ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΒΑΣΕΩΝ ΓΝΩΣΕΩΝ ΚΑΙ ΔΕΔΟΜΕΝΩΝ

Προχωρημένα Θέματα Βάσεων Δεδομένων
Ακαδημαϊκό έτος 2024-25, 9ο Εξάμηνο
Διδάσκων: Δημήτριος Τσουμάκος
Υπεύθυνος Εργαστηρίου: Νικόλαος Χαλβαντζής

Εξαμηνιαία Εργασία

Ομάδα 43
Σταύρος Λαζόπουλος 03120843
GitHub repository : https://github.com/StavrosLzp/NTUA-Advanced-DB-Project-2024.git
Jupyter Notebook : group43-sagemaker-notebook on AWS cloud

Περιγραφή Εργασίας

Στην παρούσα εξαμηνιαία εργασία ζητείται ανάλυση σε (μεγάλα) σύνολα δεδομένων, εφαρμόζοντας επεξεργασία με τεχνικές που εφαρμόζονται σε data science projects.

Τα εργαλεία που θα χρησιμοποιηθούν στα πλαίσια του project είναι τα Apache Hadoop (version>=3.0) και Apache Spark (version>=3.5). Καλούμαστε να χρησιμοποιήσετε τους πόρους στο ειδικά διαμορφωμένο περιβάλλον που μας έχει παραχωρηθεί στο AWS cloud. Συγκεκριμένα χρησιμοποιούμε python με εργαλεία της βιβλιοθήκης pyspark για να επεξεργαστούμε δεδομένα που βρίσκονται στο cloud σε S3 buckets.

Δεδομένα

Σύνολο Δεδομένων	S3 URI
Los Angeles Crime Data (2010-2019)	s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/Crime_Data_from_2010_to_2019_20241101.csv
Los Angeles Crime Data (2020-)	s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/Crime_Data_from_2020_to_Present_20241101.csv
LA Police Stations	s3://initial-notebook-data-bucket-dblab-905418150721/LA_Police_Stations.csv
Median Household Income by Zip Code	s3://initial-notebook-data-bucket-dblab-905418150721/LA_income_2015.csv
2010 Census Blocks	s3://initial-notebook-data-bucket-dblab-905418150721/2010_Census_Blocks.geojson
Race and Ethnicity Codes	s3://initial-notebook-data-bucket-dblab-905418150721/RE_codes.csv

Το βασικό σύνολο δεδομένων που θα χρησιμοποιηθεί στην εργασία (**Los Angeles Crime Data**) προέρχεται από το δημόσιο αποθετήριο δεδομένων της κυβέρνησης των Ηνωμένων Πολιτειών της Αμερικής. Συγκεκριμένα, περιλαμβάνει δεδομένα καταγραφής εγκλημάτων για το Los Angeles από το 2010 μέχρι σήμερα.

Συμπληρωματικά με τα παραπάνω δεδομένα, θα χρησιμοποιηθεί μια σειρά δεδομένων μικρότερου όγκου τα οποία επίσης είναι διαθέσιμα σε δημόσια αποθετήρια ή πηγές. Αυτά είναι το **2010 Census Blocks**, ένα σύνολο δεδομένων που παρουσιάζει απογραφικά στοιχεία που αφορούν στην Κομητεία του Los Angeles για το έτος 2010 σε geojson format, το **Median Household Income by Zip Code**, ένα ακόμα μικρό σύνολο δεδομένων που περιέχει δεδομένα σχετικά με το μέσο εισόδημα ανά νοικοκυριό και ταχυδρομικό κώδικα (ZIP Code) στην Κομητεία του Los Angeles, το **LA Police Stations**, μικρό σύνολο δεδομένων που περιέχει δεδομένα σχετικά με την τοποθεσία των 21 αστυνομικών τμημάτων που βρίσκονται στην πόλη του Los Angeles, και το **Race and Ethnicity codes**, ένα μικρό σύνολο δεδομένων που περιέχει τις πλήρες περιγραφές που αντιστοιχούν στην κωδικοποίηση του φυλετικού προφίλ που χρησιμοποιείται στο βασικό σύνολο δεδομένων.

