Προηγμένα Θέματα Βάσεων Δεδομένων: Θεωρητική Ανάλυση και Αναφορά Εργασίας

Ομάδα 15
Κατσαϊδώνης Νικόλαος 03121868
Τζαμουράνης Γεώργιος 03121141

Δεκέμβριος 2025

1 Σκοπός

Η παρούσα εργασία, διεξάγεται στα πλαίσια του μαθήματος **"Προχωρημένα Θέματα Βάσεων Δεδομένων"** της σχολής **Ηλεκτρολόγων Μηχανικώ και Μηχανικών Υπολογιστών ΕΜΠ** και σκοπός της είναι η εξοικείωση με την επεξεργασία και ανάλυση Big Data καθώς και η ενασχόληση με κατανεμημένα συστήματα. Συγκεκριμένα θα χρησιμοποιήσουμε το σύστημα Apache Spark για εκτέλεση διάφορων queries πάνω σε δεδομένα σχετικά με το dataset "Los Angeles Crime Data" (κάνοντας χρήση και άλλων δευτερευόντων datasets) και θα πειραματιστούμε τόσο με τις στρατηγικές εκτέλεσης join καθώς και με τους διαφορετικούς πόρους (executors, RAM, cores) τους οποίους το Spark μας παρέχει. Η εργασία πραγματοποιείται εντός του περιβάλλοντος Amazon Sage Maker AWS.

2 Κώδικας Εργασίας

Παραθέτουμε παρακάτω link του github repository με τον κώδικα της εργασίας. Συγκεκριμένα, περιέχει ένα notebook όπου σε κάθε κελί περιέχεται καθένα από τα ζητούμενα της εργασίας.

Github Repository link: `https://github.com/NikosK10/Advanced_DataBase_Systems`

3 Ζητούμενα εργασίας

Παρατήρηση: Τονίζουμε ότι κατά τις μετρήσεις τις οποίες βλέπετε παρακάτω, προσπαθήσαμε να είμαστε όσο πιο αντικειμενικοί ήταν εφικτό και για αυτόν τον λόγο, πριν τον τελικό χρόνο που καταγράψαμε κάναμε **Kernel Restart για να αποφύγουμε το caching** που ενδεχομένως να υπήρχε. Επιπλέον, σε όλα τα προς σύγκριση ζητούμενα **χρονομετρήθηκε όμοια και αξιοκρατικά το data loading και εξαιρέθηκε ο χρόνος δημιουργίας του spark session** ώστε να έχουμε δίκαιη σύγκριση.

3.1 Query 1: Ηλικιακές Ομάδες Θυμάτων "Βαριάς Σωματικής Βλάβης"

Να ταξινομημηθούν, σε φθίνουσα σειρά, οι ηλικιακές ομάδες των θυμάτων σε περιστατικά που περιλαμβάνουν οποιαδήποτε μορφή "βαριάς σωματικής βλάβης". Θεωρείστε τις εξής ηλικιακές ομάδες: Παιδιά<18, Ενήλικοι 25--64, Νεαροί ενήλικοι 18--24, Ηλικιωμένοι>64.

1. **DataFrame με UDF**: Εκτέλεση μέσω της Spark Dataframe API με χρήση UDF για τον υπολογισμό της ηλικιακής ομάδας.
   Χρόνος εκτέλεσης: 11.92 δευτερόλεπτα

2. **DataFrame χωρίς UDF**: Υπολογισμός της ηλικιακής ομάδας με native Spark expressions (when/otherwise) χωρίς χρήση UDF.
   Χρόνος εκτέλεσης: 10.56 δευτερόλεπτα

3. **RDD**: Κλασική υλοποίηση με RDD, όπου τα δεδομένα διαβάστηκαν και επεξεργάστηκαν με map, filter, reduceByKey και sortBy.
   Χρόνος εκτέλεσης: 16.45 δευτερόλεπτα

Σχολιασμός Αποτελεσμάτων

Η **DataFrame χωρίς UDF** υλοποίηση ήταν η ταχύτερη (10.56 s), ακολουθούμενη από τη **DataFrame με UDF** (11.92 s) και τελευταίο το **RDD** (16.45 s).

- Το γεγονός ότι η χρήση UDF στο DataFrame είναι ελαφρώς πιο αργή από τη native Spark υλοποίηση είναι αναμενόμενο. Οι UDF εκτελούνται ως μαύρο κουτί Python κώδικα για κάθε row, οπότε **δεν επωφελούνται από τις βελτιστοποιήσεις του Catalyst optimizer της Spark.** Αντίθετα, οι native Spark expressions επιτρέπουν στον optimizer να κάνει pushdown, vectorization και άλλες βελτιστοποιήσεις.

