Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Προχωρημένα Θέματα Βάσεων Δεδομένων
Εξαμηνιαία Εργασία

Κωνσταντίνος Κατσικόπουλος (03120103)
Χαρίδημος Παπαδάκης (03120022)

Ιανουάριος 2025

Query 1: Ανάλυση Ηλικιακών Ομάδων με DataFrame & RDD

Το ερώτημα απαιτεί την ταξινόμηση, σε φθίνουσα σειρά, των ηλικιακών ομάδων των θυμάτων σε περιστατικά που περιλαμβάνουν οποιαδήποτε μορφή "βαριάς σωματικής βλάβης" (Aggravated Assault). Οι ομάδες ορίζονται ως:

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

Η υλοποίηση έγινε με δύο τρόπους, αξιοποιώντας **4 Spark executors** σε περιβάλλον AWS:

DataFrame API

- Φορτώσαμε τα δεδομένα (2010–2019 & 2020–*present*) και τα ενώσαμε σε ένα ενιαίο `DataFrame`.
- Χρησιμοποιήσαμε φίλτρο με `like(%aggravated assault%)` ώστε να εστιάσουμε μόνο στα περιστατικά που χαρακτηρίζονται ως Aggravated Assault.
- Δημιουργήσαμε μια νέα στήλη (*Age Group*) όπου ταξινομήθηκαν οι ηλικίες των θυμάτων στις τέσσερις ομάδες.
- Τέλος, ομαδοποιήσαμε ανά *Age Group* και υπολογίσαμε το πλήθος, ταξινομώντας σε φθίνουσα σειρά ως προς τον αριθμό των περιστατικών.

Ο κώδικας είναι αρκετά high-level και sql-like — οι συναρτήσεις `filter`, `withColumn`, `groupBy`, `agg` και `orderBy` αρκούν για να περιγράψουμε εύκολα την επιθυμητή λογική, ενώ ο Catalyst optimizer του Spark αναλαμβάνει εσωτερικά τις βελτιστοποιήσεις.

RDD API

- Διαβάσαμε τα ίδια αρχεία σε μορφή `RDD` και τα ενώσαμε.
- Εφαρμόσαμε `filter` με συνθήκη `if 'aggravated assault' in row['Crm Cd Desc'].lower()`.
- Εν συνεχεία, χρησιμοποιήσαμε `map` για να μετατρέψουμε την ηλικία (`Vict Age`) σε μία από τις τέσσερις ηλικιακές ομάδες, και δημιουργήσαμε key-value pairs `(AgeGroup, 1)` σε κάθε περιστατικό.
- Με `reduceByKey` υπολογίσαμε το άθροισμα περιστατικών ανά ηλικιακή ομάδα.
- Η ταξινόμηση έγινε με χρήση `sortBy` σε φθίνουσα σειρά.

Η υλοποίηση με χρήση RDD είναι λιγότερο "αυτοματοποιημένη": αντί να γράψουμε ένα απλό `groupBy` το οποίο αναλαμβάνει αυτόματα ο Catalyst, υλοποιήσαμε την ομαδοποίηση και την ταξινόμηση με ειδικούς μετασχηματισμούς RDD.

Σύγκριση Επίδοσης & Συμπεράσματα

– **Αποτελέσματα**:

```
DataFrame Time: 22.04
DataFrame Results:
Adults: 121093
Young Adults: 33605
Children: 15928
Seniors: 5985

RDD Time: 22.25
RDD Results:
Adults: 121093
Young Adults: 33605
Children: 15928
Seniors: 5985
```

– **Χρόνοι εκτέλεσης:** Στην τελευταία δοκιμή, η υλοποίηση με χρήση DataFrames ολοκληρώθηκε σε $\sim$22.04s, ενώ με χρήση RDDs σε $\sim$22.25s. Ωστόσο, επαναλαμβάνοντας τα ερωτήματα *πολλαπλές φορές*, παρατηρήσαμε ότι υπάρχουν μικρές διακυμάνσεις, κυρίως λόγω του περιβάλλοντος AWS (διαφορετικές συνθήκες φόρτου στο cluster, caching κ.λπ.). Σε κάθε περίπτωση, και οι δύο χρόνοι ήταν πρακτικά παραπλήσιοι.

– **Ερμηνεία:** Το DataFrame API είναι εμφανώς πιο υψηλού επιπέδου και ευανάγνωστο, εκμεταλλευόμενο τις εσωτερικές βελτιστοποιήσεις του Spark. Το RDD API επιτρέπει λεπτομερέστερο έλεγχο, αλλά συνήθως απαιτεί περισσότερο κώδικα. Στο συγκεκριμένο query, οι διαφορές ταχύτητας ήταν αμελητέες, δεδομένου ότι πρόκειται για μία αρκετά απλή περίπτωση.

