ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ

<u>Ομάδα 37</u>
Γιαννακόπουλος Στέφανος (03120829) 9ο Εξάμηνο
Κασιούμη Παρασκευή (03120122) 9ο Εξάμηνο

Github Repository:
https://github.com/stefanosGiannakopoulos/AdvancedDB_SemesterProject2024-2025

<u>Όλα τα αποτελέσματα για τα queries καθώς και τα ζητούμενα αρχεία, βρίσκονται στο public github repository που καταγράφεται παραπάνω.</u>

<u>ΣΕ ΟΛΕΣ ΤΙΣ ΧΡΟΝΟΜΕΤΡΗΣΕΙΣ ΛΑΒΑΜΕ ΥΠΟΨΙΝ ΤΟ LAZY EVALUATION</u>

Query 1ο

Στρατηγική

α) DataFrame

Ουσιαστικά, απλά ενώσαμε τα 2 csv για να έχουμε ένα ενιαίο dataframe στο οποίο κρατήσαμε μόνο τις περιπτώσεις όπου είχαμε βαριά σωματική βλάβη. Ύστερα, συναρτήσει της ηλικίας των θυμάτων τα ταξινομήσαμε στις ακόλουθες κατηγορίες (Children, Young Adults, Adults, Elderly, Invalid Age), όπου ομαδοποιήσαμε ανά ομάδα (group by), κάναμε καταμέτρηση των περιστατικών (περάσαμε aggregation count στο group) και κάναμε sorting σε φθίνουσα σειρά.

β) RDD

Κάναμε ό,τι και πριν απλά εδώ χρησιμοποιήσαμε map reduce. Ειδικότερα:

Map: Μετατροπή κάθε εγγραφής σε ζεύγος (key, value), με το key να είναι η ηλικιακή κατηγορία και το value να είναι 1.

Reduce: Ομαδοποίηση όλων των ζευγών με το ίδιο key και άθροισμα των value για κάθε κατηγορία.

Sort: Ταξινόμηση των αποτελεσμάτων κατά φθίνουσα σειρά.

Ζητούμενο 1ο

Για το DataFrame API απαιτήθηκαν 6.2 s για την εύρεση του τελικού αποτελέσματος, ενώ για το RDD API απαιτήθηκαν 15 s . Η διαφορά στον χρόνο εκτέλεσης μεταξύ του DataFrame API (6.2 δευτερόλεπτα) και του RDD API (15 δευτερόλεπτα) οφείλεται κυρίως στη βελτιστοποίηση που προσφέρει το DataFrame API μέσω του Catalyst Optimizer, ο οποίος εφαρμόζει τεχνικές όπως προώθηση φίλτρων, συγχώνευση λειτουργιών και δημιουργία αποδοτικών σχεδίων εκτέλεσης. Το DataFrame API χρησιμοποιεί αποδοτική αναπαράσταση δεδομένων (Internal Row) και εκτελεί λειτουργίες σε bytecode, ενώ το RDD API χειρίζεται γενικά αντικείμενα, που είναι πιο αργά και καταναλώνουν περισσότερη μνήμη. Επιπλέον, το DataFrame API περιορίζει την κίνηση δεδομένων στο δίκτυο και εκμεταλλεύεται καλύτερα τη συμπίεση και τη διαχείριση μνήμης. Αντίθετα, το RDD API προκαλεί περισσότερες λειτουργίες αναδιάταξης δεδομένων (shuffling) και δεν βελτιστοποιεί αυτόματα τις λειτουργίες, γεγονός που οδηγεί σε μεγαλύτερο χρόνο εκτέλεσης.

Query 2ο

Στρατηγική

α) i) DataFrame

Κάναμε ό,τι και πριν για την ανάγνωση του dataset , όπου δημιουργήσαμε μια νέα στήλη που χαρακτηρίζει κάθε υπόθεση ως κλειστή ή ανοικτή, με βάση το column (Status Desc). Ομαδοποιήσαμε τα δεδομένα ανά έτος και τμήμα, όπου υπολογίσαμε τον συνολικό αριθμό υποθέσεων, των κλεισμένων υποθέσεων και το ποσοστό επιλυμένων υποθέσεων. Με χρήση Window κάθε τμήμα κατατάσσεται ανά έτος με βάση το ποσοστό επιλυμένων υποθέσεων. Από το σύνολο των δεδομένων επιλέξαμε τα 3 κορυφαία τμήματα ανά έτος και τα ταξινομήσαμε.

ii) SQL

Αρχικά, το DataFrame που περιέχει τα δεδομένα μετατρέπεται σε view, ώστε να επιτρέπεται η εκτέλεση SQL queries πάνω στο σύνολο δεδομένων. Το query αποτελείται από τρία βασικά τμήματα.

Στο πρώτο τμήμα WITH processed_data, τα δεδομένα ομαδοποιούνται ανά έτος (Year) και τμήμα (AREA NAME), ενώ υπολογίζονται: ο συνολικός αριθμός υποθέσεων (total_cases), ο αριθμός κλεισμένων υποθέσεων (closed_cases) και το ποσοστό επιλυμένων υποθέσεων (closed_case_rate). Οι κλεισμένες υποθέσεις προσδιορίζονται μέσω CASE, που ελέγχει αν η κατάσταση της υπόθεσης (Status Desc) δεν είναι UNK ή Invest Cont. Το ποσοστό επιτυχίας υπολογίζεται ως το πηλίκο των κλεισμένων υποθέσεων προς το σύνολο των υποθέσεων, εκφρασμένο σε ποσοστό.

Στο δεύτερο τμήμα WITH ranked_data, τα rows από το προηγούμενο στάδιο κατατάσσονται για κάθε έτος, συναρτήσει του ποσοστού επιλυμένων υποθέσεων. Χρησιμοποιούμε τη Window Function ROW_NUMBER(), η οποία κατατάσσει τα τμήματα ανά έτος (PARTITION BY year) με φθίνουσα ταξινόμηση του ποσοστού επιτυχίας (ORDER BY closed_case_rate DESC).

Στο κύριο query επιλέγονται τα κορυφαία 3 τμήματα για κάθε έτος, με βάση την κατάταξη (rank <= 3), και τα αποτελέσματα ταξινομούνται κατά έτος και κατάταξη. Το τελικό αποτέλεσμα περιλαμβάνει τα columns year, precinct, closed_case_rate και rank.

