Προχωρημένα Θέματα Βάσεων Δεδομένων
Εξαμηνιαία Εργασία

Ομάδα 16
Αρετή Μέη: 03120062
Ελισάβετ Παπαδοπούλου : 03120190

Github Repository

Μπορείτε να βρείτε τον κώδικα των υλοποιήσεών μας στον παρακάτω σύνδεσμο:
GitHub Repository.

1 Εισαγωγή

Αντικείμενο της παρούσας εξαμηνιαίας εργασίας είναι η ανάλυση σε μεγάλα σύνολα δεδομένων, εφαρμόζοντας επεξεργασία με τεχνικές που εφαρμόζονται σε Data Science Projects. Στα πλαίσια του project, χρησιμοποιήθηκαν τα εργαλεία Apache Hadoop (versio n¿= 3.0) και Apache Spark (version ¿= 3.5). Η εργασία εκτελέστηκε σε ένα ειδικά διαμορφωμένο περιβάλλον στο AWS cloud.

2 Δεδομένα

Τα σύνολα δεδομένων που χρησιμοποιήθηκαν είναι διαθέσιμα και δωρεάν, και έχουν περισυλλεγεί από διαφορετικές πηγές. Λίγα λόγια για τα data-sets παρατίθενται παρακάτω:

2.1 Βασικό data-set: Los Angeles Crime Data

Το βασικό σύνολο δεδομένων το οποίο καλούμαστε να χρησιμοποιήσουμε περιλαμβάνει δεδομένα καταγραφής εγκλημάτων για το Los Angeles, από το 2010 μέχρι και σήμερα. Πρόκειται, συγκεκριμένα, για δύο .csv αρχεία, ένα για τα εγκλήματα που διαδραματίστηκαν από το 2010 ως το 2019, και ένα από το 2020 εώς και το 2023.

2.2 LA Police Stations

Πρόκειται για ένα μικρό σύνολο δεδομένων, που περιέχει τις πληροφορίες σχετικά με την τοποθεσία των 21 αστυνομικών τμημάτων που βρίσκονται στην πόλη του Los Angeles.

2.3 2010 Census Blocks (Los Angeles County)

Πρόκειται για ένα σύνολο δεδομένων απογραφικών στοιχείων, αναφορικά με την Κομητεία του Los Angeles για το έτος 2010 σε geojson format. Συνοδεύεται από το αρχείο που περιγράφει τα πεδία του.

2.4 Median Household Income by Zip Code (Los Angeles County)

Άλλο ένα μικρό σύνολο δεδομένων, που περιέχει πληροφορίες σχετικά με το μέσο εισόδημα ανά νοικοκυριό και ταχυδρομικό κώδικα στην Κομητεία του Los Angeles. Το συγκεκριμένο σύνολο παρήχθηκε με βάση τα αποτελέσματα της απογραφής του έτους 2015.

2.5 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.

3 Ζητούμενα

3.1 Query 1

Υλοποίηση

Το Query 1 ζητάει την ταξινόμηση, σε φθίνουσα σειρά, των ηλικιακών ομάδων των θυμάτων σε περιστατικά που περιλαμβάνουν οποιαδήποτε μορφή "βαριάς σωματικής βλάβης". Οι ηλικιακές ομάδες που εξετάζονται είναι:

- Παιδιά: < 18
- Νεαροί ενήλικοι: 18 – 24

- Ενήλικοι: 25 – 64
- Ηλικιωμένοι: > 64

Η υλοποίηση του Query 1 περιλαμβάνει τα εξής βήματα:

1. **Φόρτωση δεδομένων:** Φορτώνονται δύο αρχεία CSV, που περιέχουν δεδομένα εγκλημάτων από το 2010 έως το 2019 και από το 2020 έως το παρόν, αντίστοιχα.
2. **Συνένωση δεδομένων:** Τα δύο σύνολα δεδομένων ενώνονται σε ένα ενιαίο DataFrame ή RDD, εξαιρώντας εγγραφές που περιέχουν μηδενικές συντεταγμένες (LAT, LON).
3. **Φιλτράρισμα περιστατικών:** Φιλτράρονται μόνο τα περιστατικά που περιλαμβάνουν τη φράση `\ aggravated assault "` στη στήλη `Crm Cd Desc`.
4. **Ομαδοποίηση ηλικιών:** Χρησιμοποιώντας την τιμή της στήλης `Vict Age`, τα θύματα ταξινομούνται στις εξής ηλικιακές ομάδες:
   - Παιδιά, αν $1 \leq$ `Vict Age` < 18.
   - Νεαροί ενήλικοι, αν $18 \leq$ `Vict Age` ≤ 24.
   - Ενήλικοι, αν $25 \leq$ `Vict Age` ≤ 64.
   - Ηλικιωμένοι, αν `Vict Age` ≥ 65.

   Εγγραφές με μη έγκυρες ηλικίες κατατάσσονται στην κατηγορία `Unknown`, η οποία στη συνέχεια εξαιρείται από την ανάλυση.
5. **Αριθμός περιστατικών ανά ομάδα:** Υπολογίζεται ο συνολικός αριθμός περιστατικών για κάθε ηλικιακή ομάδα, και τα αποτελέσματα ταξινομούνται σε φθίνουσα σειρά με βάση τον αριθμό περιστατικών.

Αποτελέσματα και Σχολιασμός

Τα αποτελέσματα που προέκυψαν από το Query φαίνονται παρακάτω:

Table 1: Query 1 Results

Age Group	Incident Count
Adults	121052
Young Adults	33588
Children	10825
Elderly	5985

Σύγκριση Dataframe - RDD APIs

Στο πρώτο ερώτημα καλούμαστε να υλοποιήσουμε το Query 1 χρησιμοποιώντας τόσο **DataFrame**, αλλά και **RDD APIs**. Οι δύο υλοποιήσεις εκτελέστηκαν με **4 Spark Executors**. Προτού οδηγηθούμε στον σχολιασμό των χρόνων εκτελέσεων των δύο μεθόδων, παραθέτουμε θεωρητικές γνώσεις σχετικές με τα δύο APIs.