- Η RDD υλοποίηση είναι η πιο αργή (16.45 s), κάτι που επίσης είναι αναμενόμενο. Οι RDDs είναι **χαμηλού επιπέδου**, δεν επωφελούνται από τις βελτιστοποιήσεις της Spark SQL engine, και η επεξεργασία μέσω map και filter σε Python κώδικα έχει μεγαλύτερο κόστος από τις DataFrame υλοποιήσεις.

3.2 Query 2: Top 3 Φυλετικά Γκρουπ Θυμάτων Ανά Έτος

Ανά έτος, να βρεθούν τα 3 φυλετικά γκρουπ με τα περισσότερα θύματα καταγεγραμμένων εγκλημάτων (Vict Descent) στο Los Angeles. Τα αποτελέσματα να εμφανιστούν με φθίνουσα σειρά αριθμού θυμάτων ανά φυλετικό γκρουπ – να υπολογιστεί και να εμφανιστεί επίσης το ποσοστό επί του συνολικού αριθμού θυμάτων ανα περίπτωση.

1. **DataFrame API**: Υλοποίηση μέσω Spark DataFrame transformations (groupBy, join, withColumn, Window, row_number).
   Χρόνος εκτέλεσης: 20.89 δευτερόλεπτα

2. **SQL API**: Υλοποίηση μέσω Spark SQL queries με χρήση temp views και SQL window functions.
   Χρόνος εκτέλεσης: 30.21 δευτερόλεπτα

Σχολιασμός Αποτελεσμάτων

- Οι **transformations στο DataFrame API** εκτελούνται άμεσα από τον **Catalyst optimizer**, που κάνει **predicate pushdown**, **physical planning** και **vectorization**.
- Η **SQL API** υλοποίηση απαιτεί την δημιουργία **temp views**, **parsing του SQL query και translation σε DataFrame σχέδιο πριν εκτελεστεί**, προσθέτοντας **επιπλέον κόστος**.
- Οι χρόνοι εκτέλεσης δείχνουν ότι η χρήση **SQL queries σε Spark** δεν είναι απαραίτητα πιο αποδοτική από τα **native DataFrame transformations**, ιδιαίτερα σε workflows που περιλαμβάνουν σύνθετες επεξεργασίες όπως **window functions**, **joins** και υπολογισμό ποσοστών.

3.3 Query 3: Μέθοδοι διάπραξης εγκλημάτων και MO Codes

Να ταξινομηθούν και να εμφανιστούν με φθίνουσα σειρά συχνότητας εμφάνισης οι μέθοδοι διάπραξης εγκλημάτων και οι αντίστοιχοι κωδικοί τους (Mocodes). Χρησιμοποιήστε το σύνολο MO Codes για να αντιστοιχίσετε τους κωδικούς με τις περιγραφές τους.

Πίνακας Αποτελεσμάτων

Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης και τα Physical Plans που επιλέχθηκαν από τον Catalyst Optimizer για κάθε περίπτωση.

Υλοποίηση / Στρατηγική (Hint)	Χρόνος (sec)
RDD API	17.53
DataFrame (Default: Broadcast Hash Join)	4.42
DataFrame (Broadcast Hint)	4.20
DataFrame (Sort Merge Join)	4.44
DataFrame (Shuffle Hash Join)	7.59
DataFrame (Replicate NL Join Αγνοήθηκε)	3.89

Πίνακας 1: Σύγκριση στρατηγικών join

Σχολιασμός Αποτελεσμάτων

Το DataFrame API (4.2s) είναι περίπου **4 φορές πιο γρήγορο** από το RDD API (17.53s). Αυτό το αποτέλεσμα είναι απολύτως αναμενόμενο και εξηγείται από τη θεμελιώδη αρχιτεκτονική του Spark. Το RDD API είναι "τυφλό". Ο προγραμματιστής ορίζει ακριβώς πώς θα γίνει η επεξεργασία (map, flatMap, join). Αντιθέτως, στο **DataFrame API**, ο **Catalyst Optimizer** αναλύει το query και δημιουργεί το **βέλτιστο πλάνο εκτέλεσης** (π.χ. επιλέγει Broadcast Join αντί για Shuffle Join, κάνει **filter pushdown κλπ.**).

Ανάλυση Στρατηγικών Join (DataFrame Hints)

Εδώ αναλύουμε πώς αντέδρασε ο Spark στα διάφορα hints που δώσαμε για το Join μεταξύ του μεγάλου πίνακα **Crime Data (Left side)** και του πολύ μικρού πίνακα **MO_Codes (Right side)**.