Επομένως, παρότι θεωρητικά τα DataFrames υπερτερούν σε απόδοση λόγω του Catalyst optimizer, σε αυτό το συγκεκριμένο σενάριο οι υλοποιήσεις ήταν σχεδόν ισοδύναμες. Η επαναλαμβανόμενη εκτέλεση των ερωτημάτων *επιβεβαίωσε* ότι οι μικρές αποκλίσεις οφείλονται κυρίως στη δυναμική του εκτελεστικού περιβάλλοντος παρά στη διαφορά μεταξύ DataFrame και RDD.

Query 2: Ανάλυση Κλεισμένων Υποθέσεων με Dataframe & SQL

Στόχος

Ο σκοπός του Query 2 είναι η ανάλυση των ποσοστών κλεισμένων (περατωμένων) υποθέσεων για κάθε Αστυνομικό Τμήμα στην πόλη του Los Angeles. Τα δεδομένα εξετάζονται σε ετήσια βάση, και για κάθε έτος εντοπίζονται τα τρία τμήματα με τα υψηλότερα ποσοστά περατωμένων υποθέσεων. Τα αποτελέσματα παρουσιάζονται σε σειρά ταξινόμησης ανά έτος και κατάταξη.

Υλοποίηση με DataFrame API

Η ανάλυση με το DataFrame API περιλαμβάνει τη χρήση βασικών λειτουργιών Spark για την ανάγνωση, επεξεργασία, και ανάλυση δεδομένων. Τα βήματα αναλύονται ως εξής:

Ανάγνωση δεδομένων: Διαβάσαμε τα 2 κύρια αρχεία CSV από το S3 bucket ακριβώς όπως στο προηγούμενο ερώτημα.

Ενοποίηση δεδομένων: Τα δύο αρχεία συνδυάστηκαν με τη μέθοδο `union()` αφού επιλέχθηκαν και μετονομάστηκαν οι απαραίτητες στήλες (`year`, `precinct`, `closed_case`). Για τον υπολογισμό των κλεισμένων υποθέσεων, χρησιμοποιήθηκε η συνθήκη:

```
CASE WHEN `Status Desc` != 'UNK' AND `Status Desc` != 'Invest Cont' THEN 1 ELSE 0 END
```

Ομαδοποίηση και Υπολογισμοί: Τα δεδομένα ομαδοποιήθηκαν ανά έτος και τμήμα με τη χρήση της `groupBy()` για τον υπολογισμό:

- Του ποσοστού κλεισμένων υποθέσεων (`closed_case_rate`).
- Του συνολικού πλήθους υποθέσεων (`total_cases`).

Η παραπάνω ανάλυση εφαρμόστηκε αποτελεσματικά μέσω των λειτουργιών `count()` και `when()`.

Ταξινόμηση και Κατάταξη: Χρησιμοποιήθηκε το Spark Window API με `partitionBy("year")` και `orderBy(col("closed_case_rate").desc())` για την κατάταξη των τμημάτων με βάση το ποσοστό κλεισμένων υποθέσεων. Η στήλη κατάταξης (#) προέκυψε με τη συνάρτηση `rank()`.

Υλοποίηση με SQL API

Η υλοποίηση με SQL API βασίστηκε σε ένα SQL query πολλαπλών επιπέδων. Τα δεδομένα αναλύθηκαν με τα εξής στάδια:

Ανάγνωση δεδομένων: Δημιουργήθηκαν `temporary views` για τα δύο αρχεία CSV.

Συνδυασμός και Υπολογισμοί: Με τη χρήση του `WITH combined_data AS` συνδυάστηκαν τα δεδομένα και υπολογίστηκε το `closed_case_rate` με τον τύπο:

$$\frac{\text{COUNT(CASE WHEN Status Desc} \neq \text{'UNK' AND Status Desc} \neq \text{'Invest Cont')} \times 100}{\text{COUNT(*)}}$$

Η κατάταξη πραγματοποιήθηκε με `RANK() OVER (PARTITION BY YEAR(date) ORDER BY closed_case_rate DESC)`.

Σύγκριση Υλοποίησης με DataFrame API και SQL API

Η σύγκριση μεταξύ των δύο υλοποιήσεων (DataFrame API και SQL API) εστιάζει στη διαφορά απόδοσης, ευχρηστίας, και ευελιξίας κατά την επεξεργασία δεδομένων.