1. **RDD API**: Παρέχει χαμηλού επιπέδου αφαίρεση για τα κατανεμημένα δεδομένα. Ο χρήστης πρέπει να ορίζει ρητά τους μετασχηματισμούς και τις ενέργειες, ενώ διαχειρίζεται και τη σειριοποίηση και βελτιστοποίηση μόνος του. Έτσι, προσφέρει μεγαλύτερη ευελιξία, αλλά απαιτεί περισσότερη προσπάθεια και γραφή κώδικα. Όσον αφορά την απόδοση, δεν εκμεταλλεύεται το **Query Optimization**, και κάθε ενέργεια εκτελείται όπως ορίζεται, χωρίς προσαρμογές, κάτι που μπορεί να οδηγήσει σε λιγότερο αποδοτική εκτέλεση. Γενικά, η χρήση του ενδείκνυται για αρκετά πολύπλοκες λειτουργίες που απαιτούν λεπτομερή έλεγχο των δεδομένων ή όταν χρησιμοποιούνται προσαρμοσμένες μορφές δεδομένων, ωστόσο έχει μεγαλύτερες απαιτήσεις σε μνήμη και CPU.
2. **DataFrame API**: Είναι πιο φιλικό προς το χρήστη, ειδικά αν έχει εξοικείωση με SQL, καθώς επιτρέπει την υλοποίηση Queries παρόμοιων με αυτά των βάσεων δεδομένων. Εκμεταλλεύεται τον Catalyst Optimizer και την Tungsten Execution Engine, οι οποίοι:

(a) βελτιστοποιούν το πλάνο εκτέλεσης των ερωτημάτων,

(b) χρησιμοποιούν πιο αποδοτικούς αλγορίθμους για φιλτράρισμα, συνενώσεις (joins) και ομαδοποιήσεις (aggregations)

(c) και μπορούν να αποθηκεύσουν δεδομένα σε μορφή στηλών (columnar storage) για καλύτερη απόδοση.

Έτσι, το **DataFrame API** καθίσταται ταχύτερο και πιο αποδοτικό για μεγάλους όγκους δεδομένων.

Οι παραπάνω γνώσεις μας επιβεβαιώνουν τις παρατηρήσεις για τους χρόνους εκτέλεσης που απαιτεί το κάθε API για το Query 1, οι οποίοι φαίνονται παρακάτω:

Table 2: Query 1 Execution Time Results

API	**Execution Time**
DatFrame API	29.42493221130371 seconds
RDD API	38.487494346466064 seconds

3.2 Query 2

α. Ερώτημα

Στο συγκεκριμένο ερώτημα καλούμαστε να υλοποιήσουμε το Query 2, χρησιμοποιώντας τόσο **DataFrame** αλλά και **SQL API**. Παρακάτω σχολιάζεται η υλοποίηση, καθώς επίσης και τα αποτελέσματα των εκτελέσεών μας:

Υλοποίηση

Το Query 2 ζητάει να υπολογιστούν τα τρία Αστυνομικά Τμήματα με το υψηλότερο ποσοστό κλεισμένων (περατωμένων) υποθέσεων για κάθε έτος. Τα αποτελέσματα πρέπει να περιλαμβάνουν το έτος, τα ονόματα των τμημάτων, τα ποσοστά τους, και την κατάταξή τους (ranking). Τα αποτελέσματα εμφανίζονται σε αύξουσα σειρά έτους και ranking.

Η υλοποίηση περιλαμβάνει τα εξής βήματα:

1. **Φόρτωση δεδομένων:** Φορτώνονται τα αρχεία CSV που περιέχουν δεδομένα για τα Αστυνομικά Τμήματα και τα εγκλήματα. Τα δεδομένα συνενώνονται σε ένα ενιαίο DataFrame, εξαιρώντας εγγραφές με μηδενικές συντεταγμένες (LAT, LON).

2. **Προσθήκη στήλης έτους:** Προστίθεται μια στήλη που υπολογίζει το έτος κάθε περιστατικού, με βάση τη στήλη `DATE OCC`.

3. **Φιλτράρισμα κλεισμένων υποθέσεων:** Εγγραφές με τιμή `UNK` ή `Invest Cont` στη στήλη `Status Desc` θεωρούνται μη περατωμένες και εξαιρούνται.

4. **Υπολογισμός περιστατικών:** Υπολογίζεται ο συνολικός αριθμός περιστατικών (`TotalCases`) και ο αριθμός των κλεισμένων υποθέσεων (`ClosedCases`) ανά έτος και τμήμα.

5. **Υπολογισμός ποσοστού:** Προστίθεται στήλη `ClosedCasePercentage`, που υπολογίζεται ως:

$$\texttt{ClosedCasePercentage} = \frac{\texttt{ClosedCases}}{\texttt{TotalCases}} \times 100$$

6. **Κατάταξη:** Χρησιμοποιείται η λειτουργία `ROW_NUMBER()` για την κατάταξη των τμημάτων με βάση το ποσοστό κλεισμένων υποθέσεων, ανά έτος.

7. **Φιλτράρισμα τριών κορυφαίων τμημάτων:** Διατηρούνται μόνο οι τρεις κορυφαίες καταχωρίσεις ($\texttt{Rank} \leq 3$) για κάθε έτος.

8. **Παρουσίαση αποτελεσμάτων:** Τα αποτελέσματα ταξινομούνται σε αύξουσα σειρά έτους και κατάταξης.

Αποτελέσματα και Σχολιασμός

Τα αποτελέσματα που προέκυψαν από το Query φαίνονται παρακάτω:

Σύγκριση Dataframe - SQL APIs

Αρχικά, παρατίθενται ορισμένες θεωρητικές γνώσεις, αναφορικά με τα δύο διαφορετικά APIs.

1. **Dataframe API**: Κάθε μετασχηματισμός προστίθεται στο logical query plan, το οποίο στη συνέχεια βελτιστοποιείται από το Spark. Απαιτεί περισσότερα βήματα, όπως withColumn, filter, groupBy και join, για να φτάσει στο επιθυμητό αποτέλεσμα.

2. **SQL API**: Όλη η λογική δηλώνεται εκ των προτέρων σε ένα ερώτημα ενώ μειώνεται ο επιπλέον φόρτος από πολλαπλούς μετασχηματισμούς όταν αντιμετωπίζουμε σύνθετες διαδικασίες.

Και τα δύο APIs περνούν από τον ίδιο βελτιστοποιητή (Catalyst optimizer), επομένως η απόδοση είναι σχεδόν ίδια. Σε ορισμένες περιπτώσεις, είναι πιο αποδοτικό το DataFrame API για σύνθετες λειτουργίες, καθώς προσφέρει καλύτερη διαχείριση μεθόδων και ελέγχου στο πρόγραμμα, ενώ είναι πιο ευέλικτο, καθώς μπορεί να χρησιμοποιηθεί για πιο σύνθετους αλγορίθμους. Από την άλλη, το SQL API είναι κατάλληλο για ερωτήματα που βασίζονται μόνο σε SQL, αλλά σε σύνθετες λειτουργίες μπορεί να υστερεί.

