# Εξαμηνιαία Εργασία στο Μάθημα των Προχωρημένων Θεμάτων Βάσεων Δεδομένων

Χειμερινο Εξαμηνο 2025-2026

Ομαδα 12

Ρεϊζης Εμμανουηλ
ΑΜ: 03121067

Σκοτης Αποστολος
ΑΜ: 03121212

## Εισαγωγή

Στην παρούσα εργασία του μαθήματος των Προχωρημένων Θεμάτων Βάσεων Δεδομένων θα ασχοληθούμε με την επεξεργασία δεδομένων μεγάλου όγκου καθώς και την διαχείρισή τους μέσω ενός κατανεμημένου συστήματος. Συγκεκριμένα, θα χρησιμοποιήσουμε την πλατφόρμα AWS της Amazon για να γράψουμε κώδικα και να επικοινωνήσουμε με ένα απομακρυσμένο σύστημα Apache Spark, μέσω το οποίου θα εκτελέσουμε queries πάνω στα σύνολα δεδομένων που διαθέτουμε. Επιπλέον, θα έρθουμε να πειραματιστούμε με διαφορετικά configurations πόρων για την εκτέλεση των ερωτημάτων ενώ θα αξιολογήσουμε τα πλάνα εκτέλεσης που επιλέγει ο Catalyst optimizer για τον υπολογισμό των queries.

Για τα πέντε queries που μας ζητείται να υλοποιήσουμε θα γράψουμε κώδικα PySpark, δηλαδή python με βιβλιοθήκες επέκτασης για υποστήριξη επεξεργασίας Spark RDDs και DataFrames. Τα προγράμματα που έχουμε γράψει βρίσκονται σε ένα ξεχωριστό notebook για το κάθε query και βρίσκονται ανεβασμένα στο github repo που βρίσκεται στο ακόλουθο **link**.

## Datasets

Το βασικό dataset πάνω στο οποίο θα εκπονηθεί η εργασία είναι το Los Angeles Crime Dataset, στο οποίο περιλαμβάνονται συσσωρευμένα δεδομένα για εγκλήματα που συνέβησαν στην πόλη του Los Angeles στο διάστημα 2010 έως σήμερα. Πληροφορίες που περιλαμβάνονται σε αυτό είναι ο χώρος και χρόνος διεξαγωγής του εγκλήματος, κωδικοί ετικετών που περιγράφουν το συμβάν, ηλικία και φύλο του θύματος κλπ. Συμπληρωματικά σύνολα δεδομένων είναι:

- Το Census Blocks που περιλαμβάνει απογραφικά στοιχεία του LA ανά οικοδομικό τετράγωνο για το έτος 2020
- Το Median Household Income by Zip Code που περιέχει μέσο ετήσιο εισόδημα ανά νοικοκυριό για κάθε περιοχή για το έτος 2021
- Το LA Police Stations που περιέχει πληροφορία για το όνομα και την τοποθεσία 21 αστυνομικών σταθμών στο Los Angeles

Επιπλέον, περιλαμβάνονται συμπληρωματικά σύνολα δεδομένων, όπως το Race and Ethnicity codes και MO Codes που δίνουν πιο αναλυτικές περιγραφές για attributes των πρότερων dataset. Εμείς καλούμαστε να συνδυάσουμε την πληροφορία όλων αυτών για απαντήσουμε στα επόμενα ερωτήματα.

## Query 1

Για το πρώτο Query καλούμαστε να το υλοποιήσουμε χρησιμοποιώντας:

- RDD API
- Dataframe API (με UDF)
- Dataframe API (χωρίς UDF)

Για το configuration του Spark Cluster στα Queries 1, 2 και 3 χρησιμοποιούμε 4 executors με τον καθένα να διαθέτει 1 core και 2GB μνήμης. Ως Executor ορίζεται το process στα worker nodes που είναι υπεύθυνο για την εκτέλεση των tasks. Το Core αντιστοιχεί στη μονάδα επεξεργασίας που καθορίζει πόσα tasks μπορούν να εκτελεστούν παράλληλα εντός του executor, ενώ η Μνήμη (RAM) είναι ο χώρος που δεσμεύεται για την αποθήκευση (caching) και την επεξεργασία των δεδομένων.

### RDD Implementation

Για το RDD αρχικά διαβάζουμε απο τα CSV τα crime data για τις δύο περιόδους και τα ενώνουμε σε ένα RDD. Μετά κάνουμε filter() ώστε να κρατήσουμε μόνο τα rows όπου το column "Crime Cd Desc" (x[9]) περιέχει το substring "AGGRAVATED ASSAULT". Μετά ορίζουμε μια συνάρτηση, η οποία παίρνει ενα row, εξάγει το column "Vict Age" (x[11]) και βάσει αυτού επιστρέφει ενα row [ηλικιακή ομάδα του θύματος, 1]. Αυτό το function το κάνουμε map() στο RDD μας και κάνουμε reduceByKey() με κλειδί την ηλικιακή ομάδα και ανώνυμη συνάρτηση το άθροισμα, οπότε παίρνουμε το συνολικό άθροισμα θυμάτων σε κάθε ηλικιακή ομάδα. Τέλος, κάνουμε sortBy() το πλήθος των θυμάτων και collect() ώστε να πάρουμε τα αποτελέσματα τα οποία παρουσιάζονται (μαζί με τον χρόνο εκτέλεσης [1][2]) παρακάτω:

[('adult', 121660), ('young adult', 33758), ('child', 16014), ('senior', 6011)]
Execution Time: 37.659892082214355 seconds

### Dataframe Implementation - with UDF

Για την υλοποίηση αυτή φτιάχνουμε αρχικά ένα σχήμα του dataframe που θα φορτώσουμε αντίστοιχο με το crime dataset και μετά φορτώνουμε τα CSV και ενώνουμε σε ένα dataframe. Όπως και πριν κάνουμε filter ώστε να κρατήσουμε μόνο τα rows όπου το column "Crime Cd Desc" περιέχει το substring "AGGRAVATED ASSAULT" και ορίζουμε την ίδια User Defined Function με πριν, με την διαφορά ότι τώρα επιστρέφει μόνο την ηλικιακή ομάδα ως string. Αφού την ορίσουμε με το udf(fun, return_type), την εφαρμόζουμε σε κάθε column με την βοήθεια της withColumn(), η οποία προσθέτει ένα επιπλεόν column "age group" με το αποτέλεσμα της συνάρτησή μας σε κάθε row. Τέλος, κάνουμε groupBy() με το column αυτό, count() για να πάρουμε το πλήθος των θυμάτων και sort() τον πίνακα με βάση το "count" column. Τα τελικά αποτελέσματα (μαζί με τον χρόνο εκτέλεσης) παρουσιάζονται παρακάτω:

[1] Οι χρόνοι εκτέλεσης αυτού αλλά και όλων των υπόλοιπων ερωτημάτων περιλαμβάνουν και τον χρόνο φόρτωσης των δεδομένων από τα csv's.

[2] Πριν την εκτέλεση κάθε παραλλαγής του κάθε ερωτήματος (αυτού και των επόμενων) κάνουμε restart το spark session ώστε να πάρουμε ακριβή αποτελέσματα στους χρόνους.

```
+-----------+------+
|  age group| count|
+-----------+------+
|      adult|121660|
|young adult| 33758|
|      child| 16014|
|     senior|  6011|
+-----------+------+
```

Execution Time: 15.84756088256836 seconds

### Dataframe Implementation - without UDF

Η υλοποίηση είναι αντίστοιχη με την προηγούμενη μεχρι το filtering. Μετά, αντί για UDF, χρησιμοποιούμε το withColumn με διαδοχικά when() και otherwise() τα οποία κάθε φορά ελέγχουν το column "Vict Age" και επιστρέφουν column με την αντίστοιχη ηλικιακή ομάδα. Τέλος, κάνουμε groupBy() με το column αυτό, count() για να πάρουμε ο πλήθος και κάνουμε και sort() τον πίνακα με βάσει το "count" column. Τα τελικά αποτελέσματα (μαζί με τον χρόνο εκτέλεσης) παρουσιάζονται παρακάτω:

```
+-----------+------+
|  age group| count|
+-----------+------+
|      adult|121660|
|young adult| 33758|
|      child| 16014|
|     senior|  6011|
+-----------+------+
```

Execution Time: 13.86941146850586 seconds

## Συμπεράσματα

Το πρώτο και πιο σημαντικό πόρισμα είναι ότι όλες οι υλοποιήσεις κατέληξαν στο ίδιο αποτέλεσμα, οπότε είναι όλες ισοδύναμες (ως προς την ορθότητα τους).

| **Υλοποίηση** | **Χρόνος Εκτέλεσης (sec)** |
|---|---|
| RDD | 37.66 |
| DataFrame (with UDF) | 15.85 |
| DataFrame (without UDF) | 13.87 |

Table 1: Σύγκριση Χρόνων Εκτέλεσης Query 1

Η κύρια διαφορά έγκειται στους χρόνους εκτέλεσης των υλοποιήσεων αυτών. Η πιο αργή υλοποίηση ήταν προφανώς η RDD, καθώς γνωρίζουμε ότι το API αυτό είναι πολύ low level και είναι ευθύνη του προγραμματιστή να φτιάξει ένα efficient transformation chain. Το Spark το ίδιο δεν μπορεί να κάνει optimizations (όπως και γίνεται στην περίπτωση των Dataframes) και ακόμη η υλοποίηση αυτή είναι αργή σε non-JVM γλώσσες προγραμματισμού, όπως η python που χρησιμοποιήσαμε εμείς.

Οι δύο Dataframe υλοποιήσεις είναι πολύ πιο γρήγορες απο το RDD όπως βλέπουμε από τους χρόνους πο παρουσιάσαμε παραπάνω. Όμως μεταξύ τους η υλοποίηση που χρησιμοποιεί UDF είναι πιο αργή απο αυτή που δεν χρησιμοποιεί. Ο λόγος είναι ότι το UDF αναγκαστικά πρέπει να βγεί απο το JVM και να εκτελεστεί σειριακά σε ένα python proccess και να επιστρέψει το αποτέλεσμα σειριακά στο JVM. Σε αντίθεση η υλοποίηση με τα when() και otherwise() τρέχει μέσα στο JVM και μπορεί να γίνει optimize από το Spark.