α) Default Join and Broadcast Hash Join

Χρόνοι: 4.20s & 4.42s.

Λειτουργία: Το Spark αντιλαμβάνεται ότι ο πίνακας MO_Codes είναι πολύ μικρός (χωράει στη μνήμη). Αντί να μετακινήσει τα τεράστια δεδομένα των εγκλημάτων, στέλνει (**broadcast**) ένα αντίγραφο του μικρού πίνακα σε κάθε Executor. Έτσι, πχ ο Executor 1 διαβάζει ένα partition του Crime πίνακα και έχει ολόκληρο τον Mocodes table στη RAM του. Έτσι, μπορεί να κάνει τη δουλειά του **χωρίς να μιλήσει καθόλου με τον Executor 2**. Αποφύγαμε λοιπόν την δαπανηρή και αργή μετακίνηση (**Shuffle**) του τεράστιου πίνακα των εγκλημάτων μεταξύ των executors, στέλνοντας αντ' αυτού μόνο ένα αντίγραφο του μικροσκοπικού πίνακα κωδικών στη μνήμη κάθε Executor, ώστε η ένωση να γίνει τοπικά και χωρίς καθυστερήσεις. Έπειτα, γίνεται **hashing** πάνω στο κλειδί MO_codes ώστε σε χρόνο O(1) να αντιστοιχίζεται κάθε tuple του μεγάλου crime πίνακα στον σωστό MO_code.

Γιατί επιλέχθηκε: Είναι η **βέλτιστη στρατηγική** για joins τύπου "Big Table JOIN Small Table", καθώς αποφεύγει το δαπανηρό **Network Shuffle** του μεγάλου πίνακα.

β) Sort Merge Join

Χρόνος: 4.44s.

Λειτουργία: Απαιτεί να ταξινομηθούν (Sort) και οι δύο πίνακες με βάση το κλειδί (MO_code) και μετά να συγχωνευτούν (Merge). Αυτό προκαλεί **Shuffle** (μετακίνηση δεδομένων μεταξύ executors) και στους δύο πίνακες.

- **Shuffle Phase:** Αρχικά, τα δεδομένα των δύο πινάκων (Crimes και MO_Codes) βρίσκονται διασκορπισμένα στους Executors με βάση τη φυσική τους αποθήκευση, χωρίς καμία συσχέτιση με το κλειδί συνένωσης MO_code. Κατά τη φάση του Shuffle, το Spark διαβάζει αυτά τα αρχικά partitions και τα αναδιανέμει μέσω

του δικτύου. Χρησιμοποιώντας έναν αλγόριθμο κατακερματισμού (hashing) πάνω στο MO_code, διασφαλίζει ότι **μετά την ανακατανομή**, όλες οι εγγραφές με το ίδιο MO_code (και από το αριστερό και από το δεξί DataFrame) θα καταλήξουν στο ίδιο partition στον ίδιο Executor. Αυτό είναι προαπαιτούμενο για να μπορέσει να εκτελεστεί η επόμενη φάση (Sort) και τελικά το Join. Αυτή η φάση προκαλεί υψηλό **Network I/O** και κόστος Serialization/Deserialization, καθώς μεγάλος όγκος δεδομένων μετακινείται μεταξύ των κόμβων.

- **Sort phase:** Τα δεδομένα κάθε partition ταξινομούνται κατά MO_code.

- **Merge phase:** Έχοντας πλέον ταξινομημένους τους δύο πίνακες, ο αλγόριθμος κάνει merge χωρίς να χρειάζεται να διατρέξει όλη τη λίστα για να βρει τα αντίστοιχα MO_codes από τους δύο πίνακες αφού πλέον έχει προηγηθεί ταξινόμηση.

Είναι μια αρκετά βαριά στρατηγική για "Large JOIN Large" πίνακες. Εδώ, όμως, είναι περιττή σπατάλη πόρων, καθώς αναγκάζουμε το Spark να κάνει shuffle τα εκατομμύρια εγγραφές των εγκλημάτων, ενώ θα μπορούσαμε απλά να κάνουμε broadcast τον μικρό πίνακα. Παρόλα αυτά, η διαφορά χρόνου δεν ήταν τεράστια, πιθανώς επειδή το sorting του μικρού πίνακα ήταν ακαριαίο.

γ) Shuffled Hash Join
Χρόνος: 7.59s.

Λειτουργία: Το Shuffled Hash Join αποτελεί μια εναλλακτική στρατηγική όταν τα δεδομένα είναι πολύ μεγάλα για Broadcast, αλλά θέλουμε να αποφύγουμε το υψηλό κόστος ταξινόμησης του SortMergeJoin.