Σημείωση σχετικά με τα αποτελέσματα

Τα περισσότερα αποτελέσματα των queries παρουσιάζονται σε μορφή screenshot του output του notebook. Ωστόσο παρατηρήθηκε ότι υπήρχαν αρκετές αποκλίσεις μεταξύ των χρόνων διαδοχικών εκτελέσεων του ίδιου query ακόμα και αν βάζαμε στα dataframes .unpersist(). Για αυτόν τον λόγο κάθε query εκτελούνταν μετά από restart του kernel.

Ερωτήματα

Query 1

Να ταξινομηθούν, σε φθίνουσα σειρά, οι ηλικιακές ομάδες των θυμάτων σε περιστατικά που περιλαμβάνουν οποιαδήποτε μορφή “βαριάς σωματικής βλάβης”. Θεωρείστε τις εξής ηλικιακές ομάδες:

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

Υλοποιούμε το παραπάνω query χρησημοποιώντας DataFrame και RDD APIs. Εκτελούμε και τις 2 υλοποιήσεις με 4 Spark executors και εξετάζουμε την διαφορά στην επίδοση μεταξύ των δύο APIs.

Για να δημιουργήσουμε τα 4 spark executors εισάγουμε το παρακάτω configuration

```
%%configure -f
{
    "conf": {
        "spark.executor.instances": "4",
        "spark.executor.memory": "8g",
        "spark.executor.cores": "1",
        "spark.driver.memory": "2g"
    }
}
```

και ελέγχουμε ότι το Spark session έχει αυτές τις τιμές με τον παρακάτω κώδικα

```
from pyspark.sql import SparkSession

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

# Access configuration
conf = spark.sparkContext.getConf()

# Print relevant executor settings
print("Executor Instances:", conf.get("spark.executor.instances"))
print("Executor Cores:", conf.get("spark.executor.cores"))
print("Executor Memory:", conf.get("spark.executor.memory"))
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	User	Current session?
3678	application_1732639283265_3624	pyspark	idle	Link	Link	None	✔

```
SparkSession available as 'spark'.
Executor Instances: 4
Executor Cores: 1
Executor Memory: 8g
```

Για να υλοποιήσουμε το Query 1 με το RDD API θα πρέπει να δημιουργήσουμε ένα sparkcontext αντικείμενο μέσω του οποίου θα διαβάσουμε τα αρχεία με τα crime data σε 2 RDD τα οποία κάνουμε union για να έχουμε ένα ενιαίο RDD με τα δεδομένα. Ομοίως για να υλοποιήσουμε το Query 1 με το Dataframe API θα δημιουργήσουμε ένα αντικείμενο spark μέσω του οποίου θα διαβάσουμε τα crime data.

Η εκτέλεση των queries παράγει τα εξείς outputs :

```
[('Adults (25-64)', 121093), ('Young Adults (18-24)', 33605), ('Children (<18)', 15928), ('Elderly (>64)', 5985)]
[RDD API] Time taken: 20.27 seconds
```

```
+------------+------+
|   Age Group| Count|
+------------+------+
|      Adults|121093|
|Young Adults| 33605|
|    Children| 15928|
|     Elderly|  5985|
+------------+------+

[Dataframe API] Time taken: 9.04 seconds
```

Παρατηρούμε ότι η υλοποίηση που χρησιμοποιεί το Dataframe API είναι αρκετά πιο γρήγορη από αυτήν που χρησιμοποιεί το RDD API. Το RDD API είναι ένα unstructured low-level API το οποίο ουσιαστικά χρησιμοποιείται για να πούμε στο spark το “πώς” να κάνει κάτι. Αυτό όμως σημαίνει ότι δεν γίνεται να δεχτεί optimizations αυτόματα από το Spark ούτε να εκμεταλλευτεί τα optimizations του SQL engine. Αντίθετα το high-level DataFrame API χρησιμοποιείται για να εκφράσουμε ποιο είναι το επιθυμητό μας αποτέλεσμα. Έτσι το spark μπορεί αυτόματα να κατασκευάσει την πιο αποδοτική “συνταγή" για να παράξει το επιθυμητό αποτέλεσμα μέσω του Catalyst Optimizer.

Query 2

Να βρεθούν, για κάθε έτος, τα 3 Αστυνομικά Τμήματα με το υψηλότερο ποσοστό κλεισμένων (περατωμένων) υποθέσεων. Να τυπωθούν το έτος, τα ονόματα (τοποθεσίες) των τμημάτων, τα ποσοστά τους καθώς και οι αριθμοί του ranking τους στην ετήσια κατάταξη. Τα αποτελέσματα να δοθούν σε σειρά αύξουσα ως προς το έτος και το ranking.

α) Υλοποιούμε το Query 2 χρησιμοποιώντας τα DataFrame και SQL APIs και συγκρίνουμε τους χρόνους εκτέλεσης των δύο υλοποιήσεων.

Το SQL API είναι μέρος του Dataframe API που μας επιτρέπει αντί να παράγουμε Dataframe περιγράφοντάς τα με το Dataframe API να το τροφοδοτούμε με SQL queries κατευθείαν.

Η εκτέλεση των queries παράγει τα εξείς outputs :

