Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Προχωρημένα Θέματα Βάσεων Δεδομένων
9ο εξάμηνο, ακαδημαϊκό έτος 2025-2026

Εξαμηνιαία Εργασία

Ομάδα: group16

GitHub repository:
https://github.com/nspyrop03/adv-db-2025

Τάτση Αικατερίνη Δανάη, ΑΜ: 03121146

Σπυρόπουλος Νικολας, ΑΜ: 03121202

Περιεχόμενα

Σε όλα τα ερωτήματα όταν αναφερόμαστε στον πίνακα (ή dataset) των εγκλημάτων (ή "Crimes"), εννοούμε την ένωση (*"union"*) των δύο dataset εγκλημάτων του Los Angeles.

1 Query 1

Το πρώτο ερώτημα μας ζητείται να υλοποιηθεί με 3 διαφορετικούς τρόπους - χρησιμοποιώντας το DataFrame API μία φορά χωρίς και μία φορά με User Defined Function (UDF) και χρησιμοποιώντας το RDD API. Από την θεωρία ξέρουμε ότι όταν χρησιμοποιούμε το RDD API, το φυσικό πλάνο των πράξεων το ορίζουμε εμείς επακριβώς. Δεν μπορεί, δηλαδή, να παρέμβει ο Catalyst Optimizer. Από την άλλη, όταν χρησιμοποιούμε το DataFrame API, ο Catalyst παρεμβαίνει και βελτιστοποιεί το query μας. Όμως, στην περίπτωση μιας συνάρτησης ορισμένης από τον χρήστη (UDF), ο Catalyst την "βλέπει" ως ένα μαύρο κουτί και το συγκεκριμένο κομμάτι δεν μπορεί να το αλλάξει. Επομένως, περιμένουμε η γρηγορότερη υλοποίηση να είναι το DataFrame χωρίς UDF, στην συνέχεια το DataFrame με UDF και τέλος το RDD.

Πράγματι, μετρώντας τους χρόνους εκτέλεσης των τριών παραπάνω υλοποιήσεων επαληθεύουμε αυτό που αναμέναμε. Οι χρόνοι φαίνονται στον πίνακα 1. Παρατηρούμε ότι η μη χρήση UDF είναι περίπου $1.3s$ γρηγορότερη από την χρήση μιας UDF, ενώ η υλοποίηση με RDD είναι περίπου 2 φορές πιο αργή από τις υλοποιήσεις με το DataFrame API.

Πίνακας 1: Χρόνοι εκτέλης των τριών υλοποιήσεων του Query 1

Method	Time (s)
DF-no-UDF	7.23
DF-UDF	8.53
RDD	16.9

2 Query 2

Στόχος του δεύτερου ερωτήματος ήταν η επεξεργασία του συνόλου δεδομένων εγκληματικότητας ("Crime Data") και του συνόλου δεδομένων "Race and Ethnicity codes" με στόχο την εξαγωγή δημογραφικών στατιστικών ανά έτος. Πιο συγκεκριμένα, το ζητούμενο ήταν ο εντοπισμός των τριών φυλετικών ομάδων (Victim Descent) με τον μεγαλύτερο αριθμό θυμάτων για κάθε έτος. Επιπλέον για κάθε ομάδα έπρεπε να υπολογιστεί το ποσοστό των θυμάτων επί του συνολικού αριθμού εγκλημάτων του συγκεκριμένου έτους. Η υλοποίηση του ερωτήματος ζητείται να γίνει με δύο τρόπους, DataFrame και SQL APIs, προκειμένου να συγκρίνουμε την απόδοση τους.

Πίνακας 2: Χρόνοι εκτέλεσης των δύο υλοποιήσεων του Query 2

Method	Time (s)
DF	17.94
SQL	16.48

Παρατηρώντας τα αποτελέσματα διαπιστώνουμε ότι οι χρόνοι έχουν αμελητέα διαφορά. Επιβεβαιώνεται λοιπόν η αρχιτεκτονική του Spark SQL Engine. Ανεξάρτητα από το αν η υλοποίηση του προβλήματος γίνεται με DataFrames ή SQL και οι δύο προσεγγίσεις περνούν από τον Catalyst Optimizer. Το SQL query αλλά και οι εντολές DataFrame μεταφράζονται στο ίδιο φυσικό πλάνο και ο Catalyst Optimizer εφαρμόζει τις ίδιες βελτιστοποιήσεις. Επομένως η επιλογή μεταξύ DataFrame και SQL API δεν επηρεάζει την ταχύτητα της εκτέλεσης.

3 Query 3

Σε αυτό το ερώτημα πέρα από τα δύο μεγάλα datasets για τα εγκλήματα, χρησιμοποιούμε και το dataset "MO Codes" για την λεκτική περιγραφή κάθε κωδικού εγκλήματος. Επομένως, είναι αναγκαίο να ενώσουμε (join) τα δύο datasets μας για το σωστό σχηματισμό του query. Μπορούμε να δούμε ότι το dataset των εγκλημάτων είναι πολύ μεγαλύτερο από το dataset των κωδικών. Από την θεωρία γνωρίζουμε ότι για τέτοιες περιπτώσεις, ο αλγόριθμος Broadcast Hash Join είναι ο κατάλληλος. Πράγματι, χρησιμοποιώντας την συνάρτηση *"explain"* στο τελικό dataframe object βλέπουμε ότι το Spark χρησιμοποιεί αυτόν τον αλγόριθμο χωρίς να του δώσουμε καμία εντολή ή παράμετρο επιπλεόν.