- **Shuffle Phase:** Όπως και στο SortMergeJoin, και οι δύο πίνακες υφίστανται repartitioning βάσει του κλειδιού (MOcode). Διασφαλίζεται ότι τα δεδομένα με το ίδιο κλειδί συναντώνται στον ίδιο Executor. Προφανώς στη διαδικασία αυτή εμπεριέχεται **υψηλό I/O cost**.

- **Hash Build and Probe Phase:** Εδώ είναι που διαφέρει από τον αλγόριθμο Sort Merge Join: Ο Executor επιλέγει το μικρότερο από τα δύο data frames (εδώ το τμήμα των MOCodes) και δημιουργεί ένα **Hash Map** στη μνήμη (Build). Στη συνέχεια, κάνει scan γραμμικά το μεγαλύτερο partition (Crimes). Για κάθε εγγραφή, κάνει look-up στο Hash Map σε χρόνο O(1) και πραγματοποιεί το Join (Probe). .

Συνήθως χρησιμοποιείται όταν η μία πλευρά είναι μικρή αλλά όχι αρκετά μικρή για Broadcast. Επίσης, σε σχέση με την sort, δεν σπαταλά χρόνο και CPU για sorting. Αντίθετα, χτίζει ένα **Hash Index** στη μνήμη για τον μικρότερο πίνακα και κάνει **άμεση αναζήτηση** O(1) για κάθε εγγραφή του μεγάλου πίνακα.

δ) Shuffle Replicate NL

Παραλήφθηκε και εφαρμόστηκε Broadcast Hash Join. Χρόνος: 3.89s.

Λειτουργία: Το Shuffle Replicate NL (Nested Loop) είναι συνήθως η λιγότερο αποδοτική μέθοδος συνένωσης, με πολυπλοκότητα O(MxN). Συνήθως χρησιμοποιείται σε **Cross Joins** (όπου δεν υπάρχει κλειδί συνένωσης ON) ή **Non-Equi Joins** (π.χ. συνθήκες ανισότητας όπου δεν μπορούμε να κάνουμε hashing).

Παρατήρηση: Παρόλο που δόθηκε ρητά το hint "shuffle replicate NL" στον κώδικα, ο Catalyst Optimizer επέλεξε να το αγνοήσει στο τελικό Physical Plan και εφάρμοσε στρατηγική BroadcastHashJoin όπως φαίνεται και στο explain. Αυτό είναι ένα αξιοπερίεργο γεγονός ωστόσο θα μπορούσε να είναι μια μορφή προστασίας του Spark από πολύ δαπανηρές συγκρίσεις που είναι άσκοπες. Διαφορετικά, κάποιο λάθος έγινε κατά τον πειραματισμό το οποίο δεν εντοπίσαμε με επιτυχία.

Ποια είναι η καταλληλότερη στρατηγική;

Για το συγκεκριμένο query (**Join ενός πολύ μεγάλου table με έναν πολύ μικρό**), η πιο καταλληλη στρατηγική είναι το **BroadcastHashJoin**. Έτσι πετυχαίνουμε:

- **Ελαχιστοποίηση Network I/O**: Δεν μετακινούνται τα "βαριά" δεδομένα των εγκλημάτων μέσω του δικτύου (no shuffle).
- **Ταχύτητα**: Το lookup σε Hash Map που βρίσκεται στη μνήμη (RAM) κάθε executor είναι πολύ γρήγορο (**O(1)**).
- **Αποφυγή Sorting**: Δεν απαιτείται ταξινόμηση των δεδομένων όπως στο SortMergeJoin.

Το query επιβεβαιώνει ότι το Spark (μέσω του Default plan) κάνει σωστά τη δουλειά του επιλέγοντας Broadcast αυτόματα, ενώ το RDD API υστερεί σημαντικά σε απόδοση λόγω έλλειψης αυτών των αυτοματισμών.

3.4 Query 4:Πλησιέστερα αστυνομικά τμήματα εγκλημάτων

Να υπολογιστεί, ανά αστυνομικό τμήμα, ο αριθμός εγκλημάτων που έλαβαν χώρα πλησιέστερα σε αυτό από οποιοδήποτε άλλο, καθώς και η μέση απόστασή του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά. Τα αποτελέσματα να εμφανιστούν ταξινομημένα κατά αριθμό περιστατικών, με φθίνουσα σειρά.

