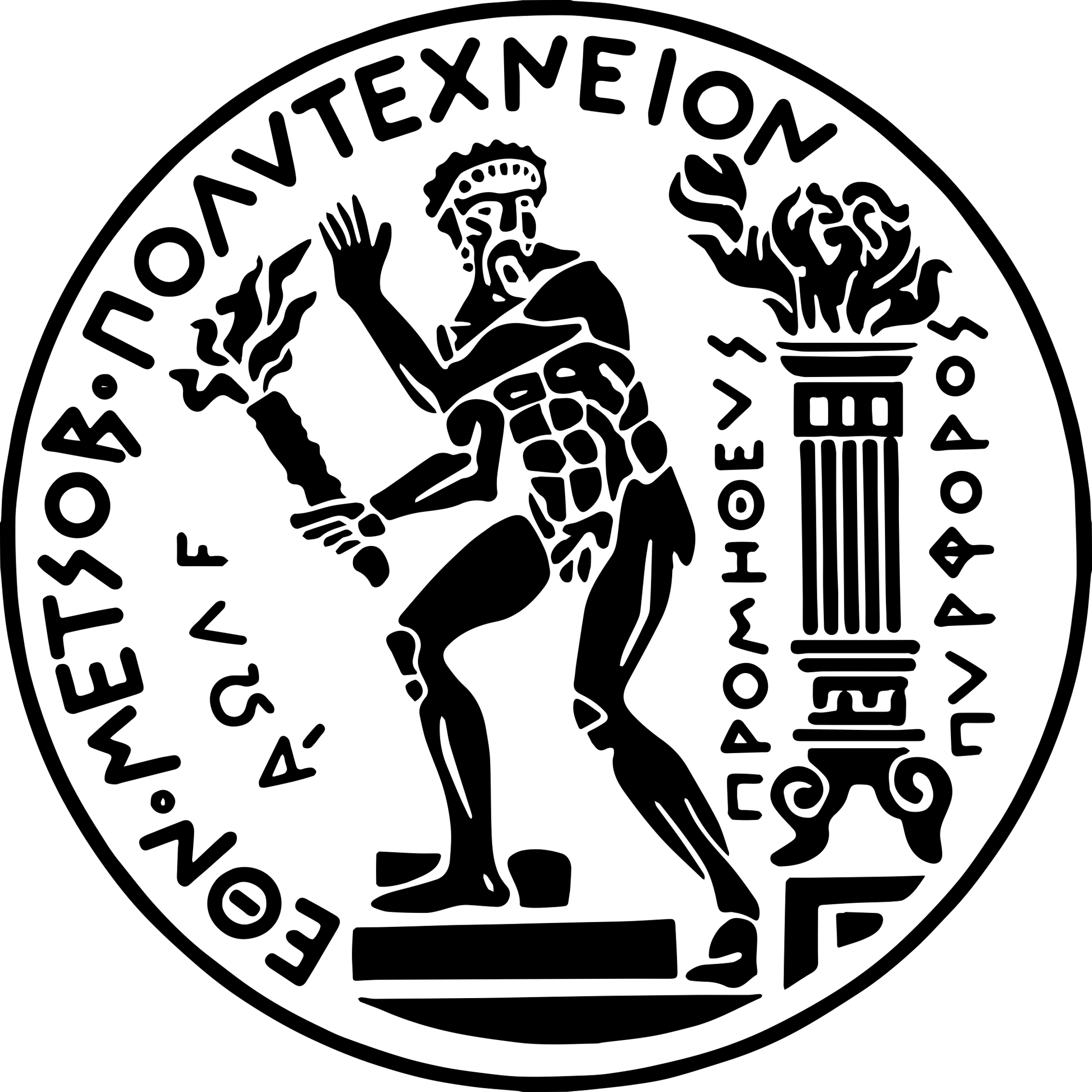
ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ



Σπουδαστής: Γιαννακόπουλος Στέφανος (03120829) 9ο Εξάμηνο

Σπουδάστρια: Κασιούμη Παρασκευή (03120122) 9ο Εξάμηνο

Αρ.Ομάδας: 37

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

***Όλα τα αποτελέσματα για τα queries καθώς και τα ζητούμενα αρχεία, βρίσκονται στο public github repository που καταγράφεται παραπάνω.***

ΣΕ ΟΛΕΣ ΤΙΣ ΧΡΟΝΟΜΕΤΡΗΣΕΙΣ ΛΑΒΑΜΕ ΥΠΟΨΙΝ ΤΟ LAZY EVALUATION

Queries:

1)

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:

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.

2)

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](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.html) 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()](https://www.codecademy.com/resources/docs/sql/window-functions/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.

Ζητούμενα:

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:

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.

3)

Ζητούμενο:

Για το πρώτο join:

Για το δεύτερο join:

Για το τρίτο join:

5)

Στρατηγική:

Αρχικά, τα δεδομένα εγκλημάτων φορτώνονται και φιλτράρονται ώστε να διατηρηθούν μόνο οι εγγραφές με έγκυρες συντεταγμένες. 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 και το όνομα του τμήματος, και μετατρέπονται και αυτά σε αντικείμενα ST\_Point. 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](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.html) που ομαδοποιεί τα δεδομένα ανά σημείο εγκλήματος 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:

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.

Στο configuration με **2 εκτελεστές, 4 πυρήνες και 8 GB μνήμης**, επιτυγχάνεται ο καλύτερος χρόνος εκτέλεσης (4.80 δευτερόλεπτα). 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 (15.26 δευτερόλεπτα). 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.

Η χειρότερη απόδοση παρατηρείται στο configuration με **8 executors, 1 πυρήνα και 2 GB μνήμης**, με χρόνο εκτέλεσης 20.36 δευτερόλεπτα. 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.