## Query 2

Στο δεύτερο Query καλούμαστε να χρησιμοποιήσουμε:

- Dataframe API + SQL
- Dataframe API

Το configuration του Spark Cluster είναι το ίδιο με το προηγούμενο query και κάνουμε restart το Spark Session μεταξύ queries για να έχουμε το πραγματικό (cold-start) performace του κάθε query.

### Dataframe API + SQL Implementation

Αρχικά, ορίζουμε το σχήμα για τα csv, φορτώνουμε τα δεδομένα και τα μετατρέπουμε σε **Temporary Views** "crimes" και "re_codes" ώστε να εκτελέσουμε SQL ερωτήματα. Εκτελούμε 4 διαδοχικά queries και αποθηκεύουμε το αποτέλεσμα κάθε φορά πάλι σε ένα Temporary View.

- **Query 0:** Προσθέτουμε ενα column "year", το οποίο έχει εξάγει τον χρόνο από το "DATE OCC" column στο "crimes" table.
- **Query 1:** Απο το "crimes" table κάνουμε GROUP BY year, "Vict Decent", αποκλείοντας τα NULL και κάνουμε COUNT(*) AS Total. Κρατάμε μόνο τα columns (year, 'Vict Descent', Total).
- **Query 2:** Κάνουμε PARTITON τα αποτελέσματα του παραπάνω πάνω στο year και κάνουμε sort με ORDER BY Total DESC. Απο αυτα κρτάμε τα top 3 Total και υπολογίζουμε τα ποσοστά με ROUND(100*(Total/sum),2) as '%'.
- **Query 3:** Τέλος, κάνουμε NATURAL JOIN (εφοσον το όνομα του κοινού column είναι το ίδιο) του προηγούμενου table με το "re_codes" table ώστε να πάρουμε ολόκληρο το όνομα του ethnicity. Κρατάμε μόνο τις στήλες (year, 'Vict Descent Full' AS 'Victim Descent', '#', '%') και κάνουμε sort με ORDER BY year DESC, '#' DESC.

Τα αποτελέσματα της υλοποίησης βρίσκονται στο εκτελεσμένο notebook καθώς ο πίνακας είναι πολύ μεγάλος για να παραθέσουμε εδώ. Ο χρόνος εκτέλεσης είναι:

Execution Time: 25.627448558807373 seconds

### Dataframe API

Για την υλοποίηση αυτή φορτώνουμε τα δεδομένα όπως πριν, αλλά τα κρατάμε στα dataframes αντί για Views. Με αυτά κάνουμε groupBy() με το year και Vict Descent columns και count() στο crime_data_df ώστε να πάρουμε το σύνολο των θυμάτων ανα χρόνο και εθνικότητα ενώ παράλληλα αφαιρούμε τις NULL τιμές. Μετά ορίζουμε δύο pyspark Windows: ένα το οποίο κάνει partitionBy("year") και μετά orderBy(desc("count")) και ένα το οποίο κάνει partitionBy("year") μόνο χωρίς ordering. Με αυτά μπορούμε να προσθεσουμε ένα column "rank" το οποίο έχει row_number().over() το πρώτο window και κάνει rank τις εθνικότητες μέσα σε ένα χρόνο ώστε μετά μπορούμε με ένα filter() να κρατήσουμε μόνο τις πρώτες 3 από κάθε χρόνο. Ακόμη προσθέτουμε ένα column "total_count" το οποίο κάνει sum("count").over() με το δεύτερο παράθυρο ώστε να έχουμε το σύνολο των θυμάτων κάθε

χρονο και να μπορούμε να παράγουμε τα ποσοστα σε ένα νέο column. Τέλος, μετονομάζουμε τα columns , κρατάμε μόνο αυτά που χρειάζονται και κάνουμε orderBy(desc("year"), desc("#")).

Τα αποτελέσματα της υλοποίησης βρίσκονται στο εκτελεσμένο notebook καθώς ο πίνακας είναι πολύ μεγάλος για να παραθέσουμε εδώ. Ο χρόνος εκτέλεσης είναι:

Execution Time: 24.2682204246521 seconds

### Συμπεράσματα

Οι δύο υλοποιήσεις κατέληξαν στο ίδιο αποτέλεσμα, οπότε είναι ισοδύναμες (ως προς την ορθότητα τους). Παρατηρούμε ότι οι χρόνοι εκτέλεσης είναι συγκρίσιμοι, με την SQL υλοποιήση να είναι πιο γρήγορη σε κάποιες εκτελέσεις και την Dataframe σε άλλες εκτελέσεις. Και στις δύο περιπτώσεις, ο **Catalyst Optimizer** του Spark μετατρέπει τον κώδικα (είτε SQL είτε DataFrame operations) σε βελτιστοποιημένο φυσικό πλάνο εκτέλεσης, επομένως οι διαφορές στην απόδοση είναι αμελητέες και αφορούν κυρίως το overhead του parsing και της ανάλυσης του πλάνου.