Table 3: Closed Case Percentage by Year and Division

Year	AREA NAME	Closed Case Percentage	Rank
2010	Rampart	32.85090742017	1
2010	Olympic	31.515289821999087	2
2010	Harbor	29.36028339237341	3
2011	Olympic	35.03192688118192	1
2011	Rampart	32.500296103280824	2
2011	Harbor	28.516260162601625	3
2012	Olympic	34.295435879385195	1
2012	Rampart	32.461037450569904	2
2012	Harbor	29.534834324553948	3
2013	Olympic	33.58217940999398	1
2013	Rampart	32.1060382916053	2
2013	Harbor	29.70696405267529	3
2014	Van Nuys	32.00295639320029	1
2014	West Valley	31.512710797885727	2
2014	Mission	31.21740874448456	3
2015	Van Nuys	32.265140677157845	1
2015	Mission	30.466622852314334	2
2015	Foothill	30.353001803658852	3
2016	Van Nuys	32.1880650994575	1
2016	West Valley	31.404702970297027	2
2016	Foothill	29.908647228131645	3
2017	Van Nuys	32.04915003695491	1
2017	Mission	31.055387158996968	2
2017	Foothill	30.469700657094183	3
2018	Foothill	30.731346958877126	1
2018	Mission	30.730034296913278	2
2018	Van Nuys	28.905206942590123	3
2019	Mission	30.719878207916484	1
2019	West Valley	30.57529223011689	2
2019	Foothill	29.208050182958704	3
2020	West Valley	30.804455445544555	1
2020	Mission	30.328549401020044	2
2020	Harbor	29.780741410488247	3
2021	Mission	30.555555555555557	1
2021	West Valley	29.05611645926274	2
2021	Foothill	28.19376964847099	3
2022	West Valley	26.536367172306498	1
2022	Harbor	26.337538060026098	2
2022	Topanga	26.236786469344608	3
2023	Foothill	26.750524109014673	1
2023	Topanga	26.538022616453986	2
2023	Mission	25.662731120516817	3
2024	Foothill	18.667786174083403	1
2024	77th Street	17.6147382029735	2
2024	Mission	17.187827911857294	3

Παρακάτω παρατίθενται οι τελικοί χρόνοι εκτέλεσης για τα δύο APIs:

Table 4: Query 2a Execution Time Results

API	Execution Time
DatFrame API	2.2361557483673096 seconds
SQL API	2.17 seconds

Και τα δύο APIs καταλήγουν σε παρόμοια πλάνα εκτέλεσης, καθώς μεταφράζονται στο εσωτερικό λογικό μοντέλο του Spark. Οι διαφορές που ενδέχεται να προκύψουν οφείλονται στα ενδιάμεσα στάδια που δημιουργούνται από τα πολλά βήματα του DataFrame API. Παρατηρούμε στο ερώτημά μας ότι η διαφορά μεταξύ των χρόνων δεν είναι μεγάλη. Συμπεραίνουμε ότι η επιλογή ανάμεσα στο **DataFrame API** και το **SQL API** εξαρτάται από:

- Την εξοικείωση του χρήστη με προγραμματισμό ή SQL.
- Την πολυπλοκότητα των εργασιών.
- Τις απαιτήσεις του έργου σε ευελιξία και συντήρηση.

Και τα δύο APIs είναι εξίσου αποδοτικά, αλλά το **DataFrame API** είναι πιο ισχυρό και προσαρμόσιμο σε περίπλοκα σενάρια.

β. Ερώτημα

Στο δεύτερο ερώτημα μας ζητείται η συγγραφή κώδικα **Spark**, ο οποίος μετατρέπει το κυρίως data set σε parquet file format και αποθηκεύει ένα μοναδικό **.parquet αρχείο** στο S3 bucket της ομάδας μας. Στην συνέχεια, καλούμαστε να συγκρίνουμε τους χρόνους της εκτέλεσης της εφαρμογής μας, για την περίπτωση που τα δεδομένα εισάγονται σαν **.csv** και σαν **.parquet**.

Επιλέξαμε την υλοποίηση του **Dataframe API** για να πειραματιστούμε με το .parquet τύπο αρχείων και έχουμε:

1. **CSV**: Τα αρχεία CSV είναι λιγότερο αποδοτικά σε ταχύτητα ανάγνωσης/εγγραφής λόγω της απουσίας βελτιστοποιημένης μορφοποίησης. Η ανάγνωση δεδομένων από CSV απαιτεί χρόνο για την αναγνώριση τύπων δεδομένων και τη διαχείριση του text parsing.
2. **Parquet**: Το Parquet είναι μία column-based μορφή αρχείου που συμπιέζει δεδομένα και βελτιστοποιεί την ανάγνωση συγκεκριμένων στηλών. Η εγγραφή σε Parquet είναι αργή λόγω της συμπίεσης, αλλά η ανάγνωση είναι σημαντικά ταχύτερη.

Οι παρατηρήσεις μας επιβεβαιώνονται από τα αποτελέσματα που πήραμε, όπου και βλέπουμε σημαντική μείωση του χρόνου, και συγκεκριμένα μια ολόκληρη τάξη μεγέθους.

Table 5: Query 2a Execution Time Results

File Format	Execution Time
Parquet	0.36 seconds
CSV	2.2361557483673096 seconds

3.3 Query 3

Η υλοποίηση του Query 3 αφορά τον υπολογισμό του μέσου ετήσιου εισοδήματος ανά άτομο και της αναλογίας συνολικού αριθμού εγκλημάτων ανά άτομο για κάθε περιοχή του Los Angeles, με βάση δεδομένα απογραφής πληθυσμού του 2010 και εισοδήματος ανά νοικοκυριό του 2015.

Δεδομένα

Χρησιμοποιήθηκαν τα παρακάτω datasets και τα συγκεκριμένα κελιά τους:

- **Crime Data**: Περιέχει δεδομένα εγκλημάτων, με σημαντικές στήλες:
  - `LAT`, `LON`: Γεωγραφικές συντεταγμένες.
  - `COMM`: Όνομα περιοχής.
  - `ZCTA10`: Ταχυδρομικός κώδικας.
- **LA Income Data**: Περιέχει δεδομένα εισοδήματος ανά νοικοκυριό του 2015, με σημαντικές στήλες:
  - `Community`: Όνομα περιοχής.
  - `Zip Code`: Ταχυδρομικός κώδικας.
  - `Estimated Median Income`: Μέσο εισόδημα ανά νοικοκυριό.
- **2010 Census Blocks**: Περιέχει δεδομένα απογραφής πληθυσμού και κατοικιών, με σημαντικές στήλες:
  - `COMM`: Όνομα περιοχής.
  - `ZCTA10`: Ταχυδρομικός κώδικας.
  - `POP_2010`: Συνολικός πληθυσμός.
  - `HOUSING10`: Συνολικές κατοικίες.