```
+----+-----------+-------------------+----+
|Year|  AREA NAME| Percentage_Closed|rank|
+----+-----------+-------------------+----+
|2010|    Rampart| 32.84713448949121|   1|
|2010|    Olympic|31.515289821999087|   2|
|2010|     Harbor| 29.36028339237341|   3|
|2011|    Olympic|35.040060090135206|   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|31.224939855653567|   3|
|2015|   Van Nuys|32.265140677157845|   1|
|2015|    Mission|30.463762673676303|   2|
|2015|   Foothill|30.353001803658852|   3|
|2016|   Van Nuys|32.194518462124094|   1|
|2016|West Valley| 31.40146437042384|   2|
+----+-----------+-------------------+----+
only showing top 20 rows

[Dataframe API] Time taken: 6.87 seconds
```

```
+----+-----------+-----------------+----+
|Year|  AREA_NAME| PercentageClosed|Rank|
+----+-----------+-----------------+----+
|2010|    Rampart|32.94735585531813|   1|
|2010|    Olympic|31.96270619172842|   2|
|2010|     Harbor|29.63203463203463|   3|
|2011|    Olympic|35.21216768916155|   1|
|2011|    Rampart|32.51177963030083|   2|
|2011|     Harbor|28.65220520201501|   3|
|2012|    Olympic|34.41481831052383|   1|
|2012|    Rampart|32.94641810294290|   2|
|2012|     Harbor|29.81513327601032|   3|
|2013|    Olympic|33.52812271731191|   1|
|2013|    Rampart|32.08287360549222|   2|
|2013|     Harbor|29.16422459266206|   3|
|2014|   Van Nuys|31.80567315834039|   1|
|2014|West Valley|31.31198995605775|   2|
|2014|    Mission|31.16279069767442|   3|
|2015|   Van Nuys|32.64134698172773|   1|
|2015|West Valley|30.27597402597403|   2|
|2015|    Mission|30.17946067838016|   3|
|2016|   Van Nuys|31.88075572011773|   1|
|2016|West Valley|31.54798761609907|   2|
+----+-----------+-----------------+----+
only showing top 20 rows