Χρόνοι Εκτέλεσης: Η απόδοση των δύο μεθόδων μετρήθηκε σε όρους χρόνου εκτέλεσης τόσο για την ανάγνωση δεδομένων όσο και για την ανάλυση:

- **DataFrame API:**
  - Χρόνος ανάγνωσης δεδομένων: `13.4` δευτερόλεπτα
  - Χρόνος ανάλυσης και υπολογισμών: `5.8` δευτερόλεπτα
  - Συνολικός χρόνος: `19.2` δευτερόλεπτα
- **SQL API:**
  - Χρόνος ανάγνωσης δεδομένων: `13.77` δευτερόλεπτα
  - Χρόνος ανάλυσης και υπολογισμών: `6.07` δευτερόλεπτα
  - Συνολικός χρόνος: `19.84` δευτερόλεπτα

Συμπεράσματα:

- Οι δύο υλοποιήσεις έχουν προφανώς το ίδιο read time.
- Οι αμελητέες διαφορές στο execution time είναι λογικές, καθώς το SQL API και το Dataframe API χρησιμοποιούν τον Catalyst Optimizer. Έτσι το execution plan θα έπρεπε να είναι σχεδόν ίδιο λόγω της ίδιας σημασιολογίας των queries και το execution time.

Σύγκριση μεταξύ CSV και Parquet

Τα δεδομένα μετατράπηκαν σε μορφή Parquet για τη βελτιστοποίηση της ανάγνωσης και της ανάλυσης:

- Η μορφή Parquet προσφέρει καλύτερη συμπίεση και ταχύτερη πρόσβαση λόγω columnar storage.
- Συγκρίθηκαν οι χρόνοι ανάγνωσης και εκτέλεσης με CSV και Parquet για την SQL API υλοποίηση.

Αποτελέσματα σύγκρισης:

- Χρόνος ανάγνωσης με CSV: `13.77` δευτερόλεπτα
- Χρόνος ανάγνωσης με Parquet: `7.54` δευτερόλεπτα
- Χρόνος ανάλυσης και εκτέλεσης με CSV: `6.07` δευτερόλεπτα
- Χρόνος ανάλυσης και εκτέλεσης με Parquet: `8.46` δευτερόλεπτα

Συμπεράσματα:

- Η χρήση του Parquet μειώνει σημαντικά τον χρόνο ανάγνωσης ($\sim 45\%$), ακριβώς όπως αναμέναμε.
- Για τον χρόνο εκτέλεσης παρατηρούμε μία αύξηση που μπορεί να οφείλεται σε διάφορους εξωτερικούς παράγοντες εκτός από την διαχείρηση των δεδομένων μέσω Parquet(κυρίως σχετικούς με το περιβάλλον AWS).

Σημαντική σημείωση: Παρόλο που οι ενέργειες στο Spark είναι lazily evaluated, είναι εφικτό να μετρήσουμε τον χρόνο ανάγνωσης στην προκειμένη περίπτωση. Επειδή έχουμε θέσει την παράμετρο `inferSchema` του `spark.read` ως αληθή, το Spark αναγκαστικά θα διαβάσει το αρχείο εισόδου την στιγμη που καλούμε την συνάρτηση ώστε να γίνει ο συμπερασμός τύπων των στηλών του dataframe που θα προκύψει.

Query 3: Ανάλυση Εισοδήματος & Εγκληματικότητας

Στόχος

Το Query 3 συνδυάζει:

- Δεδομένα **πληθυσμού** (απογραφή 2010) και **εισοδήματος** (2015), με στόχο τον υπολογισμό ενός κατά προσέγγιση *μέσου ετήσιου εισοδήματος ανά άτομο* (`AVERAGE_INCOME_PER_PERSON`).
- Δεδομένα **εγκλημάτων** (2010 έως σήμερα), χωρικά συνενωμένα (`ST_Within(geom, geometry)`) στα Census Blocks του Λος Άντζελες, προκειμένου να υπολογιστεί η *αναλογία εγκλημάτων ανά κάτοικο* (`CRIME_RATIO_PER_PERSON`).

