς το GDP_Per_Capita. Τα αποτελέσματα μαζί με τους χρόνους εκτέλεσης για το κάθε configuration φαίνονται παρακάτω:

i. (2 executors, 4 core, 8GB memory)

```
Full data Correlation = 0.05699006354516788
Full data correlation execution time: 23.830943822860718 seconds
Top 10 areas Correlation = 0.2930302995361155
Top 10 areas execution time: 16.838924646377563 seconds
Bottom 10 areas Correlation = 0.17968480714580115
Bottom 10 areas execution time: 15.748841047286987 seconds
```

ii. (4 executors, 2 core, 4GB memory)

```
Full data Correlation = 0.05699006354516789
Full data correlation execution time: 26.067071676254272 seconds
Top 10 areas Correlation = 0.2930302995361155
Top 10 areas execution time: 20.86649990081787 seconds
Bottom 10 areas Correlation = 0.17968480714580115
Bottom 10 areas execution time: 16.88815140724182 seconds
```

iii. (8 executors, 1 core, 2GB memory)

```
Full data Correlation = 0.05699006354516789
Full data correlation execution time: 27.428951501846313 seconds
Top 10 areas Correlation = 0.2930302995361155
Top 10 areas execution time: 19.404740571975708 seconds
Bottom 10 areas Correlation = 0.17968480714580115
Bottom 10 areas execution time: 17.249613046646118 seconds
```

Παρατηρούμε πως ο χρόνος εκτέλεσης για τα φιλτραρισμένα για τις κορυφαίες 10 και τελευταίες 10 περιοχές, είναι σαφώς μικρότερος από τον χρόνο εκτέλεσης του query για το σύνολο των περιοχών του LA. Αυτό πέρα από τα λιγότερα δεδομένα που καλείται να διαχειριστεί το spark πριν από το τελευταίο join(), οφείλεται κυρίως στη σειρά εκτέλεσης του κάθε action. Το 1ο action χρειάζεται να κάνει setup το περιβάλλον του Spark, αρχικοποίηση του Sedona, στήσιμο του JVM κτλ.

Όσον αφορά τα διαφορετικά configurations, παρά την αύξηση των executors οι χρόνοι αντί να μειώνονται, αντιθέτως παρουσιάζουν μικρή αύξηση. Αυτό είναι λογικό, καθώς αυξάνονται οι workers, χρειάζεται περισσότερο task scheduling από το Spark. Επίσης, γίνονται περισσότερα broadcasts και εμφανίζεται μεγαλύτερο network overhead, καθώς αυξάνεται η παραλληλοποίηση. Επιλέον, όσο πολλαπλασιάζονται οι executors, μειώνεται η διαθέσιμη μνήμη σε κάθε έναν και αυξάνεται η πιθανότητα να γίνονται memory spills στον δίσκο. Τέλος, τα συνολικά cores που

χρησιμοποιούνται σε κάθε εκτέλεση παραμένουν ίδια, κάτι που εξηγεί το γεγονός ότι κάθε φορά που αυξάνονται οι executors απλά αυξάνεται το συνολικό Overhead και εν τέλη ο χρόνος εκτέλεσης.

Όσον αφορά τις στρατηγικές Join που επέλεξε το spark για τα διάφορα joins:

1) **group_flattened ⋈ crimes_df → Broadcast Hash Join**

   Το group_flattened είναι πολύ μικρό σε μέγεθος (join στο πεδίο COMM), γεγονός που επιτρέπει τον εύκολο και αποδοτικό broadcast του πίνακα.

2) **crime_data_2021_df ⋈ range → Range Join**

   Πρόκειται για χωρικό join που υλοποιείται από το Sedona και εξαρτάται από γεωμετρικά predicates. Για τον λόγο αυτό δεν μπορεί να χρησιμοποιηθεί hash join. Επιπλέον, αποτελεί το πιο υπολογιστικά απαιτητικό join του query.

3) **group_flattened ⋈ median_df → Broadcast Hash Join**

   Το median_df είναι επίσης πολύ μικρό σε μέγεθος, με αποτέλεσμα να επιλέγεται broadcast hash join.

4) **crime_joined ⋈ joined → Sort Merge Join**

   Και οι δύο πίνακες είναι μεγάλοι, επομένως το broadcast δεν είναι εφικτό. Το Spark επιλέγει μεταξύ ShuffleHashJoin και SortMergeJoin. Ωστόσο, δεδομένου ότι απαιτείται shuffle στο πεδίο COMM, πραγματοποιείται ταξινόμηση κατά τη διάρκεια του shuffle και στη συνέχεια merge, οδηγώντας στην επιλογή του SortMergeJoin.

5) **Top-10 / Bottom-10: top_10 ⋈ crime_joined→ Broadcast Hash Join**

   Τα top_10 και bottom_10 είναι και τα δύο πολύ μικρά, άρα γίνεται εύκολα broadcast.

   Συνοπτικά:

Stage	Join chosen	Reason
group_flattened ⋈ crimes	BroadcastHash	small ⋈ huge
crimes ⋈ geometry	RangeJoin	spatial predicate
group_flattened ⋈ median	BroadcastHash	tiny ⋈ tiny
crime_joined ⋈ population	SortMerge	huge ⋈ huge
top/bottom ⋈ crime_joined	BroadcastHash	10 rows ⋈ huge