[SQL API] Time taken: 6.51 seconds
```

Παρατηρούμε ότι οι χρόνοι είναι πολύ κοντά μεταξύ τους. Αυτό οφείλεται στο γεγονός ότι και οι δύο υλοποιήσεις πρακτικά βασίζονται πάνω στον Catalyst optimizer και το SQL engine του spark. Εφόσον λοιπόν ζητάμε και στις δύο περιπτώσεις να υπολογιστεί το ίδιο πράγμα, παράγονται πανομοιότητα execution plans πάνω στα ίδια δεδομένα. (Τα πλάνα που παράγονται με .explain() παρουσιάζονται παρακάτω για να γίνει εμφανής η ομοιότητα τους)

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [Year#12307 ASC NULLS FIRST, rank#12411 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(Year#12307 ASC NULLS FIRST, rank#12411 ASC NULLS FIRST, 1000), ENSURE_REQUIREMENTS, [plan_id=1969]
      +- Filter (rank#12411 <= 3)
         +- Window [dense_rank(Percentage_Closed#12403) windowspecdefinition(Year#12307, Percentage_Closed#12403 DESC NULLS LAST, specifiedwindowframe(RowFrame, unboundedpreceding$(), currentrow$())) AS ra
sed#12403 DESC NULLS LAST]
            +- WindowGroupLimit [Year#12307], [Percentage_Closed#12403 DESC NULLS LAST], dense_rank(Percentage_Closed#12403), 3, Final
               +- Sort [Year#12307 ASC NULLS FIRST, Percentage_Closed#12403 DESC NULLS LAST], false, 0
                  +- Exchange hashpartitioning(Year#12307, 1000), ENSURE_REQUIREMENTS, [plan_id=1964]
                     +- WindowGroupLimit [Year#12307], [Percentage_Closed#12403 DESC NULLS LAST], dense_rank(Percentage_Closed#12403), 3, Partial
                        +- Sort [Year#12307 ASC NULLS FIRST, Percentage_Closed#12403 DESC NULLS LAST], false, 0
                           +- HashAggregate(keys=[Year#12307, AREA NAME#12142], functions=[sum(CASE WHEN (NOT (Status#12155 = IC) AND NOT (Status#12155 = UNK)) THEN 1 ELSE 0 END), count(DR_NO#12137)], sche
                              +- Exchange hashpartitioning(Year#12307, AREA NAME#12142, 1000), ENSURE_REQUIREMENTS, [plan_id=1960]
                                 +- AdaptiveStatsCollector Year#12307 (distinctCount: ?)
                                    +- HashAggregate(keys=[Year#12307, AREA NAME#12142], functions=[partial_sum(CASE WHEN (NOT (Status#12155 = IC) AND NOT (Status#12155 = UNK)) THEN 1 ELSE 0 END), partial_
                                       +- Union
                                          :- Project [DR_NO#12137, AREA NAME#12142, Status#12155, year(cast(gettimestamp(DATE OCC#12139, MM/dd/yyyy hh:mm:ss a, TimestampType, Some(UTC), false) as date)) AS
                                          :  +- FileScan csv [DR_NO#12137,DATE OCC#12139,AREA NAME#12142,Status#12155] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[s3:
418150721/CrimeData/Crime_D..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DR_NO:string,DATE OCC:string,AREA NAME:string,Status:string>
                                          +- Project [DR_NO#12193, AREA NAME#12198, Status#12211, year(cast(gettimestamp(DATE OCC#12195, MM/dd/yyyy hh:mm:ss a, TimestampType, Some(UTC), false) as date)) AS
                                             +- FileScan csv [DR_NO#12193,DATE OCC#12195,AREA NAME#12198,Status#12211] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[s3:
418150721/CrimeData/Crime_D..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DR_NO:string,DATE OCC:string,AREA NAME:string,Status:string>


== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [Year#10380 ASC NULLS FIRST, Rank#10384 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(Year#10380 ASC NULLS FIRST, Rank#10384 ASC NULLS FIRST, 1000), ENSURE_REQUIREMENTS, [plan_id=2075]
      +- Filter (Rank#10384 <= 3)
         +- Window [rank(PercentageClosed#10383) windowspecdefinition(Year#10380, PercentageClosed#10383 DESC NULLS LAST, specifiedwindowframe(RowFrame, unboundedpreceding$(), currentrow$())) AS Rank#1038
DESC NULLS LAST]
            +- WindowGroupLimit [Year#10380], [PercentageClosed#10383 DESC NULLS LAST], rank(PercentageClosed#10383), 3, Final
               +- Sort [Year#10380 ASC NULLS FIRST, PercentageClosed#10383 DESC NULLS LAST], false, 0
                  +- Exchange hashpartitioning(Year#10380, 1000), ENSURE_REQUIREMENTS, [plan_id=2070]
                     +- WindowGroupLimit [Year#10380], [PercentageClosed#10383 DESC NULLS LAST], rank(PercentageClosed#10383), 3, Partial
                        +- Sort [Year#10380 ASC NULLS FIRST, PercentageClosed#10383 DESC NULLS LAST], false, 0
                           +- HashAggregate(keys=[Year#10380, AREA_NAME#10350], functions=[count(CASE WHEN NOT Status#10139 IN (IC,UNK) THEN 1 END), count(1)], schema specialized)
                              +- Exchange hashpartitioning(Year#10380, AREA_NAME#10350, 1000), ENSURE_REQUIREMENTS, [plan_id=2066]
                                 +- AdaptiveStatsCollector Year#10380 (distinctCount: ?)
                                    +- HashAggregate(keys=[Year#10380, AREA_NAME#10350], functions=[partial_count(CASE WHEN NOT Status#10139 IN (IC,UNK) THEN 1 END), partial_count(1)], schema specialized)
                                       +- Union
                                          :- Project [AREA NAME#10126 AS AREA_NAME#10350, Status#10139, substring_index(substring_index(Date Rptd#10122,  , 1), /, -1) AS Year#10380]
                                          :  +- FileScan csv [Date Rptd#10122,AREA NAME#10126,Status#10139] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[s3://initial-r
rimeData/Crime_D..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Date Rptd:string,AREA NAME:string,Status:string>
                                          +- Project [AREA NAME#10182 AS AREA_NAME#14253, Status#10195, substring_index(substring_index(Date Rptd#10178,  , 1), /, -1) AS Year#14254]
                                             +- FileScan csv [Date Rptd#10178,AREA NAME#10182,Status#10195] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[s3://initial-r
rimeData/Crime_D..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Date Rptd:string,AREA NAME:string,Status:string>
```