Βασική Προσέγγιση

1. **Πληθυσμός + Εισόδημα (Population+Income Join):** Κάνουμε `join` τα Census Blocks (με `COMM`, `ZIPCODE`, `POPULATION`, `HOUSING_UNITS`) και τα δεδομένα εισοδήματος 2015 (`MEDIAN_INCOME`), χρησιμοποιώντας το `ZIPCODE` ως κλειδί. Μετά από αυτό το join κάθε Census Block έχει το δικό του Median Household Income. Ωστόσο, σε μία περιοχή (`COMM`) μπορεί να αντιστοιχούν Blocks με διαφορετικά Median Household Incomes. Επομένως, ομαδοποιούμε (`groupBy("COMM")`) για να αθροίσουμε:

   $$\text{TOTAL\_POPULATION} = \sum(\text{POPULATION}),\ \text{TOTAL\_INCOME} = \sum(\text{MEDIAN\_INCOME} \times \text{HOUSING\_UNITS})$$

   και εφαρμόζουμε `ST_Union_Aggr` πάνω στο `geometry`, ενοποιώντας όλα τα blocks ανά περιοχή. Έτσι ορίζουμε:

   $$\text{AVERAGE\_INCOME\_PER\_PERSON} = \frac{\text{TOTAL\_INCOME}}{\text{TOTAL\_POPULATION}}.$$

2. **Εγκλήματα + PopIncome Join (ST_Within):** Διαβάζουμε τα εγκλήματα (2010–σήμερα), κατασκευάζουμε `ST_Point(LON, LAT)` και φιλτράρουμε τα δεδομένα. Με το `ST_Within`(geom, geometry) αναθέτουμε κάθε έγκλημα στην κατάλληλη περιοχή. Τέλος, ομαδοποιούμε για να βρούμε το συνολικό πλήθος εγκλημάτων (`TOTAL_CRIMES`) και υπολογίζουμε:

   $$\text{CRIME\_RATIO\_PER\_PERSON} = \frac{\text{TOTAL\_CRIMES}}{\text{TOTAL\_POPULATION}}.$$

Σύγκριση Στρατηγικών Join & Χρόνοι Εκτέλεσης

Για να αξιολογήσουμε την επίδραση διαφορετικών `join` strategies στο Spark, εκτελέσαμε πολλαπλές επαναλήψεις με *hint* (`broadcast`, `shuffle_hash`, `shuffle_replicate_nl`, `merge`) και με μια *default* υλοποίηση για να δούμε ποια στρατηγική επιλέγει το Spark. Καταγράψαμε το `Physical Plan` με `.explain()` και τους χρόνους εκτέλεσης.

1. Population + Income Join (ZIPCODE)

Στρατηγικές:

- **BroadcastHashJoin**
- **ShuffledHashJoin**
- **CartesianProduct** (όταν ορίστηκε `shuffle_replicate_nl`)
- **SortMergeJoin**

Χρόνοι Εκτέλεσης:

- `broadcast`: 11s
- `shuffle_hash`: 11.56s
- `shuffle_replicate_nl`: 9.83s
- `merge`: 10.03s

Παρατηρούμε ότι οι χρόνοι εκτέλεσης *διαφέρουν ελάχιστα* (εντός $\sim$1 δευτερολέπτου). Στην *default* περίπτωση το Spark χρησιμοποίησε **BroadcastHashJoin**, πιθανόν επειδή το dataset των εισοδημάτων είναι αρκετά μικρό για broadcast.

Ποια (θεωρητικά) Στρατηγική είναι καλύτερη:

- *BroadcastHashJoin* είναι **συνήθως** πιο αποδοτική όταν ένα από τα δύο DataFrames είναι αρκετά μικρό, για να αποφεύγονται shuffles.
- *SortMergeJoin* είναι γενικά η προεπιλογή σε μεγάλα DataFrames.
- *ShuffledHashJoin* μπορεί να είναι πιο γρήγορη αν τα δεδομένα δεν είναι ήδη ταξινομημένα, αλλά απαιτεί αρκετή μνήμη.
- *shuffle_replicate_nl* (Cartesian Product) είναι συνήθως θεωρητικά η πιο αργή—εφαρμόζεται όταν είναι απαραίτητο.

Στο συγκεκριμένο παράδειγμα, το broadcast απέδωσε παρόμοια με τις υπόλοιπες στρατηγικές, διότι τα δεδομένα δεν ήταν πολύ μεγάλα ώστε το σύστημα να τα διαχειριστεί.

2. Crime + PopIncome Join (`ST_Within`)

Παρότι ζητήσαμε τις ίδιες 4 στρατηγικές, *στην πράξη* μόνο δύο εφαρμόστηκαν:

- **BroadcastIndexJoin**, όταν ζητήθηκε `broadcast`,
- **RangeJoin**, όταν ζητήθηκε `shuffle_hash`, `shuffle_replicate_nl` ή `merge`, καθώς και στο `default`.

Δηλαδή, αν και ορίσαμε συγκεκριμένες στρατηγικές το Spark/Sedona αγνόησε τις αγνόησε και εφάρμοσε `RangeJoin`, ίσως επειδή δεν είναι κατάλληλες για spatial queries.