## Query 3

Στο τρίτο ερώτημα καλούμαστε να εμφανίσουμε τις μεθόδους διάπραξης εγκλημάτων σε φθίνουσα συχνότητα εμφάνισης μαζί με τις περιγραφές τους. Εδώ χρειαζόμαστε δύο datasets, τα εγκλήματα και το MO_Codes.txt. Ο υπολογισμός του query θα γίνει με:

- RDD API
- Dataframe API

### RDD Implmentation

Ξεκινάμε με τον υπολογισμό με χρήση των RDDs. Πρώτα, φορτώνουμε από τα csv τα crime datasets για τα έτη 2010-2019 και 2020-2025 και παίρνουμε την ένωσή τους. Στην συνέχεια φορτώνουμε το MO_Codes.txt, στο οποίο περιλαμβάνεται ένα σύνολο αριθμών με τον καθένα να αντιστοιχίζεται σε ένα κακούργημα ή γενικότερα σε έναν χαρακτηρισμό ενός εγκλήματος. Επειδή τα δεδομένα δίνονται σε μορφή plaintext θα πρέπει με κατάλληλο mapping να διαχωρίσουμε το πρώτο substring της κάθε γραμμής (που δηλώνει τον κωδικό) από το υπόλοιπο, που αποτελεί την περιγραφή. Οι κωδικοί αυτοί εμφανίζονται στο crime dataset στην στήλη Mocodes και είναι πιθανό στην στήλη αυτή να αντιστοιχίζονται για ένα έγκλημα περισσότεροι του ενός κωδικοί. Θα πρέπει, λοιπόν, πριν προχωρήσουμε περαιτέρω να διαχωρίσουμε τους κωδικούς. Αυτό γίνεται μέσω ενός flatmap που αντιστοιχεί κάθε τέτοιο record στο σύνολο των κωδικών που το αποτελούν. Η χρήση του flatMap μας επιτρέπει να πάρουμε μία συνολική λίστα για όλους του κωδικούς. Έπειτα, μπορούμε να αντιστοιχίσουμε κάθε mo code στο ζευγάρι (mo_code, 1) και να κάνουμε reduceByKey ώστε να υπολογίσουμε τον συνολικό αριθμό εμφανίσεων κάθε κωδικού στα εγκλήματα.

Μετά από αυτήν την προετοιμασία των δεδομένων, το μόνο που μένει είναι είναι να κάνουμε join το rdd των μετρημένων κωδικών με τις περιγραφές τους. Τέλος, ταξινομούμε ως προς το πλήθος των εμφανίσεων και εμφανίζουμε τα αποτελέσματα. Ο χρόνος εκτέλεσης που καταγράφουμε είναι 37.12 δευτερόλεπτα.

### Dataframe Implmentation

Προχωράμε στην υλοποίηση με Dataframe API. Τα βήματα που πρέπει να ακολουθηθούν είναι κατά βάση τα ίδια, με την διαφορά ότι εδώ χρησιμοποιούμε κυρίως έτοιμες συναρτήσεις του API και όχι δικές μας. Έτσι, ο διαχωρισμός των mo codes του crime dataset γίνεται αρχικά με split και έπειτα με explode, έτσι ώστε να πάρουμε ένα ξεχωριστό row για κάθε κωδικό, με όλα τα υπόλοιπα attributes ίδια. Αντιθέτως, για την σωστή επεξεργασία του MO_Codes.txt φτιάχνουμε δύο UDFs που, με την βοήθεια της withColumn, μας δίνουν την επιθυμητή μορφή στα δεδομένα. Επιπλέον, είναι σημαντικό κατά την φόρτωση των δεδομένων να τους αποδώσουμε και το κατάλληλο σχήμα που τους αναλογεί.

Μετά από την προεπεξεργασία μπορούμε να κάνουμε group By ως προς τα mo_codes ώστε να συναθροίσουμε το πλήθος τους, να εκτελέσουμε το join των δύο σχέσεων και να ταξινομήσουμε ως προς το count, ακριβώς όπως κάναμε και στα rdds. Ωστόσο, η σημαντικότερη διαφορά μεταξύ των δύο υλοποιήσεων αναδεικνύεται στους χρόνους εκτέλεσης: το dataframe χρειάστηκε μόλις 15.53 δευτερόλεπτα έναντι των 37.12 του RDD. Εδώ αναδεικνύεται η αξία των DataFrames, όπου δεν εκτελούν απλώς τα βήματα υπολογισμού του query αλλά χρησιμοποιούν έναν optimizer, ένα πρόγραμμα δηλαδή που αναλύει το execution plan και το μετασχηματίζει κατάλληλα ώστε να μειωθεί ο χρόνος εκτέλεσης. Ο

optimizer που χρησιμοποιεί το Spark ονομάζεται Catalyst.

Με την εντολή explain μπορούμε να τυπώσουμε το πλάνο εκτέλεσης που εφάρμοσε ο optimizer για τον υπολογισμό του query. Έτσι, μπορούμε να δούμε την φόρτωση των δεδομένων εγκλημάτων από το csv, την ένωσή και την ομαδοποίησή τους ως προς τους κωδικούς μέσω της πράξης HashAggregate. Έπειτα, παρατηρούμε την φόρτωση του MO_Codes.txt, την εκτέλεση του join και την τελική ταξινόμηση. Εμπεριέχονται και περισσότερες πληροφορίες για τις επιμέρους επεξεργασίες που αναφέραμε παραπάνω, αλλά εδώ θα εστιάσουμε στο join. Ο αλγόριθμος εκτέλεσης του join που επιλέχθηκε είναι το Broadcast Hash Join.

