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

Κωνσταντίνος Κατσικόπουλος (03120103)
Χαρίδημος Παπαδάκης (03120022)

Ιανουάριος 2025

1 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:

1.1 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.

1.2 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.

1.3 Σύγκριση Επίδοσης & Συμπεράσματα

– **Αποτελέσματα**:

```
DataFrame Time: 18.40
DataFrame Results:
Adults: 121093
Young Adults: 33605
Children: 15928
Seniors: 5985

RDD Time: 19.67
RDD Results:
Adults: 121093
Young Adults: 33605
Children: 15928
Seniors: 5985
```

– **Χρόνοι εκτέλεσης:** Στην τελευταία δοκιμή, η υλοποίηση με χρήση DataFrames ολοκληρώθηκε σε $\sim$18.4s, ενώ με χρήση RDDs σε $\sim$19.7s. 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.

2 Query 3: Ανάλυση Εισοδήματος & Εγκληματικότητας

2.1 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`).

2.2 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 ένα 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

$$\texttt{ST\_Within}(\text{geom}, \text{geometry})$$

αναθέτουμε κάθε έγκλημα στην κατάλληλη περιοχή. Τέλος, ομαδοποιούμε για να βρούμε το συνολικό πλήθος εγκλημάτων (`TOTAL_CRIMES`) και υπολογίζουμε:

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

2.3 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.

2.3.1 1. Population + Income Join (ZIPCODE)

Στρατηγικές:

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

Χρόνοι Εκτέλεσης:

- `broadcast`: 13.75s
- `shuffle_hash`: 12.55s
- `shuffle_replicate_nl`: 12.84s
- `merge`: 13.38s

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.

2.3.2 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`: 13.11s (*BroadcastIndexJoin*)
- `shuffle_hash`: 13.26s (*RangeJoin*)
- `shuffle_replicate_nl`: 13.34s (*RangeJoin*)
- `merge`: 12.99s (*RangeJoin* αντί SortMergeJoin)

Και σε αυτό το χωρικό join, οι χρόνοι διαφέρουν πολύ λίγο (0.35s). 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.

2.4 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.

3 Query 5: Ανάλυση Πλησιέστερου Αστυνομικού Τμήματος

3.1 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`).

3.2 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`).

3.3 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 $\rightarrow$ 40.38s

3.3.1 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* σημαίνει *λιγότερο 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* ο συντονισμός 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 40.4s). 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 πόλλους 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.

3.4 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
...	...	...