Ζητούμενο 2(α)

Το SQL API (4.96 s) είναι ταχύτερο από το DataFrame API (10.26 s) επειδή αξιοποιεί άμεσα τον Catalyst Optimizer, που εφαρμόζει βελτιστοποιήσεις πιο αποτελεσματικά. Η δηλωτική φύση του SQL API επιτρέπει στο Spark να παραλείψει περιττά ενδιάμεσα βήματα, ενώ το DataFrame API απαιτεί περισσότερη μετάφραση σε λογικό σχέδιο. Επιπλέον, το SQL API

εκμεταλλεύεται βελτιστοποιήσεις, όπως προώθηση φίλτρων, βελτιστοποίηση joins και χρήση μεταδεδομένων από τον Catalog Manager, για αποδοτικότερη εκτέλεση. Ειδικά για σύνθετα ερωτήματα, το SQL API είναι καλύτερο στη δημιουργία αποδοτικών σχεδίων εκτέλεσης χωρίς επιβάρυνση από ενδιάμεσες λειτουργίες.

β) Στρατηγική

Ό,τι κάναμε και στο α) i) χρησιμοποιώντας το DataFrame API, ωστόσο μετατρέψαμε το DataFrame σε parquet και διαβάσαμε από αυτό ξεκινώντας την χρονομέτρηση από την στιγμή που κάνουμε read το φτιαγμένο parquet μας.

Ζητούμενο 2(β)

Καταλήξαμε ότι η εκτέλεση με Parquet (4.39 s) είναι σημαντικά γρηγορότερη από την αντίστοιχη με CSV (10.26 s). Αυτό συμβαίνει γιατί το Parquet αποθηκεύει τα δεδομένα σε μορφή στήλης , γεγονός που επιτρέπει την πρόσβαση μόνο στις στήλες που απαιτούνται από ένα ερώτημα, μειώνοντας τον όγκο των δεδομένων που διαβάζονται. Αντίθετα, το CSV αποθηκεύει τα δεδομένα σε μορφή γραμμής, αναγκάζοντας την ανάγνωση ολόκληρης της γραμμής ακόμη και αν χρειάζονται μόνο ορισμένες στήλες. Επιπλέον, το Parquet υποστηρίζει ενσωματωμένη συμπίεση, καθιστώντας τα αρχεία μικρότερα σε μέγεθος, με αποτέλεσμα να απαιτείται λιγότερη μεταφορά δεδομένων από τον αποθηκευτικό χώρο στη μνήμη.

Ακόμα το Parquet υποστηρίζει την ύπαρξη μεταδεδομένων, όπως τύποι δεδομένων, στατιστικά στοιχεία και εύρη τιμών για κάθε στήλη. Αυτά τα μεταδεδομένα επιτρέπουν σε εργαλεία όπως το Spark να βελτιστοποιούν τα ερωτήματα. Αντίθετα, το CSV δεν περιέχει μεταδεδομένα, με αποτέλεσμα τα δεδομένα να πρέπει να αναλυθούν από την αρχή σε κάθε ανάγνωση, μια διαδικασία υπολογιστικά πιο απαιτητική λόγω της διάσπασης των τιμών και της αναγνώρισης τύπων δεδομένων.

Query 3ο

Στρατηγική

Θα υλοποιήσουμε το query χρησιμοποιώντας το DataFrame API. Μας ζητείται, για κάθε περιοχή (“COMM”) της πόλης του Los Angeles, να υπολογίσουμε το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο. Τα αποτελέσματα αυτά πρέπει να συγκεντρωθούν σε έναν πίνακα.

Υπολογισμός μέσου ετήσιου εισοδήματος ανά άτομο

Θα ξεκινήσουμε με τον υπολογισμό του μέσου ετήσιου εισοδήματος ανά άτομο. Θα χρειαστούμε τα σύνολα δεδομένων “2010 Census Blocks (Los Angeles County)” και “Median Household Income by Zip Code (Los Angeles County)”.

Το σύνολο δεδομένων “2010 Census Blocks (Los Angeles County)” περιέχει πληροφορίες για κάθε ένα οικοδομικό τετράγωνο που ανήκει στο Los Angeles County. Από αυτές τις πληροφορίες μόνο κάποιες μας είναι χρήσιμες. Πιο συγκεκριμένα, για κάθε οικοδομικό τετράγωνο θα χρειαστούμε:

- το zip code του (“ZCTA10”),
- το πλήθος των νοικοκυριών (“HOUSING10”) για το έτος 2010,
- το πλήθος των κατοίκων (“POP_2010”) για το έτος 2010,
- την περιοχή (“COMM”) στην οποία αυτό ανήκει,
- την πόλη (“CITY”) στην οποία αυτό ανήκει και
- τις συντεταγμένες – γεωμετρία που το περιγράφουν.

Έπειτα, μελετάμε το σύνολο δεδομένων “Median Household Income by Zip Code (Los Angeles County)”. Στο σύνολο αυτό, για κάθε ένα zip code του Los Angeles County, καταγράφεται το μέσο ετήσιο εισόδημα νοικοκυριού για το έτος 2015. Και αυτή είναι και η μόνη πληροφορία που θα χρειαστούμε από αυτό το σύνολο δεδομένων.

Αφού μελετήσαμε τη δομή των δύο συνόλων δεδομένων, είμαστε έτοιμοι να γράψουμε το πρώτο σκέλος του 3ου query, ακολουθώντας τα παρακάτω βήματα:

1. Διαβάζουμε τα δεδομένα του συνόλου “Median Household Income by Zip Code (Los Angeles County)”. Επεξεργαζόμαστε την στήλη που περιέχει το μέσο ετήσιο εισόδημα νοικοκυριού. Πιο συγκεκριμένα, αφαιρούμε τους χαρακτήρες “$” και “ , ” και μετατρέπουμε το string σε decimal. Ονομάζουμε το dataframe που προκύπτει: “income_df”.
2. Διαβάζουμε τα δεδομένα του συνόλου “2010 Census Blocks (Los Angeles County)”. Κρατάμε μονάχα τις εγγραφές που έχουν μη αρνητικό πλήθος νοικοκυριών και μη αρνητικό πλήθος κατοίκων. Από αυτές κρατάμε τις εγγραφές που αναφέρονται σε οικοδομικά τετράγωνα που ανήκουν στην πόλη του Los Angeles. Ονομάζουμε το τελικό dataframe: “non_zero”.
3. Ομαδοποιούμε τα οικοδομικά τετράγωνα του non_zero βάσει του ζεύγους (COMM, ZCTA10). Έπειτα, για κάθε ζεύγος (COMM, ZCTA10), υπολογίζουμε το συνολικό αριθμό

νοικοκυριών και το συνολικό αριθμό κατοίκων. Ονομάζουμε το τελικό dataframe: “population_agg”.