### Broadcast Hash Join

Στο Broadcast Hash Join η μικρότερη εκ των δύο σχέσεων που συμμετέχουν στην πράξη γίνεται broadcast, μεταφέρεται δηλαδή ολόκληρη σε κάθε executor. Κάθε εργάτης, έπειτα, χρησιμοποιεί το partition που διαθέτει για την μεγάλη σχέση για να κάνει join με όλη την μικρή. Σε αυτό το στάδιο χρησιμοποιείται hash join, δηλαδή το κλειδί πάνω στο οποίο γίνεται η πράξη εισέρχεται σε μία συνάρτηση κατακερματισμού έτσι ώστε τα rows των δύο πινάκων που έχουν ίδια τιμή για αυτό να πέσουν σε κοινό bucket. Προϋποθέσεις για να μπορεί να εκτελεστεί το BHJ είναι η μία σχέση να είναι αρκετά μικρή ώστε να χωρά ολόκληρη στην κύρια μνήμη του κάθε worker και το join να γίνεται πάνω σε σχέση ισότητας. Στην περίπτωσή μας ικανοποιούνται και οι δύο προϋποθέσεις, αφού το MO_codes.txt περιλαμβάνει μόλις 615 γραμμές και το join γίνεται πάνω σε ισότητα του κωδικού. Να σημειωθεί, ακόμη, ότι το BHJ είναι η πιο αποτελεσματική τεχνική συνένωσης σε σχέση με τις υπόλοιπες (όταν μπορεί να εφαρμοστεί) λόγω της μειωμένης κίνησης στο δίκτυο που παράγει και της αποτελεσματικότητας που παρέχει το hashing του joined attribute.

Θα επιχειρήσουμε να πειραματιστούμε με διαφορετικές τεχνικές για το join και να τις συγκρίνουμε μεταξύ τους ώστε να βρούμε την καλύτερη. Μπορούμε να κατευθύνουμε το spark να χρησιμοποιήσει συγκεκριμένο αλγόριθμο για το join μέσω της εντολής hint. Σε αυτές τις περιπτώσεις τυπώνουμε μόνο το πρώτο row των αποτελεσμάτων [3] (εφόσον αυτό έχει ήδη τυπωθεί προηγουμένως στο συγκεκριμένο notebook) και τον χρόνο εκτέλεσης.

### Shuffle Hash Join

Αρχικά, επαναλαμβάνουμε την εκτέλεση χρησιμοποιώντας Shuffle Hash Join. Στο SHJ partitions και των δύο σχέσεων μεταφέρονται μεταξύ των workers έτσι ώστε rows που έχουν το ίδιο join attribute να καταλήξουν στον ίδιο εργάτη (γίνεται δηλαδή shuffle των σχέσεων). Έπειτα, χρησιμοποιείται hash join ώστε να υπολογιστεί ο σύνδεσμος, όπως έγινε και στο Broadcast Hash Join. Ο χρόνος εκτέλεσης που προκύπτει τώρα είναι 17.5 seconds. Η αύξηση στον χρόνο είναι αναμενόμενη. Το shuffle είναι πιο χρονοβόρα διαδικασία από το broadcast καθώς πρέπει να γίνει κατάλληλος διαχωρισμός των join κλειδιών στους executors και έπειτα να μετακινηθεί μεγάλο πλήθος από records μεταξύ όλων των συμμετεχόντων κόμβων. Έτσι, δημιουργείται μεγαλύτερη κίνηση στο δίκτυο κατά την μεταφορά, το οποίο επιφέρει και συμφόρηση. Αντιθέτως, ένα broadcast σε κάθε node γίνεται μία φορά σε όλους.

[3] είναι απαραίτητο να χρησιμοποιήσουμε την εντολή show ώστε να εκτελεστεί το query καθώς αυτό είναι το μόνο action που περιλαμβάνει ο κώδικάς μας

### Sort Merge Join

Κατόπιν, εκτελούμε sort merge join. Εδώ και πάλι γίνεται shuffle των δύο σχέσεων αλλά σε αυτήν την περίπτωση ο κάθε κόμβος σε εκτελεί hash join αλλά sort merge join. Στο sort merge join τα δύο dataframes ταξινομούνται ως προς το join attribute και έπειτα χρησιμοποιούνται δύο δείκτες, έναν για κάθε σχέση, οι οποίοι διατρέχουν σταδιακά τα tuples του κάθε πίνακα ώστε να εντοπίσουν αυτά που ικανοποιούν την συνθήκη του σύνδεσμου. Το πλεονέκτημα του sort merge join είναι ότι δουλεύει σε non equality joins, αλλά επιβαρύνει την εκτέλεση με το κόστος της ταξινόμησης των σχέσεων, που είναι πιο αργή διαδικασία σε σχέση με την εφαρμογή μίας hash function. Πράγματι, ο νέος καταγεγραμμένος χρόνος είναι 18.07 seconds, μεγαλύτερος από τους δύο προηγούμενους.

