Αριστοτελειο Πανεπιστημιο Θεσσαλονικης

Διπλωματικη Εργασια

Μελέτη και Επεξεργασία Top-k Ερωτημάτων Κυριαρχίας με Apache Spark

Συγγραφέας:
Γεώργιος Οικονομίδης

Επιβλέπων Καθηγητής:
Απόστολος Ν. Παπαδόπουλος

Διπλωματική εργασία υποβαλλόμενη για την απόκτηση Μεταπτυχιακού Διπλώματος Ειδίκευσης

στο

Data Engineering Lab
Τμήμα Πληροφορικής

Φεβρουάριος 2017

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

Περίληψη

Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής

Μεταπτυχιακό Δίπλωμα Ειδίκευσης

Μελέτη και Επεξεργασία Top-k Ερωτημάτων Κυριαρχίας με Apache Spark

του Γεώργιου Οικονομίδη

Τα ερωτήματα top-k κυριαρχίας, τα οποία υπάγονται στην κατηγορία των ερωτημάτων προτίμησης, αποτελούν ένα πολύ ισχυρό εργαλείο για τον αναλυτή δεδομένων, καθώς του προσφέρουν έναν ευκολονόητο τρόπο εντοπισμού σημαντικών αντικειμένων μέσα σε ένα σύνολο δεδομένων. Ως εκ τούτου, αυτού του είδους τα ερωτήματατα βρίσκουν ευρεία εφαρμογή σε διάφορες περιοχές όπως υποστήριξη λήψης αποφάσεων, εξόρυξη δεδομένων, αναζήτηση στον ιστό και σε εφαρμογές πολυκριτηριακής ανάκτησης. Στην εποχή της πληροφορίας, όπου ο όγκος των δεδομένων συνεχώς αυξάνεται, οι υπάρχοντες σειριακοί αλγόριθμοι επεξεργασίας τέτοιων ερωτημάτων δεν μπορούν να ανταπεξέλθουν σε πραγματικά δεδομένα. Στην παρούσα εργασία προτείνεται ένας κατανεμημένος αλγόριθμος επεξεργασίας top-k ερωτημάτων κυριαρχίας, υλοποιημένος στο προγραμματιστικό περιβάλλον Apache Spark. Ο προτεινόμενος αλγόριθμος, στηριζόμενος σε έναν υπάρχων σειριακό, εκμεταλεύεται τη δύναμη του παραλληλισμού που προσφέρει το Spark και οδηγεί σε πολύ καλύτερους χρόνους εκτέλεσης.

Aristotle University of Thessaloniki

Abstract

Faculty of Sciences
School of Informatics

Master of Science

Processing of Top-k Dominating Queries using Apache Spark

by George Economides

Top-k dominating queries, a type of preference-based queries, are a powerful tool to the data analyst, as they provide an intuitive way for finding significant objects in a dataset. As a result, they are widely used in decision support systems, data mining, web search and multi-criteria retrieval applications. In the era of information, where data size continuously grows, existing sequential algorithms that process those queries cannot cope with real-world data anymore. In this thesis, a distributed algorithm for processing top-k dominating queries, implemented in Apache Spark, is proposed. The proposed algorithm, based on an existing sequential one, utilizes the power of parallelism offered by Spark and delivers way faster execution times.