4. Πραγματοποιούμε το πρώτο μας join. Πρόκειται για ένα inner join μεταξύ των dataframes “population_agg” και “income_df”. Κάθε ένα από αυτά τα δύο dataframes έχει μία στήλη που περιέχει το zip code και με τη βοήθεια αυτής της στήλης πραγματοποιείται το join. Με αυτό το join, για κάθε ζεύγος (COMM, ZCTA10), υπολογίζουμε το γινόμενο:

   $$(συνολικό\ πλήθος\ νοικοκυριών) \cdot (μέσο\ ετήσιο\ εισόδημα\ νοικοκυριού)$$

   Το γινόμενο αυτό μας δίνει το άθροισμα όλων των εισοδημάτων για κάθε ζεύγος (COMM, ZCTA10). Κρατάμε, επίσης, αμετάβλητη και την στήλη που περιέχει το συνολικό αριθμό κατοίκων για κάθε ζεύγος (COMM, ZCTA10). Ονομάζουμε το τελικό dataframe: “comm_income”.
5. Ομαδοποιούμε το dataframe comm_income βάσει του COMM. Για κάθε COMM, υπολογίζουμε το άθροισμα όλων των εισοδημάτων και το συνολικό αριθμό των κατοίκων. Πλέον, αρκεί να πραγματοποιήσουμε τη διαίρεση:

   $$(\text{άθροισμα όλων των εισοδημάτων})/(\text{συνολικός αριθμός των κατοίκων})$$

   για να βρούμε το μέσο ετήσιο εισόδημα ανά άτομο για κάθε COMM της πόλης του Los Angeles. Ωστόσο, επιλέγουμε να πραγματοποιήσουμε τη διαίρεση αυτή αργότερα, αφού ξεκινήσουμε, τους υπολογισμούς και για το δεύτερο σκέλος του 3ου query.

<u>Αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο</u>

Για τον υπολογισμό της αναλογίας του συνολικού αριθμού εγκλημάτων ανά άτομο, θα χρειαστούμε, επιπλέον, τα δύο βασικά datasets με τα εγκλήματα του Los Angeles. Από αυτά τα datasets, θα μας χρειαστεί μονάχα το id του κάθε εγκλήματος (“DR_NO”) και οι συντεταγμένες του εγκλήματος αυτού (“LAT” και “LON”).

Παραθέτουμε τα βήματα που ακολουθήσαμε για την επίλυση του ερωτήματος:

1. Διαβάζουμε τα δεδομένα με τα εγκλήματα. Αφαιρούμε οποιαδήποτε εγγραφή αναφέρεται στο Null Island (δηλαδή έχει LAT = LON = 0). Χρησιμοποιώντας τις στήλες “LAT” και “LON” και τη συνάρτηση ST_POINT(), δημιουργούμε για κάθε έγκλημα ένα geometry object που αναπαριστά το σημείο – τοποθεσία του εγκλήματος. Ονομάζουμε το τελικό dataframe: “crime_data”
2. Έπειτα, πραγματοποιούμε το δεύτερό μας join. Το join αυτό γίνεται μεταξύ των dataframes non_zero και crime_data. Σκοπός μας είναι να βρούμε ποια εγκλήματα του crime_data έλαβαν χώρα εντός κάθε COMM του non_zero. Το αποτέλεσμα είναι ένα νέο dataframe με όνομα “crime_comm”. Κάθε row του νέου dataframe περιέχει ένα έγκλημα του crime_data και το COMM, εντός του οποίο συνέβη το έγκλημα αυτό.
3. Ομαδοποιούμε το dataframe “crime_comm” κατά COMM και, για κάθε COMM, μετράμε το πλήθος των εγκλημάτων που συνέβησαν σε αυτό. Ονομάζουμε το dataframe που προκύπτει: “num_of_crimes”.

Συνδυασμός και τελικά αποτελέσματα

Υλοποιούμε το τρίτο και τελευταίο join μας. Πρόκειται για ένα outer join μεταξύ των dataframes “num_of_crimes” και “comm_income”. Κάθε ένα από αυτά τα δύο dataframes διαθέτει μία στήλη COMM και πάνω σε αυτήν πραγματοποιείται το join. Με αυτό το join, για κάθε COMM, υπολογίζουμε τους λόγους:

$$(\text{άθροισμα όλων των εισοδημάτων})/(\text{συνολικός αριθμός των κατοίκων})$$

και

$$(\text{συνολικός αριθμός εγκλημάτων})/(\text{συνολικός αριθμός των κατοίκων})$$

Το τελικό dataframe ονομάζεται “crimes_income_per_comm” και περιέχει το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία εγκλημάτων ανά άτομο για κάθε περιοχή (“COMM”) της πόλης του Los Angeles. Οπότε, υλοποιήσαμε επιτυχώς το 3ο query.

Ζητούμενο 3ο

Τώρα, θα πειραματιστούμε με διάφορες στρατηγικές για τα τρία joins που γράψαμε και μελετήσαμε παραπάνω. Πιο συγκεκριμένα, θα χρησιμοποιήσουμε τις στρατηγικές:

- BROADCAST
- MERGE
- SHUFFLE_HASH
- SHUFFLE_REPLICATE_NL

Πρώτα, όμως, θα εξηγήσουμε πώς λειτουργεί θεωρητικά κάθε μία από τις προαναφερόμενες στρατηγικές join. Ας υποθέσουμε ότι έχουμε δύο πίνακες Α και Β και πραγματοποιούμε ένα join ανάμεσα σε αυτούς. Τοποθετούμε τον πίνακα Α στην αριστερή πλευρά του join (left – hand side table) και τον πίνακα Β στη δεξιά πλευρά του join (right – hand side table).

Broadcast Join

Στο spark, οι όροι broadcast join και broadcast hash join αντιστοιχούν στην ίδια στρατηγική. Το broadcast join χρησιμοποιείται κυρίως σε equi – joins (“=”). Πραγματοποιείται σε τρία στάδια:

1. Στάδιο broadcast:
   Όλες οι εγγραφές του πίνακα Β εκπέμπονται σε όλα τα executors (broadcast). Γι’ αυτό το λόγο, συνήθως στη δεξιά πλευρά του join, τοποθετείται ο πίνακας με το μικρότερο μέγεθος.

2. Στάδιο hashing:
   Σε κάθε executor, γίνεται hashing του πίνακα Β, συνήθως βάσει του join key.

3. Στάδιο joining:
   Κάθε executor διαθέτει ένα τμήμα (partition) του πίνακα Α. Οπότε, πραγματοποιεί το join μεταξύ του τμήματος του Α και του hash table του πίνακα Β.