Βήματα Υλοποίησης

1. **Φόρτωση και Καθαρισμός Δεδομένων**:
   - Τα δεδομένα φορτώνονται από αρχεία `CSV` και `GeoJSON`.
   - Το πεδίο `Estimated Median Income` καθαρίζεται από ειδικούς χαρακτήρες ('$', ',') και μετατρέπεται σε αριθμητικό τύπο.
2. **Ομαδοποίηση Πληθυσμού και Κατοικιών**: Ο πληθυσμός και οι κατοικίες ομαδοποιούνται ανά περιοχή (`COMM`) και ταχυδρομικό κώδικα (`ZCTA10`) για τον υπολογισμό των συνολικών τιμών (`total_population`, `total_households`).
3. **Ενοποίηση Δεδομένων Εισοδήματος και Πληθυσμού**: Τα δεδομένα εισοδήματος και πληθυσμού ενώνονται με βάση το όνομα περιοχής (`COMM`) και τον ταχυδρομικό κώδικα (`ZCTA10`).
4. **Γεωχωρική Ενοποίηση Δεδομένων Εγκλημάτων**: Τα δεδομένα εγκλημάτων ενσωματώνονται χρησιμοποιώντας γεωχωρική ένωση (`ST_Within`), αντιστοιχίζοντας εγκλήματα στις αντίστοιχες περιοχές με βάση τις γεωγραφικές συντεταγμένες (`LAT`, `LON`).
5. **Υπολογισμοί**:
   - Το μέσο εισόδημα ανά άτομο υπολογίζεται ως:

$$\texttt{income\_per\_person} = \frac{\texttt{total\_households} \times \texttt{median\_income}}{\texttt{region\_total\_population}}$$

   - Η αναλογία εγκλημάτων ανά άτομο υπολογίζεται ως:

$$\texttt{crimes\_per\_person} = \frac{\texttt{total\_crime\_number}}{\texttt{region\_total\_population}}$$

6. **Συγκέντρωση και Ταξινόμηση Αποτελεσμάτων**: Τα αποτελέσματα συγκεντρώνονται σε έναν πίνακα με τις στήλες:

- `region`: Όνομα περιοχής.
- `income_per_person`: Μέσο εισόδημα ανά άτομο.
- `crimes_per_person`: Αναλογία εγκλημάτων ανά άτομο.

Ο πίνακας ταξινομείται ανάλογα με την αναλογία εγκλημάτων ή το εισόδημα και φαίνεται παρακάτω, κρατώντας ενδεικτικά τις πέντε περιοχές με μεγαλύτερο crime ratio και τις πέντε με μεγαλύτερο income per person.

Table 6: Income Per Person and Crimes Per Person by Region

Region	**Income Per Person**	**Crimes Per Person**
Vernon	4406.45	6.34
Downtown	19107.38	4.21
Little Tokyo	23815.34	4.02
Hollywood	25648.05	1.55
Chinatown	14058.46	1.26

Table 7: Income Per Person and Crimes Per Person by Region (Top Regions)

Region	**Income Per Person**	**Crimes Per Person**
Marina del Rey	76428.85	0.0792
Century City	73014.41	0.8558
Pacific Palisades	72058.72	0.4509
Malibu	67132.45	7.91×10^{-5}
Marina Peninsula	65235.69	0.6124

Hint & Explain

Ο Catalyst Optimizer του Spark επιλέγει στρατηγικές `join` βασισμένος σε διάφορους παράγοντες, όπως το μέγεθος των δεδομένων, τη διαθέσιμη μνήμη, τη θέση των δεδομένων, και τα μεταδεδομένα των στηλών. Παρακάτω παραθέτουμε τις επιλογές του σε καθένα από τα τέσσερα διαφορετικά join που εκτελέσαμε, όπως αυτά φαίνονται στο πεδίο **Physical Plan** του **Explain** που εκτελέσαμε για το προϊόν έκαστου join.

1. **comm_pop with la_income_clean**
   Κάνουμε join τα:

   - Flattened δεδομένα του dataset **2010_Census_Blocks_fields.csv**, έχοντας πρώτα απαλείψει τις καταχωρήσεις με μηδενικό housing ή μηδενικό πληθυσμό, έχουμε ομαδοποιήσει με βάση το ζευγάρι **("COMM", "ZCTA10")**, και για τα οποία έχουμε υπολογίσει τον συνολικό αριθμό **households**, και **population**,
   - με το καθαρισμένο από τα σύμβολα "$" και "," **LA_income_2015.csv** dataset.

   Η τεχνική που επιλέχθηκε: **BroadcastHashJoin**

2. **crime_data_geom with flattened_blocks_df**
   Κάνουμε join τα:

   - Δεδομένα του union των dataset **Crime_Data_from_2010_to_2019_20241101.csv** - **Crime_Data_from_2020_to_Present_20241101.csv**, έχοντας πρώτα αφαιρέσει τις καταχωρήσεις με γεωγραφική θέση **(LON, LAT) = (0,0)** και έχοντας προσθέσει ένα column με όνομα geom, το οποίο προκύπτει από την συνάρτηση **ST_Point** και αναπαριστά την γεωγραφική θέση όπου διαδραματίστηκε η κάθε καταχώρηση.
   - με τα Flattened δεδομένα του dataset **2010_Census_Blocks_fields.csv**, έχοντας πρώτα απαλείψει τις καταχωρήσεις με μηδενικό housing ή μηδενικό πληθυσμό.

   Η τεχνική που επιλέχθηκε: **RangeJoin**

3. **income_result with crime_result**
   Κάνουμε join τα:

- Δεδομένα που έχουν προκύψει από το προηγούμενο join, αφού αυτά ομαδοποιηθούν κατά **("COMM", "ZCTA10")** και προσμετρηθούν τα εγκλήματα που αναλογούν σε αυτά τα ζεύγη, υπολογιστούν τα εγκλήματα και οι συνολικοί πληθυσμοί ανά περιοχή (COMM) και στην συνέχεια προστεθεί η στήλη "crimes_per_person" με την διαίρεση τους.
- με το αποτέλεσμα του πρώτου join, αφού σε αυτό υπολογιστούν το συνολικό εισόδημα και ο συνολικός πληθυσμός ανά περιοχή, και προστεθεί μία στήλη με την διαίρεσή τους.

Η τεχνική που επιλέχθηκε: **SortMergeJoin**

Σχολιασμός

Για να εξετάσουμε εναλλακτικές στρατηγικές, χρησιμοποιήσαμε `hints` για να αναγκάσουμε τον Optimizer να υλοποιήσει διαφορετικά `join strategies`. Αρχικά, θα κάνουμε μια θεωρητική αναφορά σε κάθε μία από τις τέσσερις (4) ζητούμενες στρατηγικές **join**:

- **Broadcast Join**: Πρόκειται για μια βελτιστοποιημένη στρατηγική join, όπου ένας από τους δύο πίνακες είναι αρκετά μικρός για να γίνει broadcast σε όλα τα nodes του cluster. Το Spark στέλνει το μικρότερο dataset σε όλους τους επεξεργαστές, επιτρέποντας να γίνει join τοπικά σε κάθε κόμβο χωρίς να χρειαστεί shuffling του μεγαλύτερου dataset. Συγκεκριμένα:

  1. Το μικρότερο dataset στέλνεται με broadcast σε όλους τους κόμβους.
  2. Κάθε κόμβος επεξεργάζεται το δικό του partition του μεγαλύτερου dataset, και το ταιριάζει με το αντίγραφό του του μικρότερου join που βρίσκεται στην μνήμη του.
  3. Τα αποτελέσματα όλων των κόμβων συνδυάζονται.