### Shuffle Nested Loop Join

Τέλος, θα εφαρμόσουμε το Shuffle Nested Loop Join. Σε αυτήν την περίπτωση το join που υλοποιείται σε κάθε κάθε κόμβο είναι nested loop, περιλαμβάνει δηλαδή έναν διπλά εμφωλιασμένο βρόχο μέσα στον οποίο συγκρίνεται κάθε tuple της μίας σχέσης με όλα τα tuples της άλλης. Βλέπουμε ότι αυτή η μέθοδος είναι η λιγότερο αποδοτική από όλες τις προηγούμενες εφόσον χρειάζεται να ελέγξουμε nxm tuples (αν οι σχέσεις έχουν n και m rows αντίστοιχα) και ουσιαστικά πραγματοποιούμε το καρτεσιανό γινόμενο των δύο σχέσεων. Όντως, ο χρόνος εκτέλεσης προκύπτει ίσος με 20.22 δευτερόλεπτα, ο μεγαλύτερος από όλους όσους εξετάσαμε. Συμπερασματικά, η κατάταξη των μεθόδων join ως προς τις επιδόσεις τους είναι Broadcast Hash Join > Shuffle Hash Join > Shuffle Sort-Merge Join > Shuffle Nested Loop Join.

## Query 4

Στο τέταρτο ερώτημα ζητείται να υπολογιστούν για κάθε αστυνομικό τμήμα ο συνολικός αριθμός των εγκλημάτων που συνέβησαν πλησιέστερα σε αυτό μαζί με την μέση απόστασή τους από αυτό. Ακόμη, τα τμήματα πρέπει να ταξινομηθούν με φθίνοντα αριθμό εγκλημάτων. Για αυτό το ερώτημα θα χρειαστούμε δύο datasets, το Los Angeles Crime Dataset και το LA Police Stations. Ξεκινάμε φορτώνοντας τα κατάλληλα csv στο notebook, με βάση το σχήμα δεδομένων που τα αντιπροσωπεύει. Η υλοποίηση του ερωτήματος θα γίνει με SQL API.

Ξεκινάμε αφαιρώντας από τα εγκλήματα αυτά που οι συντεταγμένες τους βρίσκονται στο null island (δηλαδή στο (0,0)) καθώς αυτά είναι "θορυβώδη" δεδομένα που θα οδηγήσουν σε εσφαλμένα αποτελέσματα. Στην συνέχεια εκτελούμε ένα sql query που υπολογίζει για κάθε έγκλημα το αστυνομικό τμήμα που απέχει την ελάχιστη απόσταση από αυτό και την τιμή αυτής της απόστασης σε χιλιόμετρα. Πιο αναλυτικά, εκτελούμε το καρτεσιανό γινόμενο μεταξύ των LA crimes και LA stations και για κάθε συνδυασμό υπολογίζουμε την απόσταση μεταξύ τους. Έπειτα, κάνουμε ένα partition πάνω στα crimes, ταξινομούμε σε αύξουσα σειρά τις αποστάσεις του κάθε partition και επιλέγουμε το πρώτο. Έτσι, καταλήγουμε σε μία σχέση που περιέχει την απαραίτητη πληροφορία που χρειαζόμαστε.

Το μόνο που μένει είναι να κάνουμε group by LA station και να υπολογίσουμε σε aggregate function την μέση τιμή των αποστάσεων και το πλήθος των rows. Τέλος, ταξινομούμε ως προς το πλήθος και παρουσιάζουμε τα αποτελέσματα.

Να σημειωθεί ότι οι αποστάσεις υπολογίστηκαν με την βοήθεια της βιβλιοθήκης sedona, η οποία είναι κατάλληλη για επεξεργασία γεωγραφικών δεδομένων που έχουν datatype geometry. Για αυτόν τον σκοπό, πρώτα μετατρέπουμε τις γεωγραφικές συντεταγμένες σε τύπο geometry μέσω της συνάρτησης ST_POINT και έπειτα υπολογίζουμε την απόσταση των geometry types με την συνάρτηση ST_DistanceSphere, η οποία λαμβάνει υπόψιν και την καμπυλότητα της γης.

Χρησιμοποιώντας την εντολή explain εδώ μπορούμε να δούμε ότι το execution plan του Catalyst για την παραπάνω ακολουθία ερωτημάτων περιλαμβάνει Broadcast Nested Loop Join. Αυτό αφορά τον υπολογισμό του καρτεσιανού γινομένου των εγκλημάτων με τα αστυνομικά τμήματα και είναι η καλύτερη δυνατή επιλογή για το join. Πράγματι, δεδομένου ότι δεν υπάρχει condition στο join η μόνη επιλογή που απομένει είναι το nested loop join ώστε να παρθούν όλοι οι δυνατοί συνδυασμοί. Ωστόσο, ο Catalyst εκμεταλλεύεται το γεγονός ότι η σχέση LA Police Stations είναι πολύ μικρή (μόνο 21 rows) για την κάνει broadcast σε όλους τους executors. Έτσι, μειώνεται η ποσότητα των δεδομένων που μεταφέρεται στο δίκτυο και κάθε executor μπορεί να δουλέψει ατομικά στο join του partition του με όλη την βάση των LA stations.