Χρόνοι Εκτέλεσης:

- `broadcast`: 13.54s (*BroadcastIndexJoin*)
- `shuffle_hash`: 13.50s (*RangeJoin*)
- `shuffle_replicate_nl`: 13.42s (*RangeJoin*)
- `merge`: 13.47s (*RangeJoin* αντί SortMergeJoin)

Και σε αυτό το χωρικό join, οι χρόνοι διαφέρουν πολύ λίγο. Στο *default*, χρησιμοποιήθηκε επίσης **RangeJoin**.

Ποια (θεωρητικά) Στρατηγική είναι καλύτερη:

- *BroadcastIndexJoin* (όταν τα γεωμετρικά δεδομένα είναι μικρά) μπορεί να είναι πολύ γρήγορη, γιατί αποφεύγεται μεγάλο shuffle.
- *RangeJoin* (`ST_Within`) θεωρείται από το Sedona ως «βέλτιστη» όταν τα δεδομένα είναι μεγαλύτερα.
- *SortMergeJoin* δεν υλοποιήθηκε καθόλου εδώ, διότι οι χωρικές συναρτήσεις δεν υποστηρίζονται εύκολα με κλασικό sort-merge.

Συμπεράσματα

- **Μικρές διαφορές χρόνων**: Σε όλα τα πειράματα, οι διάφορες στρατηγικές παρουσίασαν πολύ μικρές αποκλίσεις (0.5–1s). Οι συχνές επαναλήψεις επιβεβαίωσαν ότι δεν υπάρχει ουσιώδης διαφορά, πιθανώς λόγω μεγέθους/διάρθρωσης των δεδομένων και caching στο σύστημα.
- **Default Συμπεριφορά Spark/Sedona**:
  - Στο *Population+Income Join*, το Spark προτίμησε *BroadcastHashJoin*, κάτι θεωρητικά αναμενόμενο όταν το ένα DataFrame (π.χ. εισόδημα) δεν είναι πολύ μεγάλο.
  - Στο *Crime+PopIncome Join (ST_Within)*, το Sedona επέλεξε *BroadcastIndexJoin* μόνο όταν ορίσαμε broadcast, αλλιώς χρησιμοποίησε *RangeJoin* (παρά το *merge* ή *shuffle_hash* hint).
- **Θεωρητικά**:
  - Για *μεγάλα* datasets, *SortMergeJoin* ή *ShuffledHashJoin* είναι συνήθως οι «καθιερωμένες» επιλογές, διότι το broadcast μπορεί να υπερβεί τη διαθέσιμη μνήμη.
  - Για *μικρά* datasets, το *BroadcastHashJoin* (ή *BroadcastIndexJoin* στα χωρικά) είναι κατάλληλη στρατηγική, αποφεύγοντας τα shuffles.
  - *shuffle_replicate_nl* (Cartesian Product) είναι θεωρητικά η λιγότερο αποδοτική στρατηγική.

Query 4: Ανάλυση Φυλετικής Ταυτότητας Θυμάτων

Στόχος

Στο Query 4 συνδυάζουμε:

- Δεδομένα **πληθυσμού** (απογραφή 2010) και **εισοδήματος** (2015), με στόχο τον υπολογισμό ενός κατά προσέγγιση *μέσου ετήσιου εισοδήματος ανά άτομο* (`AVERAGE_INCOME_PER_PERSON`) και επιλογή των 3 υψηλότερων και χαμηλότερων αντίστοιχα.
- Δεδομένα **εγκλημάτων** (μόνο του 2015), χωρικά συνενωμένα (`ST_Within(geom, geometry)`) στις 3 κοινότητες του Λος Άντζελες με το μεγαλύτερο και μικρότερο εισόδημα αντίστοιχα που υπολογίστηκαν παραπάνω.
- Δεδομένα αντιστοίχισης των κωδικών καταγωγής με την πλήρη περιγραφή τους

Βασική Προσέγγιση

1. **Πληθυσμός + Εισόδημα (Population+Income Join):**
   Όμοια με το **Query 3** έχουμε τον υπολογισμό του `AVERAGE_INCOME_PER_PERSON`. Στην συγκεκριμένη περίπτωση, μετά τον υπολογισμό αυτόν ταξινομούμε τις περιοχές(COMM) σε φθίνουσα(desc) και αύξουσα(asc) σειρά αντίστοιχα και επιλέγουμε τις 3 πρώτες περιοχές σε κάθε περίπτωση.
2. **Εγκλήματα + PopIncome Join (ST_Within):**
   Διαβάζουμε τα εγκλήματα (2010–σήμερα), κατασκευάζουμε `ST_Point(LON, LAT)` και φιλτράρουμε τα δεδομένα κρατώντας μόνο το 2015. Όπως και στο προηγούμενο query αναθέτουμε κάθε έγκλημα στην κατάλληλη υψηλόμισθη και χαμηλόμισθη περιοχή αντίστοιχα.
3. **CrimePopIncome + Race and Ethnicity codes Join:**
   Συνενώνουμε το αποτέλεσμα της προηγούμενης ανάθεσης με τον πίνακα φυλετικής καταγωγής για να εμφανίσουμε την κατάλληλη περιγραφή καταγωγής των θυμάτων. Στην συνέχεια κάνουμε aggragate με βάση την περιγραφή αυτή, μετρώντας των αριθμό των θυμάτων ανά φυλετική ομάδα.