Ευχαριστίες

Πριν την παρουσίαση των αποτελεσμάτων της παρούσας εργασίας, αισθάνομαι την υποχρέωση να ευχαριστήσω ορισμένους από τους ανθρώπους που γνώρισα, συνεργάστηκα μαζί τους και έπαιξαν πολύ σημαντικό ρόλο στην πραγματοποίησή της. Πρώτα απ' όλα θα ήθελα να ευχαριστήσω όλους τους διδάσκοντες του τμήματος για τη γνώση που μου προσέφεραν τον τελευταίο χρόνο, με την οποία έγινε εφικτή η πραγματοποίηση αυτής της εργασίας. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον επιβλέποντα διδάσκοντα μου, κ. αναπληρωτή καθηγητή Απόστολο Παπαδόπουλο. Πάντα φιλικός, προσιτός και συνεργάσιμος, μου έδωσε όλα τα απαραίτητα εφόδια για να ανταπεξέλθω στις όποιες δυσκολίες εμφανίστηκαν κατά τη διάρκεια της εργασίας. Η καθοδήγησή του έπαιξε καταλυτικό ρόλο, ενώ η ευκαιρία να συνεργαστώ μαζί του αποτελεί μια πολύ θετική εμπειρία στο χώρο της έρευνας. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για τη συνεχή υποστήριξη της, χάρη στην οποία οι συνθήκες σπουδών μου ήταν παραπάνω από ιδανικές.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Αντικείμενο της παρούσας εργασίας είναι η αποδοτική επεξεργασία ερωτημάτων top-k κυριαρχίας, χρησιμοποιώντας το προγραμματιστικό περιβάλλον Apache Spark. Τα ερωτήματα top-k κυριαρχίας, τα οποία υπάγονται στην κατηγορία των ερωτημάτων προτίμησης, αποτελούν ένα πολύ ισχυρό εργαλείο για τον αναλυτή δεδομένων, καθώς του προσφέρουν έναν ευκολονόητο τρόπο εντοπισμού σημαντικών αντικειμένων μέσα σε ένα σύνολο δεδομένων. Ως εκ τούτου, αυτού του είδους τα ερωτήματατα βρίσκουν ευρεία εφαρμογή σε διάφορες περιοχές, όπως υποστήριξη λήψης αποφάσεων, εξόρυξη δεδομένων, αναζήτηση στον ιστό και σε εφαρμογές πολυκριτηριακής ανάκτησης. Λόγω όμως του συνεχώς αυξανόμενου όγκου δεδομένων οι παραδοσιακοί κεντρικοποιημένοι αλγόριθμοι δεν είναι πλέον ικανοί να ανταπεξέλθουν σε πραγματικά δεδομένα. Το Apache Spark, μια πλατφόρμα υπολογισμών σε cluser, δίνει τη δυνατότητα επίλυσης τέτοιου είδους προβλημάτων, εκμεταλευόμενο τη δύναμη του παραλληλισμού. Με το Spark, είναι δυνατή η διαίρεση μιας διαδικασίας σε μικρότερα τμήματα, με κάθε τμήμα της να εκτελείται παράλληλα με τα υπόλοιπα σε διαφορετικούς κόμβους κάποιου cluster. Στόχος της εργασίας είναι η ανάπτυξη κατανεμημένου αλγόριθμου επεξερασίας ερωτημάτων top-k κυριαρχίας για το Spark. Ως σημείο εκκίνησης χρησιμοποιείται ένας υπάρχων σειριακός αλγόριθμος, πάνω στον οποίο γίνονται μετατροπές, ώστε να προκύψει η κατανεμημένη εκδοχή του. Επιπλέον, στον αλγόριθμο που προκύπτει γίνονται βελτιστοποιήσεις για περαιτέρω αύξηση αποδοτικότητας.

Η εργασία δομείται σε κεφάλαια ως εξής:

- Στο Κεφάλαιο 2 γίνεται μια εκτενής παρουσίαση των ερωτημάτων top-k κυριαρχίας. Τα ερωτήματα top-k κυριαρχίας επιτρέπουν στο χρήστη να ορίσει το μέγεθος του αποτελέσματος και δεν απαιτούν κάποια συνάρτηση βαθμολόγησης, γεγονός που τα καθιστά ένα ισχυρό εργαλείο στα χέρια του αναλυτή δεδομένων. Γίνεται επίσης αναφορά στα ερωτήματα top-k και skyline, με τα οποία είναι στενά συνδεδεμένα τα ερωτήματα top-k κυριαρχίας. Αναγνώστες γνώριμοι με αυτού του είδους ερωτήματα προτίμησης μπορούν να παραλείψουν το συγκεκριμένο κεφάλαιο.