Αρχικά, μας ζητείται η σύγκριση δύο υλοποιήσεων - μία με DataFrame API και μία με RDD API. Όπως έχουμε ήδη αναφέρει, μόνο στην πρώτη θα αξιοποιηθεί ο Catalyst optimizer και αυτό φαίνεται και από τους χρόνους που λαμβάνουμε στον πίνακα 3. Η υλοποίηση με DataFrame είναι περίπου $4s$ πιο γρήγορη.

Πίνακας 3: Χρόνοι εκτέλεσης των δύο πρώτων υλοποιήσεων του Query 3

Method	Time (s)
DF	14.2
RDD	18.1

Στην συνέχεια μας ζητείται να χρησιμοποιήσουμε την συνάρτηση *"hint"* ώστε να ωθήσουμε το Spark να χρησιμοποιήσει διαφορετικούς αλγορίθμους για το join των δύο πινάκων. Έτσι, χρησιμοποιούμε την συνάρτηση καλώντας την από τον μικρό πίνακα και παίρνουμε τις μετρήσεις του πίνακα 4. Παρατηρούμε ότι οι χρόνοι εκτέλεσης μεταξύ των διαφορετικών στρατηγικών Join (SortMerge, ShuffleHash, Broadcast) παρουσιάζουν πολύ μικρές αποκλίσεις (εύρος περίπου $0.7s$). Αυτό το φαινομενικά παράδοξο αποτέλεσμα εξηγείται από τη φύση των δεδομένων μας.

Το dataset "MO Codes" είναι αρκετά μικρό σε μέγεθος. Ως αποτέλεσμα, το κόστος μεταφοράς του (network shuffle) ή ταξινόμησής του (sort) είναι αμελητέο συγκριτικά με τον χρόνο που απαιτείται για την ανάγνωση (I/O) και επεξεργασία του κύριου dataset των εγκλημάτων. Μπορούμε, λοιπόν, να θεωρήσουμε ότι το μεγαλύτερο χρονικό μέρος καταναλώνεται από το διάβασμα του μεγάλου πίνακα (CSV read operations) και όχι από τη διαδικασία του Join.

Στον πίνακα 4 αναγράφεται "CartesianProduct" η ονομασία του join όταν δώσαμε στην συνάρτηση *"hint"* την παράμετρο *"SHUFFLE_REPLICATE_NL"*.

Πίνακας 4: Χρόνοι εκτέλεσης των διαφορετικών join

Join	Time (s)
SortMergeJoin	14.9
ShuffleHashJoin	14.5
CartesianProduct	14.3

4 Query 4

Για την υλοποιήση του τέταρτου ερωτήματος επιλέξαμε το DataFrame API. Όσον αφορά τους αστυνομικούς σταθμούς, χρησιμοποιούμε το dataset "LA Police Stations". Έτσι, χρειάστηκε να κάνουμε ένα cross join μεταξύ του πίνακα των εγκλημάτων και του πίνακα των σταθμών. Χρησιμοποιώντας την συνάρτηση *"explain"*, βλέπουμε ότι το Spark χρησιμοποιεί τον αλγόριθμο **BroadcastNestedLoopJoin** για αυτή την ένωση στο Physical Plan του. Με αυτό τον αλγόριθμο, το μικρό dataset των σταθμών γίνεται broadcast σε όλους τους executors και στην συνέχεια με nested loops και όχι hash function γίνεται η ένωση. Θεωρούμε ότι αυτή η επιλογή είναι καλή καθώς ο μικρός πίνακας είναι ικανός να χωρέσει στην μνήμη του κάθε executor.

Στην συνέχεια, μας ζητείται από την εκφώνηση να τρέξουμε το query για 3 διαφρορετικά configurations ώστε να μελετήσουμε την κλιμάκωση του κώδικά μας. Σε κάθε configuration χρησιμοποιούμε 2 executors και παίρνουμε τους χρόνους που φαίνονται στον πίνακα 5.

Πίνακας 5: Χρόνοι εκτέλεσης των διαφορετικών configuration

Configuration	Time (s)
1 core, 2GB mem	38.1
2 cores, 4GB mem	20.3
4 cores, 8GB mem	16.1

Παρατηρώντας τα αποτελέσματα του Πίνακα 5, διαπιστώνουμε ότι η αύξηση των υπολογιστικών πόρων οδηγεί σε μείωση του χρόνου εκτέλεσης, με τον βαθμό βελτίωσης να ποικίλλει ανάλογα με το configuration:

1. **Μετάβαση από Config 1 (1 core/2GB) σε Config 2 (2 cores/4GB)**: Παρατηρούμε σχεδόν γραμμική βελτιώση του χρόνου. Δίνοντας τους διπλάσιους πόρους το query μας εκτελείται περίπου στον μισό χρόνο. Αυτό δείχνει ότι το query, σε αυτό το στάδιο, είναι CPU-bound κάτι που περιμέναμε από την στιγμή που χρησιμοποιείται ένα nested loop join. Έτσι, με τους διπλάσιους πυρήνες, το Spark καταφέρνει να παραλληλοποιήσει αποτελεσματικά τον φόρτο εργασίας.