  Η συγκεκριμένη στρατηγική παρουσιάζει υψηλή απόδοση σε περιπτώσεις που το ένα dataset είναι χαρακτηριστικά μικρότερο από το άλλο, και αρκετά μικρό ώστε να μπορεί να γίνει broadcast γρήγορα. Η αποφυγή της ανάγκης για shuffle μειώνει σημαντικά τα I/O στο network, ενώ είναι ιδανικό για skewed datasets, όταν το μικρότερο dataset χωράει στην μνήμη. Προϋπόθεση ωστόσο, για την καλή του απόδοση, είναι να χωράει το μικρότερο dataset στις μνήμες των επεξεργαστών.

- **Sort-Merge Join**: Πρόκειται για την στρατηγική που προτιμάται σε δύο μεγάλα datasets, ταξινομημένα στο join key. Συγκεκριμένα:

  1. Το Spark κάνει shuffle και sort τα datasets με βάση το join key.
  2. Μετά την ταξινόμηση, τα datasets σκανάρονται παράλληλα, και τα κλειδιά που ταιριάζουν συγχωνεύονται.
  3. Η φάση του merge συμπεριλαμβάνει μια γραμμική αναζήτηση στα ταξινομημένα datasets, καθιστώντας την αποδοτική για μεγάλα datasets.

  Η συγκεκριμένη στρατηγική είναι αποδοτική για μεγάλα datasets, με ταξινομημένα κλειδιά join. Διαχειρίζεται επίσης πολύ καλά τα join που βασίζονται στην ισότητα των κλειδιών των δύο datasets. Ωστόσο, απαιτεί ένα κοστοβόρο shuffle και sort βήμα, στην περίπτωση των datasets που δεν έχουν ήδη ταξινομηθεί κατά το join key, και δεν είναι αποδοτική για skewed datasets με άνιση κατανομή δεδομένων, όπου το ένα κλειδί έχει πολλά ταιριάσματα.

- **Shuffle Hash Join**: Η συγκεκριμένη τεχνική κατασκευάζει ένα hash table για το μικρότερο dataset, αφού πρώτα κάνει shuffle, και μετά το ταιριάζει με το μεγαλύτερο dataset. Απαιτεί και τα δύο datasets να είναι partioned στο join key. Συγκεκριμένα:

  1. Και τα δύο datasets υπόκεινται σε shuffle, προκειμένου να βεβαιωθούμε πως είναι partitioned στο join key.
  2. Ένα hash table σχηματίζεται για το μικρότερο dataset σε κάθε partition.
  3. Το μεγαλύτερο dataset διασχίζεται, και τα κλειδιά ταιριάζονται πάνω στο hash table.

  Η συγκεκριμένη στρατηγική είναι πολύ αποδοτική από άποψη μνήμης για μικρά datasets, τα οποία δεν χρειάζεται να γίνουν broadcast. Ωστόσο, απαιτεί shuffling, το οποίο αυξάνει τα I/O στο network, και ο hash table πρέπει να χωράει μέσα στην μνήμη, και άρα δεν είναι ιδιαίτερα αποδοτικό για πολύ μεγάλα datasets. Χρειάζεται να είναι λοιπόν δύο datasets, με χαρακτηριστική διαφορά σε μέγεθος, αλλά χωρίς κανένα από τα δύο να μην είναι αποδοτικό να γίνει broadcast.

- **Replicated Nested Loop Join**: Η συγκεκριμένη στρατηγική αφορά την πιο brute-force τεχνική από τις τέσσερις, όπου κάθε γραμμή από το ένα dataset συγκρίνεται με κάθε γραμμή από το άλλο.

  1. Το ένα dataset αντιγράφεται σε όλους τους κόμβους του cluster.
  2. Για κάθε partition του μεγαλύτερου dataset, κάθε γραμμή του συγκρίνεται με όλες τις γραμμές στο αντιγεγραμμένο dataset.
  3. Οι γραμμές που έκαναν match συμπεριλαμβάνονται στο τελικό αποτέλεσμα.

  Η συγκεκρμένη τεχνική μπορεί να χρησιμοποιηθεί ακόμα και με non-deterministic conditions για join, είναι ωστόσο εξαιρετικά αργό για μεγάλα datasets, με χρονική πολυπλοκότητα $O(n \cdot m)$.

Στον σχολιασμό τώρα των επιδόσεων για καθένα από τα joins:

1. **comm_pop with la_income_clean**
   Ο **Catalyst Optimizer** επέλεξε για το πρώτο join την στρατηγική **Broadcast Join**, πιθανώς επειδή εντόπισε πως το la_income_clean είναι αρκετά μικρό ώστε να γίνει broadcast σε όλους τους κόμβους του cluster, και επειδή η συνθήκη δεν είναι ιδιαίτερα περίπλοκη.

   - Broadcast Join took 14.20 seconds
   - Sort-Merge Join took 11.65 seconds
   - Shuffle Hash Join took 10.86 seconds
   - Replicated Nested Loop Join took 12.24 seconds

   Όπως φαίνεται από τις παραπάνω μετρήσεις, τον καλύτερο χρόνο εδώ παρουσιάζει το **Shuffle Hash Join**, με πολύ μικρή διαφορά από τα **Sort-Merge** και με το **Replicated Nested Loop Join** να ακολουθεί. Όπως αναφέραμε και προηγούμενως, το πρώτο από τα datasets μας είναι ταξινομημένο κατά το κλειδί join του και διαθέτει χαμηλό skewness, γεγονός που ευνοεί την λειτουργία του **Shuffle Hash Join**, του οποίου οι hash buckets μειώνουν την ανάγκη για sorting του δεύτερου dataset. Για το **Sort-Merge Join**, η ταξινόμηση του κλειδιού του join στο ένα τουλάχιστον από τα δύο datasets καθιστά φθηνότερο το merge. Εμφανώς, το μικρότερο από τα δύο datasets (LA_income) παραμένει λίγο μεγαλύτερο από το ιδανικό για να το ευνοεί το broadcast. Τέλος, η κανονική, αν και όχι ανταγωνιστική, απόδοση του **Replicated Nested Loop Join**, μας υποδεικνύει πως δεν πρόκειται για εξαιρετικά μεγάλα datasets.

2. **crime_data_geom with flattened_blocks_df**
   Ο Catalyst Optimizer δεν υποστηρίζει hash-based ή sort-based joins για spatial συνθήκες όπως το ST_Within. Το crime_data_geom και το flattened_blocks_df δεν είναι μικρά, γεγονός που καθιστά απαγορευτικό το Broadcast Join.

   - Broadcast Join - crime_data_location took 30.19 seconds
   - Sort-Merge Join - crime_data_location took 25.05 seconds
   - Shuffle Hash Join - crime_data_location took 16.59 seconds
   - Replicated Nested Loop Join - crime_data_location took 15.52 seconds

   Στο συγκεκριμένο join αντιμετωπίσαμε την ιδιαιτερότητα πως το hint μας φάνηκε να επιτυγχάνει μόνο στην περίπτωση του **Broadcast Join**. Για όλα τα υπόλοιπα, επαναχρησιμοποιήθηκε η τεχνική του **RangeJoin**. Ο λόγος για αυτό είναι πως το **ST_Within** είναι μια spatial συνάρτηση και δεν είναι απλή ισοδυναμία. Το Spark δεν υποστηρίζει hash-based ή sort-based joins σε non-equijoin συνθήκες από προεπιλογή, και ακόμα και αν παρέχονται hints.