β) Γράφουμε κώδικα Spark που μετατρέπει το κυρίως dataset σε .parquet και αποθηκεύει ένα μοναδικό .parquet αρχείο στο S3 bucket της ομάδας μας.

Στην συνέχεια επιλέγουμε την υλοποιηση του υπο ερωτήματος α) με το SQL API και συγκρίνουμε τους χρόνους εκτέλεσης της εφαρμογής μας όταν τα δεδομένα εισάγονται σαν .csv και σαν .parquet.

```
+----+-----------+-----------------+----+
|Year|  AREA_NAME| PercentageClosed|Rank|
+----+-----------+-----------------+----+
|2010|    Rampart|32.94735585531813|   1|
|2010|    Olympic|31.96270619172842|   2|
|2010|     Harbor|29.63203463203463|   3|
|2011|    Olympic|35.21216768916155|   1|
|2011|    Rampart|32.51177963030083|   2|
|2011|     Harbor|28.65220520201501|   3|
|2012|    Olympic|34.41481831052383|   1|
|2012|    Rampart|32.94641810294290|   2|
|2012|     Harbor|29.81513327601032|   3|
|2013|    Olympic|33.52812271731191|   1|
|2013|    Rampart|32.08287360549222|   2|
|2013|     Harbor|29.16422459266206|   3|
|2014|   Van Nuys|31.80567315834039|   1|
|2014|West Valley|31.31198995605775|   2|
|2014|    Mission|31.16279069767442|   3|
|2015|   Van Nuys|32.64134698172773|   1|
|2015|West Valley|30.27597402597403|   2|
|2015|    Mission|30.17946067838016|   3|
|2016|   Van Nuys|31.88075572011773|   1|
|2016|West Valley|31.54798761609907|   2|
+----+-----------+-----------------+----+
only showing top 20 rows

[SQL .parquet] Time taken: 1.93 seconds
```

```
+----+-----------+-----------------+----+
|Year|  AREA_NAME| PercentageClosed|Rank|
+----+-----------+-----------------+----+
|2010|    Rampart|32.94735585531813|   1|
|2010|    Olympic|31.96270619172842|   2|
|2010|     Harbor|29.63203463203463|   3|
|2011|    Olympic|35.21216768916155|   1|
|2011|    Rampart|32.51177963030083|   2|
|2011|     Harbor|28.65220520201501|   3|
|2012|    Olympic|34.41481831052383|   1|
|2012|    Rampart|32.94641810294290|   2|
|2012|     Harbor|29.81513327601032|   3|
|2013|    Olympic|33.52812271731191|   1|
|2013|    Rampart|32.08287360549222|   2|
|2013|     Harbor|29.16422459266206|   3|
|2014|   Van Nuys|31.80567315834039|   1|
|2014|West Valley|31.31198995605775|   2|
|2014|    Mission|31.16279069767442|   3|
|2015|   Van Nuys|32.64134698172773|   1|
|2015|West Valley|30.27597402597403|   2|
|2015|    Mission|30.17946067838016|   3|
|2016|   Van Nuys|31.88075572011773|   1|
|2016|West Valley|31.54798761609907|   2|
+----+-----------+-----------------+----+
only showing top 20 rows

[SQL API] Time taken: 6.51 seconds
```