Υλοποίηση:

Για το query αυτό κάνουμε χρήση του **cross join** για τη δημιουργία Καρτεσιανού Γινομένου ώστε κάθε έγκλημα να συνδέεται με όλα τα αστυνομικά τμήματα προκειμένου να υπολογιστούν όλες οι αποστάσεις (με χρήση **DistanceSphere από Sendona**)

και να βρεθεί η ελάχιστη από τις αποστάσεις αυτές. Η εύρεση της ελάχιστης απόστασης επιτυγχάνεται με τη χρήση **Window Function** η οποία ταξινομεί το κάθε ζεύγος έγκλημα-αστυνομικό τμήμα βάσει απόστασης και κρατώντας τελικά μόνο αυτό με τη μικρότερη απόσταση (**row number=1**). Έπειτα, για την καταμέτρηση εγκλημάτων ανά division γίνεται ένα **groupBy division**. Τέλος, γίνεται ένα **aggregation** για τον υπολογισμό **average απόστασης** για κάθε ένα αστυνομικό τμήμα.

Cores per Executor	RAM	Συνολικά Cores	Χρόνος (sec)
1	2 GB	1	2.23
2	4 GB	4	2.12
4	8 GB	8	1.97

Πίνακας 2: Κλιμάκωση cores και μνήμης

Execution Plan:

Με βάση τα Physical Plans που παρήχθησαν και στα τρία πειράματα, ο Catalyst Optimizer επέλεξε σταθερά τη στρατηγική **Broadcast Nested Loop Join (BNLJ)** με τύπο Cross. Το Spark αναγνώρισε ότι το DataFrame των αστυνομικών τμημάτων είναι μικρό σε μέγεθος. Αντί να εκτελέσει ένα δαπανηρό Shuffle και των δύο πινάκων (Shuffle-Replicate-NL), έκανε **Broadcast** τον πίνακα των τμημάτων σε όλους τους Executors, δηλαδή **αποθήκευση στην RAM**. Έτσι, κάθε Executor επεξεργάστηκε τα εγκλήματα του partition του, συγκρίνοντάς τα τοπικά με την πλήρη λίστα των τμημάτων κάνοντας χρήση του αλγόριθμου Nested Loop για τον υπολογισμό της απόστασης. Για κάθε ένα έγκλημα δηλαδή, έκανε μία, δαπανηρή θα λέγαμε, σύγκριση με το κάθε station (**O(MxN)**). Ωστόσο, έχοντας κάνει broadcast, κάθε executor γλίτωσε αρκετά I/O μέσω δικτύου.

Configurations expirements:

Στο query αυτό πειραματιστήκαμε αλλάζοντας κάθε φορά το πλήθος των cores per executors καθώς και την μνήμη, διατηρώντας πάντα πλήθος 2 executors. Όπως μπορείτε να δείτε και στον Πίνακα 2 κλιμακώσαμε μνήμη και cores **από 1core/2GB σε 2 cores/4GB και τέλος σε 4 cores/8GB** παρατηρώντας **σταδιακή μείωση χρόνου**. Ας εξηγήσουμε την επίδραση των μεταβλητών αυτών στο Spark:

- **Ο Ρόλος των Cores (Πυρήνων):** Η αύξηση των cores αυξάνει την παραλληλία. Στο Πείραμα 1, το σύστημα μπορούσε να επεξεργάζεται ταυτόχρονα μόνο 2 tasks (1 ανά Executor). Στο Πείραμα 3, η παραλληλία τετραπλασιάστηκε σε 8 tasks ταυτόχρονα , δηλαδή 4 ανά Executor. Κάνοντας χρήση λοιπόν πολλών cores, οι executors μπορούν ταυτόχρονα να επεξεργάζονται περισσότερα partitions από δεδομένα πετυχαίνοντας έτσι **αύξηση του Throughput** (tasks per second). Τελικά, τα ενδιάμεσα αποτελέσματα αναδιανέμονται μέσω του δικτύου **(Shuffle)** ώστε να ομαδοποιηθούν ανά αστυνομικό τμήμα και να υπολογιστούν τα τελικά αθροίσματα.

- **Ο Ρόλος της Μνήμης:** Στο συγκεκριμένο πείραμα, ο ρόλος της μνήμης ήταν να φιλοξενήσει τον πίνακα των Αστυνομικών Τμημάτων (που στάλθηκε μέσω Broadcast) και να παρέχει τον απαραίτητο χώρο για τους γεωμετρικούς υπολογισμούς αποστάσεων. Γενικά, όσο μεγαλύτερη η μνήμη τόσο περισσότερα δεδομένα χωρούν σε αυτή και **έτσι δεν χρειάζεται μετακίνηση δεδομένων από και προς στον δίσκο**.