3. **income_result with crime_result**
   Ο Catalyst Optimizer λειτούργησε βάση του ότι το **Sort-Merge Join** είναι η καλύτερη επιλογή για equijoin συνθήκες με datasets που έχουν παρόμοιο μέγεθος.

   - Broadcast Join - result took 71.60 seconds
   - Sort-Merge Join - result took 39.25 seconds
   - Shuffle Hash Join - result took 35.52 seconds
   - Replicated Nested Loop Join - result took 26.49 seconds

Τέλος, όπως φαίνεται παραπάνω τον καλύτερο χρόνο παρουσιάσε με διαφορά το **Replicated Nested Loop Join**, το καρτεσιανό δηλαδή γινόμενο, το οποίο μας υποδεικνύει πως πρόκειται για πλέον αρκετά μικρά datasets με απλή συνθήκη join. Το **Shuffle Hash Join** εξακολουθεί να είναι ανταγωνιστικό, υποδεικνύοντας ότι το μέγεθος των δεδομένων ήταν ακόμη διαχειρίσιμο για hash partitioning. Το **Sort-Merge Join** είναι πιο αργό, πιθανώς λόγω της ταξινόμησης που απαιτείται. Τέλος, το **Broadcast Join** είναι πολύ πιο αργό, γεγονός που μας δείχνει πως ενώ τα datasets είναι σχετικά μικρά, δεν έχουν μεγάλες διαφορές μεταξύ τους.

3.4 Query 4: Ανάλυση Φυλετικού Προφίλ Θυμάτων Εγκλημάτων

Υλοποίηση

Η υλοποίηση για το **Query 4** περιλαμβάνει τα εξής βήματα:

1. **Φόρτωση Δεδομένων:** Τα δεδομένα διαβάζονται από αρχεία που είναι αποθηκευμένα στο S3:
   - `Crime_Data_from_2010_to_2019_20241101.csv`: Πληροφορίες εγκλημάτων (συντεταγμένες, ημερομηνίες, χαρακτηριστικά θυμάτων).
   - `LA_income_2015.csv`: Πληροφορίες εισοδήματος ανά περιοχή (ZIP code).
   - `RE_codes.csv`: Κωδικοί φυλετικής καταγωγής (`Vict Descent`) με την περιγραφή τους.
   - `2010_Census_Blocks.geojson`: Γεωγραφικά δεδομένα για τα blocks.
2. **Προσθήκη Γεωμετρικής Πληροφορίας:** Στον πίνακα `crime_data`, προστίθεται γεωμετρική πληροφορία (`geometry`) μέσω της συνάρτησης `ST_Point`. Η γεωμετρική σύζευξη (`ST_Within`) χρησιμοποιείται για να αντιστοιχηθούν τα εγκλήματα σε συγκεκριμένες περιοχές.
3. **Φιλτράρισμα Εγκλημάτων για το Έτος 2015:** Επιλέγονται μόνο τα εγκλήματα που διαπράχθηκαν το 2015, μέσω φιλτραρίσματος στη στήλη `DATE OCC`.
4. **Εύρεση Περιοχών με Υψηλό και Χαμηλό Εισόδημα:** Από το προηγούμενο Query γνωρίζουμε τις περιοχές με το υψηλότερο και χαμηλότερο εισόδημα. Επομένως, φιλτράρουμε τα δεδομένα του LA_income_2015, προκειμένου να διατηρήσουμε μόνο τις καταχωρήσεις που περιέχουν τα ονόματα αυτών των περιοχών στο πεδίο "Community".
   - Οι 3 περιοχές με το υψηλότερο εισόδημα εισάγονται στο dataset (`top_income`).
   - Οι 3 περιοχές με το χαμηλότερο εισόδημα εισάγονται στο dataset (`bottom_income`).
5. **Αντιστοίχιση Δεδομένων και Φιλτράρισμα με ZIP Codes:** Τα δεδομένα εγκλημάτων (`crime_data_2015`) συσχετίζονται με τις περιοχές εισοδήματος (`la_income`) μέσω της στήλης `ZCTA10` και `COMM`. Στη συνέχεια, γίνεται επιπλέον συσχέτιση με τους κωδικούς φυλετικής καταγωγής (`RE_codes`).
6. **Ομαδοποίηση και Καταμέτρηση Εγκλημάτων:**
   - Οι περιοχές με το υψηλότερο εισόδημα (`richest_crimes`) και οι περιοχές με το χαμηλότερο εισόδημα (`poorest_crimes`) φιλτράρονται βάσει των `Zip Codes`.
   - Ομαδοποιούνται τα εγκλήματα ανά φυλετική καταγωγή (`Vict Descent Full`), και μετριέται ο συνολικός αριθμός θυμάτων για κάθε ομάδα.
7. **Εμφάνιση Αποτελεσμάτων:** Τα αποτελέσματα εκτυπώνονται σε δύο πίνακες:
   - Πίνακας για τις περιοχές με το υψηλότερο εισόδημα, ταξινομημένος κατά αριθμό θυμάτων (φθίνουσα σειρά).
   - Πίνακας για τις περιοχές με το χαμηλότερο εισόδημα, ταξινομημένος επίσης κατά αριθμό θυμάτων.

Table 8: Περιοχές με Υψηλότερο Εισόδημα

Vict Descent Full	Count
White	668
Other	99
Hispanic/Latin/Mexican	77
Black	50
Unknown	48
Other Asian	23
Chinese	1
American Indian/	1

Table 9: Περιοχές με Χαμηλότερο Εισόδημα

Vict Descent Full	Count
Hispanic/Latin/Mexican	3157
Black	1292
White	195
Unknown	167
Other	113
Other Asian	24
Filipino	2
Japanese	1
Guamanian	1
Pacific Islander	1

Συγκρίσεις και Αποτελέσματα

Στην συνέχεια, καλούμαστε να εκτελέσουμε την υλοποίησή μας του συγκεκριμένου Query, εφαρμόζοντας κλιμάκωση στο σύνολο των υπολογιστικών πόρων που θα χρησιμοποιήσουμε. Καλούμαστε, δηλαδή, να εκετελέσουμε την υλοποίησή μας σε **2 executors**, με τρία διαφορετικά configurations.
Προτού παραθέσουμε και σχολιάσουμε τα αποτελέσματά μας, είναι σημαντικό να κατανοήσουμε ορισμένα δεδομένα για τα configurations αυτά στο Apache Spark.
Σε καθαρά δομικές γνώσεις, το AWS SageMaker αποτελείται από clusters, τα οποία με την σειρά τους αποτελούνται από κόμβους, που συνεργάζονται ως ένα ενιαίο σύστημα για να εκτελέσουν tasks παράλληλα. Ένα cluster αποτελείται από:

1. Ένα **Master Node**, με ρόλο την οργάνωση, την ανάθεση και την παρακολούθηση της εκτέλεσης των tasks από τα worker nodes. Στο Spark, τον ρόλο αυτό επιτελεί το **driver process**.