Όπως παρατηρούμε η χρήση του parquet για την εισαγωγή δεδομένων επιταχύνει σημαντικά την εκτέλεση του query. Αυτό οφείλεται στο μικρότερο μέγεθος του αρχείου parquet (λόγο του compression που χρησιμοποιεί) πράγμα που σημαίνει ότι και θα φορτωθεί γρηγορότερα και θα καταλάβει λιγότερο μέρος της μνήμης, καθώς και στις πληροφορίες που περιέχει το parquet για το schema που του επιτρέπουν να κάνει Predictive Pushdown δηλαδή να εφαρμώσει τα filters στο dataframe καθώς αυτό φορτώνεται στην μνήμη.

Query 3

Χρησιμοποιώντας ως αναφορά τα δεδομένα της απογραφής 2010 για τον πληθυσμό και εκείνα της απογραφής του 2015 για το εισόδημα ανα νοικοκυριό, υπολογίζουμε για κάθε περιοχή του Los Angeles τα παρακάτω: Το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο. Τα αποτελέσματα να συγκεντρώνονται σε ένα πίνακα.

Υλοποιούμε το Query 3 χρησιμοποιώντας DataFrame API. Χρησιμοποιούμε την μέθοδο explain για να βρούμε ποιες στρατηγικές join χρησιμοποιεί ο catalyst optimizer. Πειραματιζόμαστε αναγκάζοντας το Spark να χρησιμοποιήσει διαφορετικές στρατηγικές (μεταξύ των BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL) μέσω της μεθόδου hint.

Παρακάτω βλέπουμε τις στρατηγικές που επιλέγει ο Catalyst Optimizer για καθένα από τα joins με την σειρά που εμφανίζονται στο .explain του τελικού dataframe.

- +- SortMergeJoin [COMM#9431], [COMM#9918], Inner

Αυτό το join είναι το τελευταίο join που εκτελούμε στον κώδικα για να συνδυάσουμε τα dataframes για το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο. Βλεπουμε στο πλάνο ότι για να εκτελέσει αυτό το join χρειάζεται να κάνει πρώτα Sort τους πίνακες πράγμα που υποψιαζόμαστε ότι προσθέτει overhead. Παρόλα αυτά καμία από τις άλλες μεθόδους δεν επιταχύνει την εκτέλεση. Αυτό οφείλεται στο γεγονός ότι τα dataframes που κάνει join είναι μικρά (139 tuples με 2-4 columns το καθένα)

- +- BroadcastHashJoin [ZCTA10#9448], [ZCTA10#9552], Inner, BuildRight, false

Σε αυτό το join συνδυάζουμε το dataframe με τα incomes ανά zip code με τα οικοδομικά block. Το να το αλλάξουμε σε SortMergeJoin με hint merge τριπλασιάζει τον χρόνο εκτέλεσης λόγω του overhead που προσθέτει το sort. Το να το αλλάξουμε σε shuffleHashJoin με hint SHUFFLE_HASH επίσης τριπλασιάζει τον χρόνο εκτέλεσης όπως και με hint SHUFFLE_REPLICATE_NL όπου λογικά επειδή οι πίνακες δεν χωράνε στους executors το αυξημένο IO που απαιτείται καθυστερεί την εκτέλεση.

- +- RangeJoin geometry#9787: geometry, point#9375: geometry, CONTAINS

Αυτό είναι το join που ελέγχει σε ποια περιοχή ανήκει κάθε έγκλημα
Σε αυτό το join η μόνη άλλη παραλλαγή που μπορούμε να προκαλέσουμε (επειδή είναι non equi join) είναι η παρακάτω με .hint("BROADCAST")
+- BroadcastIndexJoin point#14272: geometry, LeftSide, LeftSide, Inner, CONTAINS
ST_CONTAINS(geometry#14684, point#14272)
Αυτή η αλλαγή επιβραδύνει σχετικά την εκτέλεση (μερικά δευτερόλεπτα)

Γενικά παρατηρούμε ότι ο Catalyst Optimizer επιλέγει καλά την τεχνική που χρησιμοποιεί