Αποτελέσματα με Διαφορετικές Διαμορφώσεις Spark

Το ερώτημα εκτελέστηκε σε *AWS Spark* με **8 cores & 16GB** μνήμη συνολικά, σε τρεις διαφορετικές διαμορφώσεις:

1. 2 executors × **1 cores** / **2GB**,
2. 2 executors × **2 cores** / **4GB**,
3. 2 executors × **4 cores** / **8GB**.

Μετρούμενοι Χρόνοι Εκτέλεσης:

- **Διαμόρφωση 1**: 2 executors × 1c/2GB → 76.3s
- **Διαμόρφωση 2**: 2 executors × 2c/4GB → 53.6s
- **Διαμόρφωση 3**: 2 executors × 4c/8GB → 49.6s

Σχολιασμός Διαφορετικών Διαμορφώσεων

Οι τρεις διαμορφώσεις που δοκιμάσαμε (*2 executors ×1 cores/2GB*, *2 executors ×2 cores/4GB*, *2 executors ×4 core/8GB*) εφαρμόζουν κλιμάκωση στο σύνολο των υπολογιστικών πόρων που θα χρησιμοποιήσουμε κρατώντας σταθερό τον αριθμό των executors σε 2. Από τα αποτελέσματα βλέπουμε:

- **Διαμόρφωση 1 (1 cores/2GB):**
  Αυτή η διαμόρφωση περιορίζει σοβαρά τον παραλληλισμό, καθώς κάθε executor έχει μόνο 1 πυρήνα. Αυτό σημαίνει ότι μπορεί να επεξεργάζεται μόνο ένα task κάθε φορά, γεγονός που αυξάνει τον συνολικό χρόνο εκτέλεσης. Η περιορισμένη μνήμη (2GB) πιθανόν οδήγησε σε συχνό spilling στη δευτερεύουσα μνήμη (disk spilling) κατά την ένωση των μεγάλων datasets (`crime_data`, `income_data`, `zipcode_income`), ειδικά στις λειτουργίες `join` και `groupBy`. Η φάση `shuffling` (π.χ., κατά το`groupBy`) πιθανόν επιβαρύνθηκε λόγω των περιορισμένων πόρων.
- **Διαμόρφωση 2 (2 cores/4GB):**
  Η αύξηση των πυρήνων σε 2 ανά executor επέτρεψε την εκτέλεση περισσότερων tasks ταυτόχρονα, αυξάνοντας τον παραλληλισμό και μειώνοντας τον συνολικό χρόνο εκτέλεσης. Η διπλάσια μνήμη (4GB) μείωσε τις ανάγκες για disk spilling, επιταχύνοντας λειτουργίες όπως τα `join` και `groupBy`. Παρ' όλα αυτά, η διάρκεια παραμένει υψηλή λόγω του μεγάλου όγκου δεδομένων και των πολύπλοκων λειτουργιών (π.χ., `ST_Within` για γεωχωρικές συγκρίσεις, που απαιτούν σημαντικούς υπολογιστικούς πόρους).
- **Διαμόρφωση 3 (4 cores/8GB):**
  Αν και η διαμόρφωση πέτυχε τον μικρότερο χρόνο εκτέλεσης, η βελτίωση ήταν μικρότερη σε σχέση με τη μετάβαση από τη **Διαμόρφωση 1** στη **Διαμόρφωση 2**, σχεδόν αμελητέα. Αυτό πιθανότατα οφείλεται:
  - Στους περιορισμούς παραλληλισμού λόγω του αριθμού των partitions και του τρόπου που εκτελούνται οι γεωχωρικές λειτουργίες (`ST_Within`).
  - Στη μείωση της επίδρασης της αυξημένης μνήμης (8GB), καθώς τα 4GB στη Διαμόρφωση 2 ήταν πιθανόν επαρκή.
  - Σε περιορισμούς από shuffles και bottlenecks που δεν βελτιώνονται περαιτέρω με επιπλέον πυρήνες.