- Στο Κεφάλαιο 3 παρουσιάζεται το προγραμματιστικό περιβάλλον Apache Spark, το οποίο χρησιμοποιήθηκε για την υλοποίηση αυτής της εργασίας. Γίνεται μια επισκόπηση των συστατικών μερών του και περιγράφονται κάποιες από τις βασικές προγραμματιστικές έννοιες που είναι διαθέσιμες στο χρήστη, όπως τα resilient distributed datasets (RDDs). Τέλος, επεξηγείται ο τρόπος με τον οποίο το Spark εκτελείται σε έναν cluster, με μία σύντομη ανάλυση της αρχιτεκτονικής εκτέλεσης του. Χρήστες έμπειροι με το συγκεκριμένο περιβάλλον μπορούν να παραλείψουν αυτό το κεφάλαιο.

- Στο Κεφάλαιο 4 γίνεται μια παρουσίαση του αλγορίθμου που προτείνεται σε αυτήν τη διπλωματική. Αρχικά, γίνεται επισκόπιση του σειριακού αλγορίθμου του οποίου

οι βασικές αρχές και τεχνικές χρησιμοποιούνται στον προτεινόμενο κατανεμημένο. Στη συνέχεια, δίνεται έμφαση στην υλοποίηση για Spark που προτείνεται, και στα βήματα από τα οποία αποτελείται. Τέλος, για κάθε ένα βήμα προτείνονται περαιτέρω βελτιστοποιήσεις, για την επίλυση bottlenecks που προκύπτουν κατά την εκτέλεση.

- Στο Κεφάλαιο 5 δίνονται τα αποτελέσματα της εκτέλεσης του προτεινόμενου αλγορίθμου πάνω σε δεδομένα διαφορετικών κατανομών και μεγέθους. Τα πειράματα που πραγματοποιήθηκαν εξετάζουν όλες τις διάφορες παραμέτρους της υλοποίησης, καθώς και τις διαφορετικές εκδοχές της κάθε φάσης της. Για κάθε σειρά πειραμάτων παρατίθονται τα συμπεράσματα που προέκυψαν από τα αποτελέσματα τους.

- Στο Κεφάλαιο 6 παρατίθεται μία σύνοψη της παρούσας εργασίας, ενώ γίνεται λόγος σε μελλοντικές επεκτάσεις που θα μπορούσαν να γίνουν, για αποδοτικότερη επίλυση του προβλήματος που πραγματεύεται η διπλωματική.

Κεφάλαιο 2

Ερωτήματα Top-k Κυριαρχίας

2.1 Εισαγωγή

Αντικείμενο αυτού του κεφαλαίου αποτελούν τα ερωτήματα top-k κυριαρχίας[1], καθώς η επεξεργασία αυτών είναι το αντικείμενο που πραγματεύεται η παρούσα εργασία. Ένα ερώτημα top-k κυριαρχίας επιστρέφει στο χρήστη τα k αντικείμενα ενός dataset με το υψηλότερο σκορ κυριαρχίας. Ως εκ τούτου, τα ερωτήματα top-k κυριαρχίας αποτελούν ένα χρήσιμο εργαλείο για τον αναλυτή δεδομένων, καθώς παρέχουν έναν ευκολονόητο τρόπο εντοπισμού σημαντικών αντικειμένων μέσα σε ένα dataset.

Τα ερωτήματα top-k κυριαρχίας συνδιάζουν τα πλεονεκτήματα των ερωτημάτων top-k και skyline, χωρίς όμως να μοιράζονται τα μειονεκτήματά τους. Συγκεκριμένα, τα ερωτήματα top-k κυριαρχίας επιστρέφουν ένα αποτέλεσμα του οποίου το μέγεθος ορίζει ο χρήστης, ενώ δεν απαιτούν τον ορισμό κάποιας συνάρτησης βαθμολόγησης (ranking function).