2. **Μετάβαση από Config 2 (2 cores/4GB) σε Config 3 (4 cores/8GB)**: Σε αυτό το στάδιο, η αύξηση των πόρων δεν οδηγεί σε κάποια αξιόλογη επιτάχυνση. Παρόλο που διπλασιάσαμε ξανά τους πόρους, ο καινούριος χρόνος εκτέλεσης είναι λιγότερο από $4s$ μικρότερος. Για να εξηγήσουμε αυτό το φαινόμενο χρειάζεται να επικαλεστούμε τον νόμο του Amdahl, σύμφωνα με τον οποίο το μη παραλληλοποιήσιμο τμήμα της εργασίας θέτει ένα άνω όριο στη μέγιστη δυνατή επιτάχυνση - ένα όριο που φαίνεται να το φτάσαμε. Θεωρούμε ότι αυτό συμβαίνει καθώς, πλέον το query έχει μετατραπεί σε "I/O-bound" δηλαδή περιορίζεται από την ταχύτητα ανάγνωση του μεγάλου dataset και εγγραφής των αποτελεσμάτων από και προς τον δίσκο.

Συμπερασματικά, η κλιμάκωση είναι αποδοτική μέχρι το Config 2 (2 cores/4GB), καθώς για το Config 3 βλέπουμε πως δεν υπάρχει ανάλογη επιτάχυνση και θα ήταν σπατάλη των περαιτέρω πόρων.

5 Query 5

Το πέμπτο ερώτημα είναι πιο σύνθετο καθώς συνδιάζει τρία σύνολα δεδομένων, το "Crimes" το "Census Blocks" και το "Median Household Income by Zip Code", για την εξαγωγή κοινωνικοοικονομικών συμπερασμάτων. Στόχος είναι η εύρεση της συσχέτισης μεταξύ του μέσου κατά κεφαλήν εισοδήματος με την ετήσια μέση αναλογία εγκλημάτων ανά άτομο σε κάθε περιοχή του Los Angeles. Για την υλοποίηση του ερωτήματος απαιτείται χρήση της βιβλιοθήκης Apache Sedona για τη δημιουργία Spatial Joins, ώστε να αντιστοιχιστεί κάθε έγκλημα στην περιοχή που ανήκει. Η ανάλυση του Physical Plan αποδεικνύει ότι ο optimizer λειτούργησε βέλτιστα επιλέγοντας στρατηγικές Broadcast για την ελαχιστοποίηση της κίνησης δεδομένων στο δίκτυο, αλλά και τη δημιουργία ευρετηρίου R-Tree στα δεδομένα αναφοράς για την γεωχωρική αναζήτηση. Καθοριστικό ρόλο στην απόδοση έπαιξε ο μηχανισμός Adaptive Query Execution. Κατά τη διάρκεια της εκτέλεσης, το AQE ανίχνευσε ότι το μέγεθος των δεδομένων προς ένωση ήταν μικρότερο από το αναμενόμενο. Έτσι, παρενέβη στο φυσικό πλάνο και αντικατέστησε αυτόματα το προγραμματισμένο SortMergeJoin με ένα Broadcast Join, εξαλείφοντας την ανάγκη για περιττό σάρωμα και επιταχύνοντας το ερώτημα.

Η εκτέλεση του ερωτήματος πραγματοποιήθηκε με 3 διαφορετικά configurations, διατηρώντας σταθερούς τους συνολικούς πόρους και αλλάζοντας την κατανομή. Παίρνουμε τους χρόνους που φαίνονται στον πίνακα 6.

Πίνακας 6: Χρόνοι εκτέλεσης των διαφορετικών configuration

Configuration	Time (s)
2 executors, 4 core, 8GB mem	49.22
4 executors, 2 cores, 4GB mem	54.48
8 executors, 1 cores, 2GB mem	55.67

Παρατηρούμε ότι το πρώτο configuration έχει καλύτερο χρόνο εκτέλεσης, ενώ τα άλλα δύο έχουν σχεδόν ίδιο. Παρόλο που το άθροισμα των υπολογιστικών πόρων παραμένει σταθερό σε όλα τα σενάρια, η υπεροχή του πρώτου configuration οφείλεται στη βέλτιστη διαχείριση των Broadcast Joins. Τα δεδομένα ανφοράς και το χωρικό ευρετήριο μεταφέρθηκαν και αποθηκεύτηκαν στην μνήμη μόνο 2 φορές, εξυπηρετώντας αποδοτικά τα threads του κάθε executor. Αντίθετα στις άλλες δύο περιπτώσεις απαιτείται μεταφορά και δέσμευση μνήμης για τις ίδιες δομές 4 και 8 φορές αντίστοιχα, αυξάνοντας το network traffic.