Το broadcast join συνήθως είναι η πιο γρήγορη στρατηγική. Ένας λόγος είναι επειδή δε πραγματοποιεί κανένα sort ή merge. Επίσης, κάθε executor διαθέτει ένα αντίγραφο του πίνακα Β και, έτσι, αποφεύγεται η διαδικασία του shuffling, δηλαδή η επικοινωνία μεταξύ των executors για τη μεταφορά δεδομένων του πίνακα Β.

Ωστόσο, σε κάποιες περιπτώσεις το broadcast join μπορεί να αποδειχθεί χρονοβόρο. Πιο συγκεκριμένα, αν ο πίνακας Β έχει πολύ μεγάλο μέγεθος, τότε κάποια δεδομένα του θα πρέπει να αποθηκευτούν στο δίσκο. Η πρόσβαση στο δίσκο είναι πιο αργή από την πρόσβαση στην executor memory, με αποτέλεσμα η εκτέλεση του join, τελικά, να επιβραδύνεται. Επίσης, αν ο πίνακας Β έχει πολύ μεγάλο μέγεθος, τότε η εκπομπή του σε όλους τους executors μπορεί να επιβαρύνει το δίκτυο.

Επιπλέον, το broadcast join δεν μπορεί να εφαρμοστεί σε full outer ή right outer joins. Για το right outer join, θα χρειαζόταν να γίνει broadcast ο αριστερός πίνακας (Α), γιατί με το right outer join επεκτείνουμε τον δεξιό πίνακα (Β). Για το full outer join, θα έπρεπε να γίνει broadcast και των δύο πινάκων, όχι μόνο του Β, αλλά αυτό δεν είναι εφικτό από την στρατηγική broadcast join.

Shuffle Hash Join

Όταν και οι δύο πίνακες Α και Β έχουν πολύ μεγάλο μέγεθος, τότε δε συμφέρει να επιλέξουμε το broadcast join. Επίσης, όταν εφαρμόζουμε full outer ή right outer join, πάλι δεν μπορούμε να χρησιμοποιήσουμε την στρατηγική του broadcast join. Συνήθως, σε αυτές τις περιπτώσεις επιλέγουμε το shuffle hash join, το οποίο πραγματοποιείται σε τρία στάδια:

1. Στάδιο shuffling:
   Τα δεδομένα των πινάκων Α και Β μοιράζονται σε διαφορετικούς executors. Πάντα, όμως, τα δεδομένα των Α και Β με ίδιο join key βρίσκονται στον ίδιο executor.

2. Στάδιο hashing:
   Σε κάθε executor, γίνεται hashing των δεδομένων των πινάκων Α και Β ξεχωριστά και βάσει του join key.

3. Στάδιο joining:
   Σε κάθε executor, πραγματοποιείται το join μεταξύ των hash tables των δεδομένων Α και Β. Στο τέλος, συνδυάζονται όλα τα αποτελέσματα των executors.

Δεν απαιτεί sorting, αλλά απαιτεί shuffling και hashing, δύο διαδικασίες αρκετά χρονοβόρες και υπολογιστικά βαριές. Ωστόσο, όπως τονίσαμε και προηγουμένως, σε κάποιες περιπτώσεις προτιμάμε αυτήν την στρατηγική έναντι του broadcast join.

Merge Join

Στο spark, οι όροι merge join και sort merge join αντιστοιχούν στην ίδια στρατηγική. Το merge join χρησιμοποιείται μόνο σε equi – joins (“=”) και μπορεί να εφαρμοστεί σε όλα τα join types (inner, full outer κτλ). Πραγματοποιείται σε τρία στάδια:

1. Στάδιο shuffling:
   Και οι δύο πίνακες Α και Β γίνονται shuffle μεταξύ των executors, όπως περιγράψαμε στο ομώνυμο στάδιο του shuffle hash join.

2. Στάδιο sorting:
   Κάθε executor ταξινομεί (sort) τα δεδομένα που έχει από τους πίνακες Α και Β ξεχωριστά και βάσει του join key.

3. Στάδιο merging:
   Κάθε executor διατρέχει τα ταξινομημένα δεδομένα των Α και Β που έχει βάσει του join key και πραγματοποιεί το join. Στο τέλος, συνδυάζονται τα αποτελέσματα όλων των executors.

Όπως καταλαβαίνουμε, η στρατηγική αυτή εφαρμόζεται σε περιπτώσεις στις οποίες το join key είναι sortable και, άρα, μας τίθεται αυτό ο περιορισμός. Επίσης, αυτή η στρατηγική είναι πιο χρονοβόρα από το shuffle hash join, διότι διαθέτει κα ένα επιπλέον στάδιο: το sorting. Ωστόσο, προτιμάται έναντι του shuffle hash join, αν οι πίνακές μας δεν είναι τόσο μεγάλοι και, άρα, η ταξινόμησή τους (sorting) δεν είναι τόσο χρονοβόρα. Επίσης, προτιμάται εάν τα δεδομένα μας είναι ήδη ταξινομημένα.

Shuffle Replicate NL Join

Η στρατηγική shuffle replicate nl επιστρέφει το καρτεσιανό γινόμενο των δύο πινάκων Α και Β και είναι η πιο χρονοβόρα στρατηγική από όλες. Ωστόσο, είναι χρήσιμη για non – equi joins (“>=”, “<=”, …) ή για τις περιπτώσεις, στις οποίες δεν έχουμε κάποιο join condition.

Τώρα για κάθε ένα από τα τρία joins που γράψαμε στο query 3, θα χρησιμοποιήσουμε τις στρατηγικές που αναλύσαμε παραπάνω. Για να χρησιμοποιήσουμε τις διαφορετικές στρατηγικές, θα χρειαστούμε τη μέθοδο hint(). Ωστόσο, αν κάποια στρατηγική δεν μπορεί να εφαρμοστεί ή δε συμφέρει να εφαρμοστεί σε ένα join, τότε ο catalyst optimizer θα μας αγνοήσει και θα χρησιμοποιήσει μία default στρατηγική για το join αυτό. Για να δούμε ποια στρατηγική χρησιμοποιήθηκε τελικά στο join μας, χρησιμοποιούμε τη μέθοδο explain().

Πρώτο join – “comm_income”

Ο catalyst optimizer χρησιμοποιεί ως default την στρατηγική του broadcast join.