2. Τα **Worker Nodes**, με ρόλο την εκτέλεση των tasks που τους ανατίθενται από το Master Node. Επιτελούν τον διαμοιρασμό των δεδομένων, του υπολογισμούς και την αποθήκευση ενδιάμεσων δεδομένων, καθώς επίσης και την εκτέλεση των executor processes, που εκτελούν tasks παράλληλα.

Γνωρίζουμε πως οι **executors** αποτελούν τις διεργασίες, υπεύθυνες για την εκτέλεση ανεξάρτητων tasks μιας δουλειάς Spark, μέσα σε κάθε κόμβο του δικτύου του. Φυσικά, μπορούν να υπάρχουν παραπάνω από ένας executor σε κάθε κόμβο, αλλά υπάρχει ένα threshold σε κάθε κόμβο, ο οποίος δηλώνει μέχρι πόσους executors μπορεί να υποστηρίξει ώστε να είναι το δίκτυο αποδοτικό. Ως ξεχωριστές διεργασίες, ακόμα και οι executors που τρέχουν στον ίδιο κόμβο δεν μοιράζονται την ίδια μνήμη (καθώς, εξ' ορισμού διαθέτουν διαφορετικό memory address space) και χρειάζονται την Inter-Process (Inter-Executor) επικοινωνία.
Ακόμη ένα σημαντικό αντικείμενο προς κατανόηση, είναι ο αριθμός των **cores** (πυρήνων) που ανατίθενται στον κάθε executor (εκτελεστή). Αυξάνοντας τον αριθμό των **Cores per Executor**, αντί να αυξήσουμε τον αριθμό των executors απευθείας, μπορεί να μειωθεί το κόστος της επικοινωνίας μεταξύ διαφορετικών Executors (Inter-Executor Communication) μέσω της κοινής μνήμης και να αυξηθεί η ταχύτητα ολοκλήρωσης του κάθε task.
Με βάση αυτά, μια πρώτη σκέψη για την επίτευξη της βέλτιστης απόδοσης, θα ήταν η δημιουργία **"Fat executors"**. Πρόκειται ουσιαστικά για την δημιουργία εκτελεστών με μεγάλο αριθμό πυρήνων (σχεδόν όλων των πυρήνων που διαθέτει ένας κόμβος του δικτύου), με σκοπό να διατηρηθεί μόλις ένας executor ανά working node. Ενώ ωστόσο η τεχνική αυτή θα ελαχιστοποιούσε την επικοινωνία ανάμεσα στους εκτελεστές (Inter-Executor Communication), θα προκαλούσε ορισμένα άλλα προβλήματα, όπως:

1. Προβλήματα στην μνήμη του ίδιου του executor, η οποία είναι κοινή ανάμεσα στους πυρήνες και περιορισμένη. Αποτέλεσμα της, θα ήταν οι καθυστερήσεις από τον garbage collector, ο οποίος θα πρέπει να διαχειριστεί περισσότερα δεδομένα στο ίδιο heap και άρα να απελευθερώνει συνεχώς αχρησιμοποίητη μνήμη που καταλαμβάνουν τα πολλά διαφορετικά tasks που θα εκτελούνται ταυτόχρονα στον κάθε executor.

2. Ανισορροπία στην χρήση των πόρων του cluster, με αποτέλεσμα υποεκμετάλλευση ορισμένων κόμβων.

Έχοντας πλέον μια καθαρή εικόνα της δομής ενός **Spark Cluster**, μπορούμε να προχωρήσουμε στην ανάλυση των αποτελεσμάτων από τις εκτελέσεις των διαφορετικών configurations. Παρακάτω παρατίθενται τα αποτελέσματα των 10 δοκιμών μας, για τα τρία διαφορετικά configurations, μετρημένα σε δευτερόλεπτα (seconds).

Table 10: Comparison of Execution Times for Query 4 Configurations

Configuration	Try 1	Try 2	Try 3	Try 4	Try 5
1 core/2GB memory	71.43	97.33	86.80	64.09	97.49
2 cores/4GB memory	66.84	81.29	66.52	63.32	70.49
4 cores/8GB memory	67.96	70.93	64.46	63.61	65.86

Configuration	Try 6	Try 7	Try 8	Try 9	Try 10
1 core/2GB memory	88.61	98.34	89.51	90.19	84.40
2 cores/4GB memory	74.73	70.70	68.68	71.74	68.72
4 cores/8GB memory	66.03	66.99	66.00	67.08	64.79

Σχολιασμός

Παρακάτω σχολιάζουμε τις επιδόσεις των τριών διαφορετικών configurations:

1. Αρχικά, παρατηρείται πως το configuration **1 core/2GB memory** παρουσιάζει τους μεγαλύτερους και μεγαλύτερου εύρους χρόνους ανάμεσα στις εκτελέσεις. Η χαμηλή απόδοση του συγκεκριμένου configuration μπορεί εύκολα να αποδοθεί στον χαμηλό βαθμό παραλληλοποίησης των tasks, καθώς επίσης και στην μηδενική εκμετάλλευση της κοινής μνήμης ανάμεσα στα cores ενός executor. Ακόμη, κάθε executor διαθέτει την λιγότερη μνήμη συγκριτικά με τα υπόλοιπα configurations μας, γεγονός που απαιτεί την συνεχή εκκαθάριση της προκειμένου πολλά διαφορετικά tasks να εκτελεστούν.

2. Το configuration **2 cores/4GB memory** παρουσιάζει μια χαρακτηριστική βελτίωση και πιο σταθερή συμπεριφορά σε σχέση με το πρώτο, καθώς επίσης και σε περιπτώσεις κοντινούς χρόνους με την καλύτερη εκδοχή, το τρίτο configuration. Μπορούμε να αιτιολογήσουμε αυτό το αποτέλεσμα με το γεγονός της αύξησης της παραλληλοποίησης, καθώς πλέον κάθε executor διαθέτει 2 cores, καθώς επίσης και με την αύξηση της κοινής μνήμης των executor, η οποία έχει διπλασιαστεί.

3. Όπως φαίνεται παραπάνω, το τρίτο configuration (**4 cores/8GB memory**), αποδείχθηκε σε όλες τις μετρήσεις το πιο γρήγορο και σταθερό σε χρόνους εκτέλεσης. Η έκβαση αυτή είναι πολύ λογική, καθώς παρουσιάζει τον μέγιστο βαθμό παραλληλοποίησης χωρίς να ξεφεύγει από τα προτινόμενα όρια πυρήνων ανά executor, καθώς επίσης παρέχει αρκετή μνήμη ανά executor προκειμένου να εκτελούνται χωρίς πρόβλημα τα tasks.