Παρατήρηση 1: Παρατηρούμε μια μείωση του χρόνου εκτέλεσης καθώς αυξάνονται οι πόροι (από 2.23s σε 1.97s), ωστόσο η βελτίωση δεν είναι γραμμική (δηλαδή ο τετραπλασιασμός των πυρήνων δεν έφερε υποτετραπλασιασμό του χρόνου). Αυτό ενδεχομένως να οφείλεται στο γεγονός ότι η μνήμη 2GB ήταν ήδη αρκετή για να χωρέσει το μικρό dataframe των αστυνομικών τμημάτων και άρα η αύξηση της μνήμης συνεισέφερε μόνο κατά την αποθήκευση ενδιάμεσων αποτελεσμάτων κατα τα aggregations και δεν μείωσε δραματικά τον χρόνο εκτέλεσης του query.

Παρατήρηση 2: Επιπρόσθετα, ένας παράγοντας για τη μη γραμμική βελτίωση των χρόνων με την αύξηση των cores πιθανόν είναι το Overhead, δηλαδή η διαδικασία επικοινωνίας μεταξύ Executors, το Task Scheduling, η σειριοποίηση δεδομένων, η αρχική μεταφορά δεδομένων κλπ, διαδικασίες που ούτως η άλλως δεν βελτιώνονται με την αύξηση των cores και δεν επιδέχονται κάποια παραλληλία.

3.5 Query 5: Συσχέτιση Εισοδήματος και Εγκληματικότητας

Χρησιμοποιώντας ως αναφορά τα δεδομένα της απογραφής του 2020 για τον πληθυσμό και τα οικονομικά στοιχεία του 2021 για το εισόδημα ανα νοικοκυριό, να υπολογίσετε μέσα στη διετία 2020-2021 τη συσχέτιση μέσου ετήσιου κατακεφαλήν εισοδήματος με την ετήσια μέση αναλογία εγκλημάτων ανά άτομο για κάθε περιοχή του Λος Άντζελες. Επαναλάβετε τον υπολογισμό εξετάζοντας μόνο τις 10 περιοχές με το υψηλότερο και τις 10 με το χαμηλότερο ετήσιο κατακεφαλήν εισόδημα.

3.5.1 All Communitites

All Communities	Executors	Cores per Executor	RAM per Executor	Συνολικά Cores	Συνολική μνήμη	Χρόνος (sec)
2	4	8 GB	8	16 GB	16 GB	86.16
4	2	4 GB	8	16 GB	16 GB	82.58
8	1	2 GB	8	16 GB	16 GB	70.97

Πίνακας 3: Πειραματισμός πόρων: Όλα τα communities

Υλοποίηση:

Στο query αυτό συνδυάζονται χωρικά και μη χωρικά δεδομένα. Πραγματοποιείται ένα **attribute join (βάσει ZIP code/ZCTA)**, για να αποδοθεί εισόδημα σε κάθε οικοδομικό τετράγωνο (block) και στη συνέχεια **aggregation (sum πάνω στα blocks)**

για τον υπολογισμό του κατά κεφαλήν εισοδήματος **ανά κοινότητα (COMM)** διαιρούμενο βέβαια και από τον συνολικό πλυθησμό του community (ο οποίος προέκυψε από **aggregation πάνω στο blocks population)** εφόσον πρόκειται για κατακεφαλήν. Έπειτα, τα εγκλήματα φιλτράρονται χρονικά (2020-2021) και μετατρέπονται σε **γεωμετρικά σημεία** μέσω των συντεταγμένων τους. Ακολουθεί το **χωρικό join (STContains)** για να εντοπιστεί σε ποιο Block (και κατ' επέκταση σε ποιο Community) ανήκει κάθε έγκλημα. Τέλος, τα δεδομένα **ομαδοποιούνται ανά Community (groupBy COMM)**, υπολογίζονται οι δείκτες εγκληματικότητας και συνενώνονται με τα δεδομένα εισοδήματος για τον τελικό υπολογισμό της συσχέτισης.

Παρατήρηση: Στα πρώτα αποτελέσματά μας, λάβαμε σε κάποιες πόλεις NULL τιμές οι οποίες προέκυψαν προφανώς λόγω του LEFT JOIN πάνω στα Zip Codes. Οι τιμές αυτές μετατράπηακν σε μηδέν (0).

Execution Plan:

1. **Attribute Join (Income-Blocks Join)** Επιλέχθηκε η στρατηγική **Broadcast Hash Join**. Το Spark αναγνώρισε ότι το DataFrame των εισοδημάτων είναι αρκετά μικρό, οπότε το έκανε Broadcast σε όλους τους Executors, αποφεύγοντας το Shuffle. **Αναλυτικά η διαδικασία του Broadcast Hash Join αναλύθηκε στο Query 3 παραπάνω**.