Με τη μέθοδο hint(), δοκιμάζουμε και τις άλλες στρατηγικές και, έπειτα, με τη μέθοδο explain() βλέπουμε αν, πράγματι, αυτές εφαρμόστηκαν. Για το συγκεκριμένο join υποστηρίζονται όλες οι στρατηγικές. Στον παρακάτω πίνακα φαίνονται οι χρόνοι εκτέλεσης που προκύπτουν για κάθε μία από αυτές. Σημειώνουμε ότι στο χρόνο εκτέλεσης έχουμε προσθέσει και τον χρόνο εκτύπωσης του αποτελέσματος.

Join Strategy	Χρόνος Εκτέλεσης
BROADCAST	9.85 sec
MERGE	12.69 sec
SHUFFLE_HASH	12.33 sec
SHUFFLE_REPLICATE_NL	10.50 sec

Για να μπορέσουμε να ερμηνεύσουμε τα αποτελέσματά μας, πρέπει να λάβουμε υπόψιν μας τα μεγέθη των δύο dataframes πάνω στα οποία πραγματοποιούμε το join. Πιο συγκεκριμένα, όπως έχουμε ήδη εξηγήσει το πρώτο join γίνεται μεταξύ των dataframes population_agg και income_df. Το population_agg διαθέτει 428 εγγραφές και το income_df διαθέτει 284 εγγραφές. Το dataframe income_df είναι μικρότερο από το population_agg και γι' αυτό το λόγο, τοποθετούμε το income_df στη δεξιά πλευρά του join.

Το income_df είναι σχετικά μικρό και το join μας είναι ένα equi – join. Οπότε, η στρατηγική του broadcast join φαίνεται σαν μια καλή επιλογή. Πράγματι, παρουσιάζει το χαμηλότερο χρόνο εκτέλεσης από όλες τις στρατηγικές.

Αμέσως καλύτερο χρόνο εκτέλεσης παρουσιάζει το shuffle replicate nl. Αυτό μας παραξενεύει, διότι, όπως έχουμε επισημάνει και παραπάνω, με την στρατηγική shuffle replicate nl, ουσιαστικά, υπολογίζουμε το καρτεσιανό γινόμενο των δύο πινάκων και, άρα, αναμένουμε ότι η στρατηγική αυτή θα είναι πολύ χρονοβόρα. Ωστόσο, βλέπουμε ότι αυτό δε συμβαίνει. Το shuffle replicate nl μπορεί να παρουσιάζει καλή επίδοση, επειδή οι πίνακές μας έχουν σχετικά λίγες εγγραφές, με αποτέλεσμα, τελικά, το καρτεσιανό γινόμενό τους να υπολογίζεται γρήγορα.

Τέλος, βλέπουμε ότι το shuffle hash join εμφανίζει ελαφρά καλύτερη επίδοση από το merge join, με αποτέλεσμα το merge join να είναι η πιο χρονοβόρα στρατηγική. Όπως έχουμε εξηγήσει, το merge join διαθέτει τα ίδια στάδια με το shuffle hash join και ένα ακόμη: το sorting. Τα δύο dataframes ταξινομούνται βάσει του join key, το οποίο στην περίπτωσή μας είναι το zip code. Πριν το join, τα δύο dataframes δεν ήταν ταξινομημένα και, άρα, η ταξινόμησή τους πραγματοποιείται κατά την εκτέλεση του join. Κατά συνέπεια, το merge join θα είναι πιο αργό από το shuffle hash join, αλλά και από τις υπόλοιπες στρατηγικές join.

<u>Δεύτερο join – "crime comm"</u>

Ο catalyst optimizer χρησιμοποιεί ως default την στρατηγική του range join. Το range join είναι μια ειδική στρατηγική join και δεν έχουμε εξηγήσει πώς λειτουργεί. Το range join χρησιμοποιείται σε joins, στα οποία το join condition αποτελείται από συνθήκες ανισότητας, από ένα attribute που πρέπει να βρίσκεται εντός ενός συγκεκριμένου εύρους τιμών. Για παράδειγμα, το παρακάτω είναι ένα range join:

```
SELECT *
FROM points JOIN ranges ON points.p BETWEEN ranges.start and ranges.end;
```

Υπενθυμίζουμε ότι με το δεύτερο join, βρίσκουμε ποια εγκλήματα συνέβησαν εντός κάθε COMM. Το join γίνεται με τη βοήθεια της συνάρτησης ST_WITHIN(), η οποία συνάρτηση για

κάθε COMM, γνωρίζει το γεωγραφικό σχήμα του COMM και αναζητεί ποια σημεία όπου συνέβησαν εγκλήματα ανήκουν μέσα στο γεωγραφικό σχήμα του COMM. Όπως καταλαβαίνουμε το join αυτό είναι ένα range join και ορθά ο catalyst optimizer επιλέγει αυτήν την στρατηγική για να το υλοποιήσει.

Τώρα, με τη μέθοδο hint(), θα προσπαθήσουμε να υλοποιήσουμε το join, χρησιμοποιώντας διαφορετικές στρατηγικές. Έπειτα, χρησιμοποιώντας τη μέθοδο explain(), θα επιβεβαιώσουμε αν, πράγματι, εφαρμόστηκαν οι στρατηγικές μας.

Το δεύτερο join υλοποιείται μεταξύ των dataframes crime_data και non_zero. Το crime_data αποτελείται από 3109880 εγγραφές, ενώ το non_zero αποτελείται από 30637 εγγραφές. Το dataframe "non_zero" είναι πιο μικρό σε μέγεθος και, άρα, τοποθετούμε αυτό στη δεξιά πλευρά του join.

Παρατηρούμε ότι οι στρατηγικές των merge join, shuffle join και shuffle replicate nl δεν εφαρμόστηκαν από τον catalyst optimizer. Πιο συγκεκριμένα, όπως έχουμε εξηγήσει, το merge join και το shuffle join χρησιμοποιούνται κυρίως σε equi – joins και, άρα, ο catalyst optimizer δεν τα προτιμά για το συγκεκριμένο join. Έπειτα, το shuffle replicate nl join δεν προτιμάται, διότι οι πίνακες έχουν πολύ μεγάλο μέγεθος, με αποτέλεσμα ο υπολογισμός του καρτεσιανού τους γινομένου να είναι πολύ χρονοβόρος και να μην συμφέρει.

Η μόνη διαφορετική στρατηγική που, τελικά, εφαρμόστηκε από τον catalyst optimizer ήταν το broadcast join.