3.5 Query 5

Περιγραφή Υλοποίησης για το Query 5

Το Query 5 αφορά την εύρεση, ανά αστυνομικό τμήμα (`DIVISION`), του αριθμού των εγκλημάτων που έλαβαν χώρα πλησιέστερα σε αυτό, καθώς και της μέσης απόστασης (`average distance`) των εγκλημάτων από το αντίστοιχο τμήμα. Τα αποτελέσματα ταξινομούνται κατά αριθμό εγκλημάτων (`crime count`) σε φθίνουσα σειρά. Η υλοποίηση πραγματοποιείται ως εξής:

Αρχικοποίηση και Φόρτωση Δεδομένων:

- Δημιουργείται ένα Spark Session μέσω της συνάρτησης `init_spark`, όπου καθορίζονται οι πόροι (cores, memory, executors) για κάθε διαμόρφωση.
- Τα δεδομένα εγκλημάτων (`crime_data`) φορτώνονται από δύο αρχεία CSV, `Crime_Data_from_2010_to_2019` και `Crime_Data_from_2020_to_Present`, τα οποία ενώνονται μέσω `union`.
- Τα δεδομένα των αστυνομικών τμημάτων (`police_stations`) φορτώνονται από το αρχείο `LA_Police_Stations.csv`.

Δημιουργία Γεωμετρικών Δεδομένων:

- Δημιουργούνται γεωμετρικά σημεία (`ST_Point`) για τις συντεταγμένες των εγκλημάτων (`crime_point`) και των αστυνομικών τμημάτων (`station_point`).

Υπολογισμός Αποστάσεων:

- Πραγματοποιείται **cross join** μεταξύ των δεδομένων εγκλημάτων και αστυνομικών τμημάτων.
- Υπολογίζεται η απόσταση (`distance`) μεταξύ κάθε εγκλήματος και κάθε αστυνομικού τμήματος χρησιμοποιώντας τη συνάρτηση `ST_DistanceSphere`.

Εύρεση Πλησιέστερου Αστυνομικού Τμήματος:

- Γίνεται ομαδοποίηση των εγκλημάτων (`groupBy`) ανά `DR_NO` και υπολογίζεται η ελάχιστη απόσταση (`minimum_distance`) για κάθε έγκλημα.
- Πραγματοποιείται `join` μεταξύ του αρχικού `crime_distances` και του παραπάνω αποτελέσματος για να βρεθεί το πλησιέστερο τμήμα.
- Επιλέγονται μόνο τα απαραίτητα πεδία (`DR_NO`, `DIVISION`, `distance`).

Ομαδοποίηση και Υπολογισμός Μετρήσεων:

- Τα δεδομένα ομαδοποιούνται (`groupBy`) ανά `DIVISION`.
- Υπολογίζονται:
  - Ο συνολικός αριθμός εγκλημάτων (`crime count`) μέσω `count`.
  - Η μέση απόσταση (`average distance`) μέσω `avg`.
- Τα αποτελέσματα ταξινομούνται κατά `crime count` σε φθίνουσα σειρά (`orderBy`).

Εκτέλεση για Διαφορετικά Configurations:

- Το Query εκτελείται για τρεις διαφορετικές διαμορφώσεις (διαφορετικά cores, memory και executors).
- Χρονίζεται ο συνολικός χρόνος εκτέλεσης (`execution time`) για κάθε διαμόρφωση.

Συγκρίσεις και Αποτελέσματα

Table 11: Comparison of Execution Times for Different Configurations

Configuration	Try 1	Try 2	Try 3	Try 4	Try 5
2 executors × 4 cores/8GB	16.55	27.39	7.25	6.63	6.29
4 executors × 2 cores/4GB	8.01	18.64	6.98	6.23	6.15
8 executors × 1 core/2GB	7.18	8.57	6.72	6.28	6.20

Configuration	Try 6	Try 7	Try 8	Try 9	Try 10
2 executors × 4 cores/8GB	6.36	6.01	6.35	10.62	6.22
4 executors × 2 cores/4GB	6.00	5.98	5.83	7.37	6.11
8 executors × 1 core/2GB	6.03	5.94	6.18	6.25	6.05

Σχολιασμός

Παρακάτω σχολιάζουμε τα αποτελέσματα που πήραμε για τα παραπάνω configurations, στηριζόμενοι στην θεωρία που αναλύσαμε στο προηγούμενο ερώτημα.

1. **2 executors × 4 cores/8GB**: Το συγκεκριμένο configuration παρουσιάζει τους κατά κανόνα μεγαλύτερους χρόνους σε σύγκριση με τα υπόλοιπα. Παρόλης της ύπαρξης περισσότερης μνήμης ανά executor και περισσότερων cores, η μεγάλη διακύμανση στους χρόνους δείχνει ότι ο μικρότερος αριθμός executors δεν ευνοεί την επίδοση της εκτέλεσής μας. Το γεγονός αυτό μπορούμε να δικαιολογήσουμε αρχικά, γνωρίζοντας πως σε περίπτωση αποτυχίας ενός από τα tasks που τρέχουν, τότε και τα 4 tasks του συγκεκριμένου executor θα πρέπει να επανεπεξεργαστούν. Προφανώς, μειώνεται η ανάγκη για Inter-Executor Communication, καθώς κάθε executor διαθέτει από 8GB κοινής μνήμης για τα 4 cores του. Ωστόσο, αυτή η δυνατότητα δεν φαίνεται να βελτιώνει ιδιαίτερα τις επιδόσεις.

2. **4 executors × 2 cores/4GB**: Αυτή η εκδοχή παρουσιάζει σημαντική συνολική βελτίωση από το προηγούμενο configuration. Με την αύξηση των executors, και την ταυτόχρονη μείωση των cores τους και της κοινής μνήμης του καθενός, το σύστημά μας φαίνεται παρόλαυτά να λειτουργεί καλύτερα, γεγονός που μας οδηγεί στο να πιστέψουμε πως δεν υπήρξε μεγάλη ανάγκη σε Inter-Executor Communication για την επεξεργασία των tasks. Πράγματι, η τακτική ομαδοποίηση των DataSets κατά την κάθε φορά σημαντική πληροφορία, ενδέχεται να επιφέρει σημαντικά σε αυτή την απόδοση.

3. **8 executors × 1 core/2GB**: Το κατά τον μεγαλύτερο βαθμό βέλτιστης απόδοσης configuration μας. Από ότι φαίνεται, η μείωση της μνήμης του κάθε executor και ο περιορισμός του ενός task ανά executor δεν επηρεάζει αρνητικά την επίδοση της εκτέλεσής μας, οδηγώντας μας ξανά στην σκέψη πως η συνεχής ομαδοποίηση και οργάνωση των δεδομένων δεν απαιρεί πολλές Inter-Executor επικοινωνίες.