Συμπέρασμα:
Στο συγκεκριμένο query η αύξηση των διαθέσιμων πόρων (πυρήνες και μνήμη) βελτίωσε την απόδοση, ωστόσο παρατηρήθηκε ένα σημείο κορεσμού πέρα από το οποίο οι βελτιώσεις ήταν περιορισμένες. Αυτός ο περιοριστικός παράγοντας μπορεί να είναι το ποσοστό εφικτού παραλληλισμού (Amdahl's law), η απόδοση των λειτουργιών shuffle και η απαιτητικότητα των γεωχωρικών υπολογισμών.

Query 5: Ανάλυση Πλησιέστερου Αστυνομικού Τμήματος

Στόχος

Στο Query 5 στόχος είναι η αντιστοίχιση κάθε εγκλήματος (crime) στο *πλησιέστερο* αστυνομικό τμήμα (*police station*), καθώς και ο υπολογισμός:

1. Του **αριθμού εγκλημάτων** που αντιστοιχούν στο καθένα (δηλαδή πόσα εγκλήματα βρίσκονται πλησιέστερα σε κάθε `police_division`),
2. Της **μέσης απόστασης** (*avg distance km*) από τα εγκλήματα που αντιστοιχούν εκεί.

Τελικό ζητούμενο είναι η εμφάνιση των τμημάτων ταξινομημένων σε φθίνουσα σειρά αριθμού εγκλημάτων (`crime_count`).

Βασική Υλοποίηση σε Επίπεδο DataFrame

1. Φόρτωση Δεδομένων

- **Crime Data**: Διαβάζουμε τα αρχεία εγκλημάτων (2010–2019, 2020–*present*), φιλτράροντας $\text{LAT} = 0$ ή $\text{LON} = 0$. Για κάθε έγκλημα δημιουργείται ένα `ST_Point(LON, LAT)` (`crime_point`).
- **Police Stations**: Διαβάζουμε τις καταχωρήσεις τμημάτων (`LA_Police_Stations.csv`) και δημιουργούμε `ST_Point(X, Y)` (`station_point`) από τις στήλες `X`, `Y`.

2. Cross Join & Απόσταση Για να υπολογιστεί η απόσταση κάθε εγκλήματος από όλα τα πιθανά τμήματα, εκτελείται:

- *cross join* ($\#Crimes \times \#Stations$), και
- `ST_DistanceSphere(crime_point, station_point)/1000` ώστε να προκύψει η απόσταση σε χιλιόμετρα (`distance_km`).

3. Εντοπισμός Πλησιέστερου Τμήματος (Window Function) Καθώς κάθε έγκλημα εμφανίζεται πολλαπλές φορές (μία ανά *police station*), χρειάζεται να *επιλέξουμε μόνο* το τμήμα με τη μικρότερη απόσταση:

- Ορίζουμε `partitionBy("DR_NO").orderBy("distance_km")`.
- Με `row_number()` κατατάσσουμε τις σειρές του ίδιου `DR_NO` από τη μικρότερη έως τη μεγαλύτερη απόσταση.
- `filter` όπου `row_number` $= 1$ διατηρεί **μόνο** το κοντινότερο αστυνομικό τμήμα ανά έγκλημα.

4. Ομαδοποίηση & Τελική Ταξινόμηση Τέλος, ομαδοποιούμε (`groupBy("police_division")`) τα κοντινότερα εγκλήματα:

$$\texttt{crime\_count} = \text{count}(*), \quad \texttt{avg\_distance\_km} = \text{avg}(\text{distance\_km})$$

και ταξινομούμε σε φθίνουσα σειρά (`orderBy("crime_count", ascending=False)`). Έτσι προκύπτει, *ανά* `police_division`, πόσα εγκλήματα συσχετίζονται με αυτό (`crime_count`) και η μέση απόστασή τους (`avg_distance_km`).

Αποτελέσματα με Διαφορετικές Διαμορφώσεις Spark

Το ερώτημα εκτελέστηκε σε *AWS Spark* με **8 cores & 16GB** μνήμη συνολικά, σε τρεις διαφορετικές διαμορφώσεις:

1. **2 executors** $\times$ 4 cores / 8GB,
2. **4 executors** $\times$ 2 cores / 4GB,
3. **8 executors** $\times$ 1 core / 2GB.