Παρακάτω φαίνονται οι χρόνοι εκτέλεσης που λάβαμε. Σημειώνουμε ότι στο χρόνο εκτέλεσης έχουμε προσθέσει και το χρόνο εκτύπωσης των αποτελεσμάτων:

Join Strategy	**Χρόνος Εκτέλεσης**
BROADCAST	17.58 sec
RangeJoin	14.48 sec

Παρατηρούμε ότι η στρατηγική range join παρουσιάζει την καλύτερη επίδοση. Το broadcast join παρουσιάζει μεγαλύτερο χρόνο εκτέλεσης από το range join. Αυτό πιθανώς συμβαίνει, διότι το dataframe non_zero είναι αρκετά μεγάλο και η αντιγραφή του σε όλους τους executors επιβαρύνει το δίκτυο ή/ και οδηγεί σε αποθήκευση δεδομένων στο δίσκο.

<u>Τρίτο join – "crimes_income_per_comm"</u>

Ο catalyst optimizer χρησιμοποιεί ως default την στρατηγική του merge join.

Υπενθυμίζουμε ότι το τρίτο join είναι ένα outer join μεταξύ των dataframes comm_income και num_of_crimes και μας δίνει το πίνακα με τα τελικά αποτελέσματα του query 3. Ως outer join αναμένουμε ότι δε θα μπορεί να υλοποιηθεί με την στρατηγική του broadcast join. Και, πράγματι, όταν θέτουμε την στρατηγική ίση με broadcast join μέσω του hint() και, έπειτα, χρησιμοποιούμε τη μέθοδο explain(), διαπιστώνουμε ότι ο catalyst optimizer μας αγνόησε και εφάρμοσε merge join. Παρόμοια, όταν θέτουμε την στρατηγική σε shuffle replicate nl, ο

catalyst optimizer μας αγνοεί και εφαρμόζει merge join, ίσως γιατί το καρτεσιανό γινόμενο είναι πολύ ακριβό χρονικά και υπολογιστικά.

Η μόνη διαφορετική στρατηγική που μπορέσαμε να εφαρμόσουμε ήταν το shuffle hash join. Στον παρακάτω πίνακα συγκεντρώνουμε τα αποτελέσματά μας. Σημειώνουμε ότι στο χρόνο εκτέλεσης έχουμε προσμετρήσει και το χρόνο εκτύπωσης των αποτελεσμάτων.

Join Strategy	Χρόνος Εκτέλεσης
MERGE	17.17 sec
SHUFFLE_HASH	17.70 sec

Τα dataframes που συμμετέχουν σε αυτό το join διαθέτουν το καθένα 139 εγγραφές και, άρα, είναι σχετικά μικρά.

Παρατηρούμε ότι το merge join εμφανίζει την καλύτερη επίδοση. Όπως έχουμε εξηγήσει, το merge join διαθέτει τα ίδια στάδια με το shuffle hash join και ένα επιπλέον: το sorting. Όταν ξεκινά η εκτέλεση του merge join, οι πίνακες δεν είναι ταξινομημένοι και, άρα, η ταξινόμηση βάσει του join key (COMM) πρέπει να γίνει κατά την εκτέλεση του join. Ωστόσο, αυτό δε φαίνεται να επιβραδύνει την εκτέλεση του join, ίσως διότι οι πίνακες έχουν αρκετά μικρό μέγεθος (139 εγγραφές ο καθένας) και η ταξινόμησή τους γίνεται σχετικά γρήγορα.

Query 4ο

Στρατηγική

Υλοποιούμε αυτό το query χρησιμοποιώντας το DataFrame API. Μας ζητείται να βρούμε το φυλετικό προφίλ των καταγεγραμμένων θυμάτων εγκλημάτων (Vict Descent) για το έτος 2015 στις 3 περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα, αλλά και στις 3 περιοχές με το χαμηλότερο κατά κεφαλήν εισόδημα. Τα αποτελέσματα μας θα πρέπει να δοθούν σε δύο διαφορετικούς πίνακες.

Για αυτό το query, θα χρειαστούμε τα αποτελέσματα του 3ου query. Θα χρειαστούμε, επίσης, το βασικό dataset που περιέχει τα εγκλήματα που καταγράφηκαν στο διάστημα 2010 – 2019 και τα σύνολα δεδομένων “2010 Census Blocks (Los Angeles County)” και “Race and Ethnicity Codes”.

Το βασικό σύνολο εγκλημάτων περιέχει πληροφορίες για τα εγκλήματα που καταγράφηκαν στην πόλη του Los Angeles. Για κάθε έγκλημα, μας δίνεται το id του (“DR_NO”), η ημερομηνία καταγραφής του (“Date Rptd”), το φυλετικό προφίλ του θύματος (“Vict Descent”) και οι συντεταγμένες όπου σημειώθηκε το συγκεκριμένο έγκλημα (“LAT” και “LON”). Αυτές είναι οι στήλες που θα χρειαστούμε από αυτό το σύνολο δεδομένων.

Το σύνολο δεδομένων “2010 Census Blocks (Los Angeles County)” περιέχει πληροφορίες για όλα τα οικοδομικά τετράγωνα του Los Angeles County. Για κάθε οικοδομικό τετράγωνο, μεταξύ άλλων, μας δίνεται το όνομα της περιοχής (“COMM”), στην οποία αυτό ανήκει, το όνομα της πόλης (“CITY”) στην οποία αυτό ανήκει και οι συντεταγμένες – γεωμετρία του οικοδομικού τετραγώνου. Εμείς θα χρειαστούμε μονάχα αυτές τις πληροφορίες.

Τέλος, το σύνολο δεδομένων “Race and Ethnicity Codes” περιέχει τα κωδικά ονόματα κάθε φυλετικού γκρουπ και αντιστοιχίζει κάθε ένα από αυτά με το πλήρες όνομα του φυλετικού γκρουπ.