Στα υλοποίηση των προηγούμενων ερωτημάτων χρησιμοποιούσαμε πάντα τέσσερις executors με έναν πυρήνα και 2 GB μνήμη ο καθένας. Τώρα θα πειραματιστούμε με διαφορετικό συνδυασμό πόρων. Συγκεκριμένα, θα καταγράψουμε τους χρόνους εκτέλεσης του query για τα ακόλουθα configurations:

- 2 executors, 1 core και 2GB μνήμης per executor
- 2 executors, 2 cores και 4GB μνήμης per executor
- 2 executors, 4 cores και 8GB μνήμης per executor

Οι χρόνοι που προκύπτουν για τους παραπάνω συνδυασμούς είναι, αντίστοιχα, 38.14 sec, 35.12 sec και 27.89 sec. Παρατηρούμε ότι καθώς αυξάνουμε τους πυρήνες και την μνήμη οι χρόνοι εκτέλεσης πέφτουν. Η συμπεριφορά αυτή είναι λογική. Με περισσότερους πυρήνες μπορούμε να χωρίσουμε τις διαδικασίες υπολογισμού του query σε περισσότερους εργάτες και άρα να πετύχουμε μεγαλύτερη παραλληλία. Από την άλλη, η μεγαλύτερη μνήμη επιτρέπει στους executors να διατηρούν μεγαλύτερο κομμάτι του dataframe στην κύρια μνήμη και να μην το μεταφέρουν στον σκληρό δίσκο. Οι λειτουργίες I/O με τον μόνιμο αποθηκευτικό χώρο είναι από τις πιο χρονοβόρες διαδικασίες στα συστήματα διαχείρισης βάσεων δεδομένων και η ελαχιστοποίησή τους είναι μείζονος σημασίας. Συνεπώς, η μεγαλύτερη μνήμη ευνοεί την ταχύτητα εκτέλεσης, όπως και γενικότερα η αύξηση των resources.

## Query 5

Στο πέμπτο και τελευταίο query θα χρειαστεί να συνδυάσουμε δημογραφικά στοιχεία για το 2020 με δεδομένα εισοδημάτων ανά νοικοκυριό από το 2021 ώστε να συσχετίσουμε το μέσο κατά κεφαλήν εισόδημα με τον μέσο αριθμό εγκλημάτων ανά άτομο για κάθε περιοχή του LA. Πιο αναλυτικά, θα χρειαστούμε τα εξής δεδομένα:

- Το Los Angeles Crime Dataset που έχουμε χρησιμοποιήσει ήδη εκτενώς
- Το Median Household Income by Zip Code Dataset, στο οποίο αντιστοιχίζεται κάθε ταχυδρομικός κωδικός της πόλης με το μέσο εισόδημα ανά νοικοκυριό για το 2021 για αυτή την περιοχή
- Το Census Blocks Dataset, όπου περιλαμβάνονται πληροφορίες για κάθε οικοδομικό τετράγωνο του LA. Συγκεκριμένα, πέρα από τα γεωγραφικά χαρακτηριστικά τύπου geometry, περιλαμβάνεται ο πληθυσμός και ο αριθμός σπιτιών ανά μπλοκ

Τα census blocks ανήκουν σε ένα σύνολο περιοχών του Los Angeles που χαρακτηρίζεται από το πεδίο COMM του αντίστοιχου συνόλου δεδομένων. Σε κάθε τέτοια περιοχή θα έρθουμε να υπολογίσουμε το κατά κεφαλήν εισόδημα. Αρχικά, μετατρέπουμε, με την βοήθεια μίας udf, τα χρηματικά ποσά που περιέχονται στο LA_Income_2021.csv σε ακέραιους για ευκολότερη επεξεργασία. Κατόπιν, εκτελούμε το join ανάμεσα στο dataframe των εισοδημάτων με το dataframe των census blocks πάνω στο zip code. Έπειτα, σε κάθε μπλοκ υπολογίζουμε το γινόμενο του εισοδήματος ανά κατοικία με τον αριθμό των σπιτιών σε αυτό. Με αυτόν τον τρόπο παίρνουμε το συνολικό εισόδημα ανά οικοδομικό τετράγωνο. Τέλος, πραγματοποιούμε group by πάνω στις περιοχές (στήλη COMM), οπότε το κατά κεφαλήν εισόδημα της περιοχής δίνεται από το άθροισμα των εσόδων δια το άθροισμα του πληθυσμού των επιμέρους μπλοκ της περιοχής.

Στην συνέχεια, θα υπολογίσουμε το αριθμό των εγκλημάτων που αναλογούν σε κάθε άτομο ανά περιοχή. Για αυτόν τον σκοπό θα χρειαστεί να κάνουμε join το crime dataset (για τα εγκλήματα της διετίας 2020-2021) με τις περιοχές με βάση το πεδίο geometry. Η συνθήκη του join θα είναι το αν το έγκλημα διεπράχθη εντός της εκάστοτε γεωγραφικής περιοχής. Για να υπολογίσουμε τα γεωγραφικά όρια της περιοχής θα πρέπει να κάνουμε group by COMM τα census blocks και aggregate με χρήση της συνάρτησης ST_Union_Aggr της βιβλιοθήκης sedona. Ύστερα, ο έλεγχος του join γίνεται με την συνάρτηση ST_Within, πάλι της sedona. Αφού έχουν γίνει αυτά, χρειάζεται μία τελική συνάθροιση πάνω στις περιοχές για να υπολογίσουμε το άθροισμα των εγκλημάτων δια το σύνολο του πληθυσμού σε κάθε μία από αυτές. Ακολουθεί και ένας πολλαπλασιασμός επί 1000 για να μην προκύψουν πολύ μικρά νούμερα.