2. **Spatial Join (Crimes and Blocks)** Αναλύοντας τα Execution Plans, παρατηρούμε ότι στο ενδιάμεσο στάδιο (Explain 2) ο Optimizer αρχικά προσανατολιζόταν σε στρατηγική **Broadcast Nested Loop Join**. Ωστόσο, στο τελικό Physical Plan του ολοκληρωμένου query (Explain 3), στο σημείο που αφορά τον υπολογισμό των εγκλημάτων, εντοπίζεται η χρήση **RangeJoin**. Αυτό ενδεχομένως συμβαίνει λόγω του **AdaptiveSparkPlan isFinalPlan=false** που παρατηρούμε στην αρχή του explain που σημαίνει ότι το Spark μπορεί να αλλάζει ενδιάμεσα επιλεγμένα execution plans όταν αποφασίσει ότι πρέπει, εφόσον έχει "δει" όλο το query. Το **Range Join** είναι η τεχνική που χρησιμοποιεί το Spark μέσω της Sedona όταν έχει χτίσει τα **χωρικά indexes**.Τα χωρικά ευρετήρια μας επιτρέπουν πρώτα μια high level αναζήτηση ενός ευρύτερου block-χώρου στο οποίο ανήκει ένα γεωγραφικό σημείο. Με αυτό τον τρόπο, αποφεύγεται η σύγκριση ενός σημείου με καθένα από τα εκατομμύρια blocks του dataset κάνοντας το query αρκετά πιο γρήγορο αφού **συγκρίνουμε τελικά με τα πολύ λιγότερα αλλά πιθανά blocks**.

3. **Final Join (Income and Crimes)** Για την τελική ένωση των αποτελεσμάτων (Income per Comm + Crime per Comm), επιλέχθηκε το **Sort Merge Join** από τον optimizer, το οποίο **περιγράφεται αναλυτικά στο Query 3 παραπάνω**. Περιέργως, **δεν επέλεξε Broadcast Hash Join** το οποίο θα ήταν μία καλή τακτική από τη στιγμή που το μέγεθος των dataframes είναι ίσο με το - μικρό θα λέγαμε - πλήθος των communities. Από την άλλη, έχοντας πραγματοποιήσει ήδη όλα τα επιμέρους group by COMM στα προηγούμενα βήματα, **τα δεδομένα έχουν γίνει**

ήδη shuffle μεταξύ των executors και άρα ένα sort και ένα merge δεν αποτελούν ιδιαίτερα κοστοφόρα βήματα οπότε ίσως για αυτό επιλέχθηκε ο Sort Merge αλγόριθμος.

Configurations expirements:

Στο πείραμα αυτό διατηρήσαμε σταθερούς τους συνολικούς πόρους του συστήματος (Total Cores = 8, Total RAM = 16GB) αλλά **αλλάξαμε την κατανομή τους μεταξύ Executors και Cores**.

Σύγκριση Config 1 (2x4) VS Config 2 (4x2) VS Config 3 (8x1):

Όπως μπορούμε να δούμε από το explain, τα join strategies είναι ακριβώς ίδια με τα παραπάνω. **Η αλλαγή δηλαδή των configurations δεν επέφερε κάποια αλλαγή στον optimiazer**. Παρατηρούμε ότι οι χρόνοι είναι **κοντινοί (86.16s έναντι 82.58s, 70.97 s)**, με τις αποδόσεις να γίνονται σταδιακά ελαφρώς καλύτερες όσο προχωράμε στα configs 1,2 και 3.

Μία σημαντική παρατήρηση είναι ότι με την αύξηση των Executors και τη μείωση των πυρήνων ανά Executor, **η κοινόχρηστη μνήμη διαμοιράζεται πλέον σε λιγότερους πυρήνες** (2 αντί για 4). Αυτό **μειώνει τον ανταγωνισμό** μεταξύ των cores για τη μνήμη, **αποφεύγοντας καθυστερήσεις**, γεγονός που φαίνεται και στον βελτιωμένο τελικό χρόνο. **Με λίγα λόγια πλέον μοιράζονται την μνήμη λιγότεροι πυρήνες**.

Επιπλέον το μέρος του query είναι **Geospatial**, δημιουργεί **πολλά ενδιάμεσα αντικείμενα**(points, coordinates κλπ) που ζουν για ελάχιστο χρόνο. Σε τέτοιες περιπτώσεις, είναι καλύτερο να έχουμε μικρότερες μνήμες που καθαρίζονται πιο γρήγορα (Config 3), παρά μεγάλες μνήμες που καθαρίζουν πιο αργά(Config 1).