Αφού εξηγήσαμε τη δομή των συνόλων δεδομένων και τι χρειαζόμαστε από το καθένα, μπορούμε να προχωρήσουμε στην υλοποίηση του query. Ακολουθούμε, λοιπόν, τα παρακάτω βήματα:

1. Διαβάζουμε τα δεδομένα των dataset με τα εγκλήματα. Κρατάμε μόνο τα εγκλήματα, τα οποία δεν αναφέρονται στο Null Island, δεν περιέχουν null στην στήλη του φυλετικού γκρουπ και τα οποία καταγράφηκαν το 2015.
2. Διαβάζουμε το σύνολο δεδομένων “2010 Census Blocks (Los Angeles County)” και κρατάμε μονάχα τα οικοδομικά τετράγωνα που βρίσκονται στην πόλη του Los Angeles.
3. Διαβάζουμε τα δεδομένα του συνόλου “Race and Ethnicity Codes”.
4. Από τα αποτελέσματα του query 3, βρίσκουμε τις 3 περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα και τις 3 περιοχές με το χαμηλότερο κατά κεφαλήν εισόδημα και τις αποθηκεύουμε σε ένα dataframe. Για να ξεχωρίζουμε τις δύο ομάδες περιοχών, δημιουργούμε στο dataframe μας μία στήλη με όνομα “rank_label”. Αν η περιοχή ανήκει στην ομάδα με το υψηλότερο κατά κεφαλήν εισόδημα, το attribute “rank_label” παίρνει την τιμή “highest”. Αν η περιοχή ανήκει σε αυτές με το χαμηλότερο εισόδημα, τότε το “rank_label” παίρνει την τιμή “lowest”.
5. Πραγματοποιούμε ένα join μεταξύ του dataframe του βήματος 4 και των δεδομένων του συνόλου “2010 Census Blocks (Los Angeles County)”. Με αυτόν τον τρόπο, κατασκευάζουμε ένα νέο dataframe που θα περιέχει τα ονόματα των 3 περιοχών με το υψηλότερο κατά κεφαλήν, τα ονόματα των περιοχών με το χαμηλότερο κατά κεφαλήν εισόδημα, αλλά και τις συντεταγμένες – γεωμετρία όλων των οικοδομικών τετραγώνων που ανήκουν σε αυτές τις 6 προαναφερόμενες περιοχές. Προφανώς, κρατάμε και την στήλη “rank_label”.
6. Πραγματοποιούμε ένα join μεταξύ του dataframe του βήματος 5 και του dataframe του βήματος 1. Σκοπός μας είναι να βρούμε τα εγκλήματα που συνέβησαν εντός των 6 περιοχών που εξετάζουμε. Το τελικό dataframe θα περιέχει το id (“DR_NO”) του εγκλήματος, το φυλετικό γκρουπ του θύματος (“Vict Descent”), και το “rank_label” που μας υποδεικνύει αν το έγκλημα συνέβη είτε σε μία από τις περιοχές με το υψηλότερο εισόδημα είτε σε μία από τις περιοχές με χαμηλότερο εισόδημα.
7. Ομαδοποιούμε το dataframe του βήματος 6 κατά το ζεύγος (rank_label, Vict Descent). Έπειτα, για κάθε rank_label βρίσκουμε το πλήθος των θυμάτων που ανήκουν σε κάθε φυλετικό γκρουπ.

8. Πραγματοποιούμε join μεταξύ του dataframe του βήματος 7 και του συνόλου δεδομένων “Race and Ethnicity Codes”. Οπότε, πλέον, για κάθε rank_label, γνωρίζουμε το πλήθος των θυμάτων που ανήκουν σε κάθε φυλετικό γκρουπ, αλλά και την πλήρη ονομασία του φυλετικού γκρουπ.
9. Χωρίζουμε το dataframe του βήματος 9 σε δύο ξεχωριστά dataframes, με τη βοήθεια της στήλης rank_label. Το πρώτο dataframe περιέχει τα αποτελέσματά μας για τις 3 περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα και το δεύτερο dataframe περιέχει τα αποτελέσματά μας για τις 3 περιοχές με το χαμηλότερο κατά κεφαλήν εισόδημα.

Ζητούμενο 4ο

Αφού υλοποιήσαμε το query, τώρα θέλουμε να το εκτελέσουμε χρησιμοποιώντας 2 executors με τα ακόλουθα configurations:

- 1 core με 2GB memory
- 2 cores με 4GB memory
- 4 cores με 8GB memory

Καταγράφουμε το χρόνο εκτέλεσης για καθένα configuration και παρουσιάζουμε τα αποτελέσματά μας στον παρακάτω πίνακα:

A/A	No. of Executors	No. of Cores	Executor Memory	Χρόνος Εκτέλεσης
1	2	1	2 GB	130.28 sec
2	2	2	4 GB	114.95 sec
3	2	4	8 GB	109.15 sec

Σημειώνουμε ότι στο χρόνο εκτέλεσης έχουμε προσθέσει και το χρόνο ανάγνωσης των datasets.

Αν κατατάσσαμε τα configurations ως προς το χρόνο εκτέλεσης του query, τότε θα είχαμε:

$$exec\_time\_3 < exec\_time\_2 < exec\_time\_1$$

Άρα, με το 3ο configuration παρατηρούμε το μικρότερο χρόνο εκτέλεσης. Σε όλα τα configurations κρατάμε σταθερό τον αριθμό των executors και, γενικά, παρατηρούμε ότι κάθε φορά που αυξάνουμε το πλήθος των cores και το μέγεθος της executor memory, τότε μειώνεται και ο χρόνος εκτέλεσης. Η συμπεριφορά αυτή είναι λογική. Το συγκεκριμένο query που εξετάζουμε διαχειρίζεται δεδομένα πολύ μεγάλου μεγέθους και πραγματοποιεί σε αυτά χρονοβόρους υπολογισμούς, joins, aggregations κτλ. Συνεπώς, χρησιμοποιώντας πολλά cores, οι διάφοροι υπολογισμοί και οι εργασίες του query μοιράζονται στα διαφορετικά cores και εκτελούνται παράλληλα, με αποτέλεσμα να μειώνεται ο συνολικός χρόνος εκτέλεσης του query. Η αύξηση της executor μνήμης είναι, επίσης, ωφέλιμη και επιταχύνει την εκτέλεση του query. Χρησιμοποιώντας μεγαλύτερη executor memory, μπορούμε να αποθηκεύουμε όλο

και περισσότερα δεδομένα σε αυτήν. Άρα, γίνονται λιγότερες οι φορές που θα χρειαστεί να αποθηκεύσουμε ή να ανασύρουμε κάποιο δεδομένο από το δίσκο, ο οποίος παρουσιάζει πιο αργούς χρόνους πρόσβασης από την executor memory.