Τα αποτελέσματα της εκτέλεσης (χωρίς hints) φαίνονται παρακάτω

```
+------------------+------------+---------------+--------------------+-----------+--------------------+
|              COMM|total_income|total_population|avg_income_per_capita|crime_count|avg_crime_per_capita|
+------------------+------------+---------------+--------------------+-----------+--------------------+
|  Pacific Palisades|  1458554137|          20643|    70656.11282274863|       9307| 0.45085501138400425|
|Palisades Highlands|   256302897|           3833|    66867.43986433603|        788|  0.2055830941821028|
|   Marina Peninsula|   282927205|           4337|    65235.69402813004|       2656|  0.6124048881715471|
|            Bel Air|   520784494|           8261|    63041.33809466166|       3532|  0.4275511439293064|
|      Beverly Crest|   743010853|          12191|    60947.49019768682|       4527|  0.3713395127553113|
|          Brentwood|  1782691149|          29301|   60840.624859219824|      14756|  0.5036005597078598|
|  Mandeville Canyon|   179664630|           3233|    55572.10949582431|        848| 0.26229508196721313|
|        Playa Vista|   448660676|           8926|    50264.47187990141|       7281|  0.8157069235939951|
|            Carthay|   656158931|          13165|   49841.164527155335|      12273|  0.9322445879225219|
|             Venice|  1553435569|          32625|   47614.883340996166|      40942|  1.2549272030651342|
|       Century City|   548170741|          11890|   46103.510597140456|      10176|  0.8558452481076535|
|      Playa Del Rey|   143760360|           3158|      45522.596580114|       2454|  0.7770740975300824|
|        Studio City|   911964099|          20703|    44049.85263005362|      19245|  0.9295754238516157|
|    Hollywood Hills|  1199930991|          27895|   43015.988205771646|      22669|  0.8126545975981359|
|      South Carthay|   402017715|          10093|   39831.340037649854|       7299|   0.723174477360547|
|   West Los Angeles|  1356828991|          34165|    39714.00529781941|      28397|  0.8311722523049905|
|       Miracle Mile|   623571124|          16057|    38834.84611073052|      14163|  0.8820452139253908|
|        Rancho Park|   243868702|           6295|   38740.063860206516|       7902|  1.2552819698173154|
|             Encino|  1623576201|          42349|    38338.00564358072|      29803|   0.703747432052705|
|       Sherman Oaks|  3075894078|          81443|   37767.445673661336|      64370|  0.7903687241383545|
+------------------+------------+---------------+--------------------+-----------+--------------------+
only showing top 20 rows

Time taken: 25.50 seconds
```

Query 4

Αναζητούμε το φυλετικό προφίλ των καταγεγραμμένων θυμάτων εγκλημάτων (Vict Descent) στο Los Angeles για το έτος 2015 στις 3 περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα (Οι οποίες είναι οι Pacific Palisades, Palisades Highlands, Marina Peninsula όπως είδαμε στο προηγούμενο ερώτημα). Κάνουμε το ίδιο για τις 3 περιοχές με το χαμηλότερο εισόδημα (Οι οποίες είναι οι Rancho Park, Encino, Sherman Oaks).

Χρησιμοποιούμε την αντιστοίχιση των κωδικών καταγωγής με την πλήρη περιγραφή από το σύνολο δεδομενων Race and Ethnicity codes.

Υλοποιούμε το Query 4 χρησιμοποιώντας το DataFrame API. Να εκτελούμε την υλοποίησή μας εφαρμόζοντας κλιμάκωση στο σύνολο των υπολογιστικών πόρων που θα χρησιμοποιούμε. Συγκεκριμένα, καλούμαστε να εκτελέσουμε την υλοποίησή μας σε 2 executors με τα ακόλουθα configurations:

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

Με 1 core/2 GB memory λαμβάνουμε το εξείς output :

```
Richest 3 Areas
+--------------------+-----+
|   Vict Descent Full|count|
+--------------------+-----+
|               White|  695|
|               Other|   86|
|Hispanic/Latin/Me...|   77|
+--------------------+-----+
only showing top 3 rows

Poorest 3 Areas
+--------------------+-----+
|   Vict Descent Full|count|
+--------------------+-----+
|               White| 3497|
|               Other| 1245|
|Hispanic/Latin/Me...|  904|
+--------------------+-----+
only showing top 3 rows

Time taken: 69.91 seconds
```

Με 2 cores/4 GB memory λαμβάνουμε το ίδιο output με χρόνο:

```
Time taken: 55.55 seconds
```