Bonus:

Για την ποσοτική αξιολόγηση της σχέσης μεταξύ του κατά κεφαλήν εισοδήματος και της εγκληματικότητας ανά κάτοικο κάθε περιοχής, επιλέχθηκε η χρήση της ενσωματωμένης συνάρτηση stat.corr() του Apache Spark. Η τιμή του συντελεστή κυμαίνεται στο διάστημα [-1, 1], όπου τιμές κοντά στο **-1** υποδηλώνουν ισχυρή αρνητική συσχέτιση (δηλαδή περιοχές με υψηλότερο εισόδημα τείνουν να έχουν χαμηλότερη εγκληματικότητα), ενώ τιμές κοντά στο **0** υποδηλώνουν απουσία συσχέτισης. Τυπώνεται ο μέσος συντελεστής συσχέτισης ο οποίος είναι ίσος με **corr = -0.067**. Το **αρνητικό πρόσημο** υποδηλώνει μια ασθενή τάση όπου περιοχές με υψηλότερο κατά κεφαλήν εισόδημα εμφανίζουν **χαμηλότερη εγκληματικότητα**. Ωστόσο, επειδή η τιμή είναι εξαιρετικά κοντά στο μηδέν , συμπεραίνουμε ότι πρακτικά **δεν υπάρχει ουσιαστική γραμμική συσχέτιση**.

3.5.2 Top 10/ Bottom 10 Communities

Στο ερώτημα για τις **Top 10** περιοχές, για όλα τα configs η ανάλυση του Execution Plan αποκαλύπτει την λειτουργία του **Predicate Pushdown** από τον Catalyst Optimizer. Το Physical Plan δείχνει ότι το Spark **"έσπρωξε" το φίλτρο των 10 Commuities στην αρχή της εκτέλεσης**. Ως αποτέλεσμα, το **Spatial Join (RangeJoin)** εκτελέστηκε σε ένα υποσύνολο των γεωμετρικών δεδομένων (μόνο στα blocks των 10 περιοχών) και όχι σε ολόκληρο το dataset. Αυτό έχει ως αποτέλεσμα τη μείωση του χρόνου εκτέλεσης (**Config 1: 83.46 s έναντι 86.16s που είχαμε στο all communities query**) σε σχέση με το all communities query. Επιπλέον, η τελική ένωση πάνω στα COMM έγινε και πάλι με Sort Moerge Join και η ένωση για το income με Broadcast Hash join όπως και πριν.

Δηλαδή η σειρά των joins είναι πάλι Broadcast Hash Join → Range Spatial Join → Sort Merge Join μόνο που τώρα έχει προηγηθεί ένα push down του φίλτρου στην αρχή.

Χρόνος (sec)	**Config**
Config 1	
All Communities	86.16
Top 10	83.46
Bottom 10	92.45
Config 2	
All Communities	82.58
Top 10	92.23
Bottom 10	83.29
Config 3	
All Communities	70.97
Top 10	94.98
Bottom 10	81.13

Πίνακας 4: Πειραματισμός πόρων: Top/Bottom 10 communities

Παρατήρηση:Αξίζει να τονίσουμε, ότι στον κώδικά μας, δεν γίνεται ρητά κατά σειρά φιλτράρισμα και έπειτα τα joins. Επιλέξαμε να το κάνουμε αυτό ώστε να δούμε εάν ο catalyst optimiazer εκτελέσει το pushdown filtering ακόμη και αν ο κώδικας είναι άκομψα γραμμένος με το φιλτράρισμα στο τέλος, κάτι που επιτυχώς έκανε!

Ωστόσο, φαίνεται ότι στα config 2 και 3, παρότι το query plan πάλι δείχνει pushdown του φίλτρου στην αρχή του δέντρου, είχαμε μεγαλύτερο τελικό χρόνο (92.23 s στο config 2 και 94.98 στο config 3) γεγονός το οποίο ενδεχομένως οφείλεται σε κάποιο πειραματικό σφάλμα ή απλώς σε αυτήν την εσκεμμένη γραφή του κώδικα.

Όσον αφορά το κομμάτι των **Bottom 10** περιοχών, δηλαδή αυτών με το μικρότερο εισόδημα, κάναμε ακριβώς τα ίδια και παρατηρούμε ότι το execution plan σε κάθε config παραμένει σταθερά το ίδιο:

Pushdown filtering → Broadcast Hash Join → Range Spatial Join → Sort Merge Join

Επιπλέον, εδώ πράγματι όσο οι πυρήνες λιγοστέυουν και οι executors αυξάνονται, για τους λόγους που εξηγούμε παραπάνω, πράγματι ο χρόνος γίνεται ολοένα και μικρότερος. Μπορείτε να δείτε σχετικά αποτελέσματα στον πίνακα σύνοψης: **Πίνακα 4**.