Ωστόσο, πρέπει να σημειώσουμε ότι η αύξηση των cores και της executor memory πέραν κάποιων ορίων, διαφορετικών για κάθε εφαρμογή, μπορεί, τελικά, να επιβραδύνει την εκτέλεση του query. Πιο συγκεκριμένα, αν χρησιμοποιήσουμε περισσότερα cores απ' ότι χρειάζεται, τότε θα παρουσιαστούν καθυστερήσεις λόγω του αυξημένου φόρτου διαχείρισης και χρονοπρογραμματισμού των διαφορετικών εργασιών. Παράλληλα, θα γίνουν πιο συχνές οι επικοινωνίες μεταξύ των cores, το δίκτυο θα επιβαρύνεται και το bandwidth δε θα επιτρέπει γρήγορες επικοινωνίες. Όσον αφορά την άλογη αύξηση της μνήμης, μπορεί να οδηγήσει σε σπατάλη πόρων, καθώς δε χρειαζόμαστε όλη τη μνήμη που δεσμεύουμε, αλλά μπορεί να οδηγήσει και σε resource contention με άλλες διεργασίες που εκτελούνται, επιβραδύνοντας το χρόνο εκτέλεσης της εφαρμογής μας. Παρ' όλα αυτά, στα configurations που εξετάσαμε δεν παρατηρήσαμε κανένα από αυτά τα φαινόμενα. Βλέπουμε, όμως, ότι το 3^{ο} configuration είναι ελάχιστα πιο γρήγορο από το 2^{ο} configuration και, ίσως, αυτό μας προϊδεάζει για το ότι η περαιτέρω αύξηση των cores και της executor memory μπορεί να μην προσφέρει επιπλέον οφέλη.

Query 5^{ο}

Στρατηγική

Αρχικά, τα δεδομένα εγκλημάτων φορτώνονται και φιλτράρονται ώστε να διατηρηθούν μόνο οι εγγραφές με έγκυρες συντεταγμένες και το DR_NO. Τα δύο σύνολα δεδομένων ενώνονται σε ένα ενιαίο DataFrame, και οι συντεταγμένες κάθε εγκλήματος μετατρέπονται σε αντικείμενα ST_Point μέσω της βιβλιοθήκης Sedona. Παράλληλα, φορτώνονται τα δεδομένα των αστυνομικών τμημάτων, όπου κρατούνται οι συντεταγμένες, το όνομα του τμήματος και το DR_NO. Κάνουμε drop τα άχρηστα columns, διότι στη συνέχεια θα εκτελέσουμε cross join και δεν χρειάζεται να επιβαρύνουμε με περιττές στήλες τα tuples που θα προκύψουν.

Στη συνέχεια, εκτελείται ένα cross join μεταξύ των δύο DataFrames, συνδέοντας κάθε έγκλημα (που το αντιμετωπίζουμε ως ένα σημείο) με όλα τα αστυνομικά τμήματα. Επειδή τα δεδομένα των τμημάτων είναι μικρά, χρησιμοποιείται broadcast join για να βελτιστοποιηθεί η απόδοση (κοινώς manually στέλνουμε εμείς τα rows που περιλαμβάνουν τα τμήματα στον πίνακα που έχει τα σημεία που έγινε κάθε έγκλημα). Υπολογίζεται η γεωγραφική απόσταση σε χιλιόμετρα μεταξύ του σημείου κάθε εγκλήματος και των αστυνομικών τμημάτων με τη χρήση της συνάρτησης ST_DistanceSphere. Για κάθε έγκλημα, εντοπίζεται το πλησιέστερο τμήμα μέσω ενός Window που ομαδοποιεί τα δεδομένα ανά DR_NO και ταξινομεί τα τμήματα κατά απόσταση σε αύξουσα σειρά. Κρατείται μόνο η εγγραφή με τη μικρότερη απόσταση για κάθε έγκλημα.

Στο τελικό στάδιο, τα δεδομένα ομαδοποιούνται ανά τμήμα και υπολογίζεται η μέση απόσταση των εγκλημάτων από το τμήμα, καθώς και ο συνολικός αριθμός εγκλημάτων που έχουν ανατεθεί σε αυτό. Τα αποτελέσματα ταξινομούνται κατά φθίνουσα σειρά του αριθμού εγκλημάτων που αντιμετώπισε (στην θεωρία) το καθένα.

Ζητούμενο 5ο

Τα αποτελέσματα των μετρήσεων αποδεικνύουν ότι η απόδοση του PySpark επηρεάζεται σημαντικά από την ισορροπία μεταξύ του αριθμού των executors των πυρήνων και της μνήμης, με μεγαλύτερη έμφαση προφανώς στη μνήμη.

Στο configuration με 2 εκτελεστές, 4 πυρήνες και 8 GB μνήμης, επιτυγχάνεται ο καλύτερος χρόνος εκτέλεσης. Αυτή η διαμόρφωση εξασφαλίζει επαρκή μνήμη ανά εκτελεστή, επιτρέποντας την αποδοτική φόρτωση και επεξεργασία δεδομένων στη μνήμη, μειώνοντας την ανάγκη για I/O στον δίσκο. Παράλληλα, οι 4 πυρήνες ανά executor προσφέρουν επαρκή υπολογιστική ισχύ για την παράλληλη επεξεργασία, ενώ ο μικρός αριθμός εκτελεστών περιορίζει την επικοινωνία μεταξύ τους και το shuffling, που είναι ένα από τα πιο κοστοβόρα στάδια του Spark.

Στο configuration με 4 executors, 2 πυρήνες και 4 GB μνήμης, ο χρόνος εκτέλεσης αυξάνεται σημαντικά . Οι 2 πυρήνες ανά executor περιορίζουν την παράλληλη επεξεργασία σε επίπεδο executor. Επιπλέον, η μειωμένη μνήμη ανά executor αυξάνει την ανάγκη για προσωρινή αποθήκευση στον δίσκο, ενώ ο μεγαλύτερος αριθμός εκτελεστών αυξάνει την επικοινωνία και το κόστος shuffling, επηρεάζοντας αρνητικά την απόδοση.

Παρόμοια απόδοση παρατηρείται και στο configuration με 8 executors, 1 πυρήνα και 2 GB μνήμης. Ο μεγάλος αριθμός executors με περιορισμένους πόρους ανά executor οδηγεί σε υπερβολική επικοινωνία μεταξύ των executors και αυξημένο κόστος shuffling. Η χαμηλή μνήμη δεν επαρκεί για τη φόρτωση και την επεξεργασία των δεδομένων, με αποτέλεσμα την αυξημένη ανάγκη για I/O στον δίσκο, επιδεινώνοντας περαιτέρω την απόδοση.

Συνολικά, τα αποτελέσματα δείχνουν ότι η βέλτιστη απόδοση επιτυγχάνεται με λιγότερους, αλλά πιο ισχυρούς εκτελεστές, που διαθέτουν επαρκή μνήμη και πυρήνες. Στη συγκεκριμένη περίπτωση, η διαμόρφωση με 2 executors, 4 πυρήνες και 8 GB μνήμης παρέχει την καλύτερη ισορροπία πόρων, ελαχιστοποιώντας το κόστος επικοινωνίας και I/O με δίσκο.