Μετρούμενοι Χρόνοι Εκτέλεσης:

- **Διαμόρφωση 1**: 2 executors $\times$ 4c/8GB $\rightarrow$ 21.93s
- **Διαμόρφωση 2**: 4 executors $\times$ 2c/4GB $\rightarrow$ 24.38s
- **Διαμόρφωση 3**: 8 executors $\times$ 1c/2GB $\rightarrow$ 43.03s

Σχολιασμός Διαφορετικών Διαμορφώσεων

Οι τρεις διαμορφώσεις που δοκιμάσαμε (*2 executors $\times$4 cores/8GB*, *4 executors $\times$2 cores/4GB*, *8 executors $\times$1 core/2GB*) κατανέμουν τους ίδιους συνολικούς πόρους (8 cores, 16GB) με διαφορετικό τρόπο. Από τα αποτελέσματα βλέπουμε:

- **Διαμόρφωση 1 (2 executors, 4c/8GB):** Παρουσιάζει τον **μικρότερο χρόνο εκτέλεσης** (περίπου 21.9s). Σε αυτήν τη ρύθμιση, κάθε executor διαθέτει *επαρκή μνήμη* (8GB) για να επεξεργαστεί μεγάλο όγκο δεδομένων, ενώ οι 4 πυρήνες (`cores`) επιτρέπουν την εκτέλεση πολλαπλών tasks σε παράλληλα threads *εντός* του ίδιου executor. Σημαντικό είναι επίσης ότι *μόλις 2 executors* έχουν *μικρότερο overhead* στην επικοινωνία μεταξύ τους (fewer network/data transfers).
- **Διαμόρφωση 2 (4 executors, 2c/4GB):** Εδώ τα resources είναι μέτρια κατανεμημένα: ο κάθε executor έχει 4GB μνήμη και 2 πυρήνες. Ο χρόνος εκτέλεσης (24.38s) είναι μεγαλύτερος από την πρώτη διαμόρφωση αλλά αρκετά μικρότερος από την τρίτη. Ο λόγος είναι ότι *αυξάνεται* η ανάγκη επικοινωνίας μεταξύ των executors (4 executors αντί 2) και *μειώνεται* ο διαθέσιμος παραλληλισμός *εντός* του κάθε executor σε σχέση με την πρώτη (2 αντί 4 cores).

- **Διαμόρφωση 3 (8 executors, 1c/2GB):** Παρουσιάζει **σημαντική αύξηση** στον χρόνο (περίπου 43.03s). Παρά το ότι έχουμε 8 executors συνολικά, καθένας έχει *μόλις 1 πυρήνα* και *2GB* μνήμης. Αυτό συνεπάγεται:

  1. **Περιορισμένο παραλληλισμό εντός κάθε executor**: Ένας μόνο πυρήνας δεν επιτρέπει την εκτέλεση πολλαπλών tasks ταυτόχρονα.
  2. **Μικρή μνήμη**: Τα 2GB μπορεί να μην επαρκούν για το cross join ($\#Crimes \times \#Stations$) και το window function, προκαλώντας αρκετό I/O overhead (spill στο δίσκο).
  3. **Υψηλό overhead συντονισμού**: Με 8 executors, το Spark δαπανά περισσότερους πόρους στην επικοινωνία μεταξύ πολλών κόμβων, ειδικά όταν τα partitioned data μεταφέρονται, όπως για παράδειγμα στο shuffle.

Συμπέρασμα: Στο συγκεκριμένο σενάριο, η *πρώτη* διαμόρφωση (2 executors $\times$4c/8GB) συνδυάζει επαρκή μνήμη και πυρήνες ανά executor, ώστε να αποφεύγονται τόσο οι εσωτερικοί περιορισμοί μνήμης όσο και το μεγάλο overhead πολλαπλών executors. Αντίθετα, η τρίτη διαμόρφωση με *πολλούς μικρούς* executors προκαλεί *αυξημένο overhead* και ενδέχεται να μην αξιοποιεί πλήρως τους πόρους σε παράλληλη εκτέλεση.

Τελικό Αποτέλεσμα

Από την παραπάνω διαδικασία (window function), παράγεται ένας πίνακας *ανά* `police_division` με:

- `avg_distance_km`: μέση απόσταση από τα εγκλήματα που βρίσκονται πλησιέστερα στο συγκεκριμένο τμήμα,
- `crime_count`: το πλήθος των εγκλημάτων αυτών,

ταξινομημένα σε *φθίνουσα* σειρά ως προς `crime_count`. Δείγμα εξόδου:

police_division	avg_distance_km	crime_count
77TH STREET	2.208	7045
RAMPART	2.009	4595
FOOTHILL	3.597	3047
PACIFIC	2.739	2132
...	...	...

Κώδικας και Αποτελέσματα

Οι σχετικές υλοποιήσεις για κάθε ερώτημα και τα αντίστοιχα αποτελέσματα είναι διαθέσιμα στο GitHub.