Με 4 cores/8GB memory λαμβάνουμε το ίδιο output με χρόνο:

```
Time taken: 36.04 seconds
```

Παρατηρούμε ότι όπως και ήταν αναμενόμενο, το speedup δεν είναι γραμμικό. Κάθε φορά που διπλασιάζουμε τα cores και την μνήμη ο χρόνος μειώνεται κατά 20-25%
Αυτό οφείλεται στο ότι υπάρχει σειριακό κομμάτι στην εκτέλεση (Amdahl's law) και λειτουργούμε με πολλά δεδομένα κάνοντας λίγες πράξεις σε καθένα από αυτά (memory bound εφαρμογή) άρα καθυστερούμε αρκετά από την μεταφορά δεδομένων από την μνήμη στην cache (ίσως και από την μεταφορά από τον δίσκο στην μνήμη αν τα δεδομένα υπερχειλίζουν την RAM και χρειάζεται να μπουν σε SWAP).

Query 5

Υπολογίζουμε, ανά αστυνομικό τμήμα, τον αριθμό εγκλημάτων που έλαβαν χώρα πλησιέστερα σε αυτό, καθώς και την μέση απόστασή του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά. Τα αποτελέσματα είναι ταξινομημένα κατά αριθμό περιστατικών, με φθίνουσα σειρά

Να υλοποιηθεί το Query 5 χρησιμοποιώντας το DataFrame ή SQL API. Να εκτελέσετε την υλοποίησή σας χρησιμοποιώντας συνολικούς πόρους 8 cores και 16GB μνήμης με τα παρακάτω configurations:

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

Για 2 executors × 4 cores/8GB memory λαμβάνουμε το εξείς output :

```
+----------------+------------------+------+
|        DIVISION|  average distance|     #|
+----------------+------------------+------+
|       HOLLYWOOD| 2.076263960178721|224340|
|        VAN NUYS|2.9533697428197865|210134|
|       SOUTHWEST| 2.191398805780884|188901|
|        WILSHIRE| 2.592665532978779|185996|
|     77TH STREET| 1.716544971970102|171827|
|         OLYMPIC|1.7236036971780937|170897|
| NORTH HOLLYWOOD| 2.643006094141567|167854|
|         PACIFIC| 3.850070655307902|161359|
|         CENTRAL|0.9924764374568898|153871|
|         RAMPART|1.5345341879190049|152736|
|       SOUTHEAST| 2.421866215888179|152176|
|     WEST VALLEY|  3.03567121631408|138643|
|         TOPANGA|3.2969548417555608|138217|
|        FOOTHILL|  4.25092170842499|134896|
|          HARBOR|3.7025615993565038|126747|
|      HOLLENBECK|2.6801812377068233|115837|
|WEST LOS ANGELES| 2.792457289034109|115781|
|          NEWTON|1.6346357397097429|111110|
|       NORTHEAST| 3.623665524604075|108109|
|         MISSION|3.6909426142786046|103355|
|      DEVONSHIRE| 2.824765412800824| 77094|
+----------------+------------------+------+

Time taken: 60.60 seconds
```

Για 4 executors × 2 cores/4GB memory λαμβάνουμε το ίδιο output με χρόνο :

```
Time taken: 55.80 seconds
```

Για 8 executors × 1 core/2 GB memory λαμβάνουμε το ίδιο output με χρόνο :

```
Time taken: 73.28 seconds
```

Παρατηρούμε ότι ενώ ο συνολικός αριθμός των Core και το μέγεθος της RAM που δεσμεύουμε κάθε φορά κρατιέται σταθερό ο χρόνος εκτέλεσης στην εκτέλεση με 4 executors × 2 cores/4GB memory μειώνεται ελαφρά, πιθανώς επειδή τα διαφορετικά executors “παλεύουν” λιγότερο για πόρους όπως το bandwidth του δίσκου.

Αντίθετα στην περίπτωση με 8 executors × 1 core/2 GB memory ο χρόνος εκτέλεσης αυξάνεται. Αυτό εικάζουμε ότι συμβαίνει επειδή δεν χωράνε τα δεδομένα στην RAM κάθε executor με αποτέλεσμα να χρειάζεται να ανταλλαχθούν δεδομένα με τον δίσκο.