Τώρα που έχουμε υπολογίσει τόσο το κατά κεφαλήν εισόδημα όσο και τον αριθμό εγκλημάτων ανά άτομο, αρκεί να κάνουμε ένα ακόμα join αυτών των δύο σχέσεων πάνω στην περιοχή. Το τελικό αποτέλεσμα περιλαμβάνει για κάθε πεδίο COMM τις τιμές των μετρικών που του αντιστοιχούν. Τέλος, ταξινομούμε με αύξουσα και φθίνουσα σειρά ως προς το κατά κεφαλήν εισόδημα και παίρνουμε τα 10 πρώτα στοιχεία σε κάθε περίπτωση για να εμφανίσουμε τα αποτελέσματα για τις δέκα πιο πλούσιες και τις δέκα πιο φτωχές περιοχές του LA.

### Συμπεράσματα

Στο συγκεκριμένο query εκτελούμε 3 joins. Σύμφωνα με το execution plan που κατασκευάζει ο optimizer τα join strategies με την σειρά που εμφανίζονται στον κώδικα (το οποίο δεν ταυτίζεται την σειρά εκτέλεσης) είναι:

- Broadcast Hash Join
- Range Join
- Sort Merge Join

Το πρώτο join χρησιμοποιεί Broadcast Hash Join, καθώς το income_df dataframe (LA_income_2021.csv) είναι σχετικά μικρό, οπότε μπορεί να το κάνει broadcast εύκολα σε όλους τους executors, αποφεύγοντας πολλά shuffles μεταξύ των κόμβων. Επιπλέον το γεγονός ότι πρόκειται περί equality join καθιστά το Broadcast Hash Join το καταλληλότερο είδος συνδέσμου για αυτήν την περίπτωση.

Το δεύτερο join χρησιμοποιεί Range Join, διότι πρόκειται για join με γεωγραφικό join condition (ST_Within()) και όχι συνθήκη ισότητας, την οποία απαιτεί το Broadcast Hash Join.

Το τρίτο join χρησιμοποιεί Sort Merge Join, το οποίο είναι το default και πιο γενικό join strategy, καθώς τα δύο dataframes που χρησιμοποιούμε είναι αρκετά μεγάλα και μάλλον υπερβαίνουν το όριο για Broadcast Hash Join.

Οι χρόνοι εκτέλεσης για κάθε configuration παρουσιάζονται κάτωθι:

| **Scenario** | **Executors** | **Cores/Exec** | **Memory/Exec** | **Execution Time (s)** |
|---|---|---|---|---|
| Config A | 2 | 4 | 8g | 82.18 |
| Config B | 4 | 2 | 4g | 79.43 |
| Config C | 8 | 1 | 2g | 84.41 |

Table 2: Σύγκριση χρόνων εκτέλεσης σε διαφορετικά Spark Configurations (Total: 8 Cores, 16GB RAM)

Ένας Executor είναι ένας process του JVM που τρέχει σε έναν κόμβο (Worker Node), έχει την δική του μνήμη την οποία ορίζουμε εμείς (Memory/Exec) και την μοιράζονται μόνο τα cores του και κανένα άλλο. Οπότε, παρότι κάθε core σε κάθε exector τρέχει παράλληλα, μόνο τα cores στον ίδιο executor έχουν κοινή μνημη.

Οπότε, έχοντας σταθερό αριθμό από cores και συνολική μνήμη RAM, φτάνουμε σε ένα πρόβλημα optimization κατανομής πόρων. Από την μία πλευρά αν έχω όλα τα cores μου σε λίγα executors θα προκύπτουν συνθήκες ανταγωνισμού για τον διαμοιρασμό των κοινών πόρων, ενώ προστίθεται overhead στην διαχείριση των εργατών. Μάλιστα, αν κάποιοι πυρήνες καθυστερούν σημαντικά (για τον οποιονδήποτε λόγο) τότε θα καταναλώνουν μνήμη άσκοπα που θα μπορούσαν να αξιοποιήσουν τα άλλα cores. Από την άλλη, αν μοιράσουμε τα cores σε πολλά executors, τότε θα έχουμε λιγότερο ανταγωνισμό για την μνήμη αλλά μικρότερη ικανότητα ενός αποδοτικού scheduling ανά executor.

Από τα παραπάνω πειραματικά δεδομένα φαίνεται ότι βέλτιστη (ανάμεσα στα 3 αυτά configurations που δοκιμάσαμε) είναι μια μέση λύση με 4 executors που έχουν 4GB RAM και 2 cores/executor. Η λύση αυτή προσπαθεί να συνδυάσει τα καλά χαρακτηριστικά από τα

δύο άκρα και φαίνεται να δουλεύει καλύτερα από τα άλλα δύο.

Ακόμη, μεταξύ των δύο ακραίων υλοποιήσεων (A και C στον παραπάνω πίνακα) παρατηρούμε ότι το configuration C με τα πολλά executors άλλα λίγη μνήμη είναι λίγο πιο αργό από το A. Αυτό μπορεί να οφείλεται στο γεγονός ότι τα tables/dataframes τα οποία έχουμε είναι μεγάλα, ειδικά στην αρχή που δεν έχει γίνει καμία πράξη σε αυτά, οπότε τα tasks θα απαιτούν γενικά πολύ μνήμη που ίσως να μην αρκούν τα 2GB.