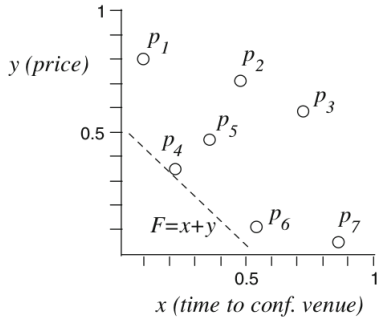
Στη συνέχεια του κεφαλαίου, γίνεται αρχικά μία συνοπτική παρουσίαση των ερωτημάτων top-k και skyline, τα οποία όπως αναφέρθηκε, σχετίζονται άμεσα με τα ερωτήματα top-k κυριαρχίας. Στη συνέχεια, γίνεται μια εκτενής αναφορά στα ερωτήματα top-k κυριαρχίας και ορίζεται η έννοια του σκορ κυριαρχίας. Επιπλέον, προσδιορίζονται τα είδη κυριαρχίας και τα άνω/κάτω φράγματα του σκορ κυριαρχίας, έννοιες που παίζουν καθοριστικό ρόλο στους αλγορίθμους που παρουσιάζονται στο κεφάλαιο 4.

2.2 Ερωτήματα Top-k και Skyline

Σε αυτήν την ενότητα παρουσιάζονται τα ερωτήματα top-k[3, 4] και τα ερωτήματα skyline[2]. Όπως προαναφέρθηκε, τα ερωτήματα top-k κυριαρχίας συνδέονται στενά με τα ερωτήματα top-k και skyline, καθώς στηρίζονται και συνδιάζουν τις βασικές αρχές αυτών. Για αυτόν το λόγο, θεωρούμε πως ο αναγνώστης θα πρέπει να είναι εξοικειωμένος με αυτών των ειδών ερωτήματα προτίμησης, πρωτού προχωρήσουμε στην παρουσίαση των ερωτημάτων top-k κυριαρχίας, με τα οποία ασχολούμαστε σε αυτήν την εργασία.

2.2.1 Ερωτήματα Top-k

Υποθέτουμε ότι έχουμε ένα σύνολο από σημεία $\mathcal{D}$ σε έναν d-διάστατο χώρο $\mathcal{R}^d$. Δεδομένης μιας (συνήθως μονότονης) συνάρτησης βαθμολόγησης $F : \mathcal{R}^d \to \mathcal{R}$, ένα ερώτημα top-k επιστρέφει τα k σημεία για τα οποία η F έχει τη μικρότερη τιμή. Για παράδειγμα, στο σχήμα 2.1 φαίνεται ένα σύνολο απο ξενοδοχεία στο 2-διάστατο χώρο, με τις δύο διαστάσεις να αντιπροσωπεύουν από ένα χαρακτηριστικό των ξενοδοχείων, την



Σχημα 2.1: Σύνολο δεδομένων με 7 ξενοδοχεία [1]

τιμή δωματίου και την απόσταση από ένα συνεδριακό χώρο. Για τη συνάρτηση βαθμολόγησης $F = x + y$, τα top-2 ξενοδοχεία είναι τα p_4 και p_6.

Ένα προφανές πλεονέκτημα που έχουν τα ερωτήματα top-k είναι πως ο χρήστης ελέγχει τον αριθμό των αποτελεσμάτων μέσω της παραμέτρου k. Από την άλλη, δεν είναι πάντα εύκολος για το χρήστη ο ορισμός μιας συνάρτησης βαθμολόγησης. Επιπλέον, τα ερωτήματα top-k δεν είναι κατάλληλα για τον εντοπισμό σημαντικών αντικειμένων σε ένα σύνολο δεδομένων, καθώς διαφορετικές συναρτήσεις βαθμολόγησης μπορεί να οδηγήσουν σε διαφορετικά αποτελέσματα.

2.2.2 Ερωτήματα Skyline

Ένα ερώτημα skyline επιστρέφει όλα τα σημεία τα οποία δεν κυριαρχούνται από κανένα άλλο σημείο. Κάνοντας την υπόθεση πως οι μικρότερες τιμές είναι πιο επιθυμητές από τις μεγαλύτερες για κάθε διάσταση, ένα σημείο p κυριαρχεί κάποιο άλλο σημείο p' ($p \succ p'$) όταν

$$(\exists\ i \in [1,\ d],\ p[i] < p'[i]) \wedge (\forall\ i \in [1,\ d],\ p[i] \leq p'[i])$$

όπου το $p[i]$ αντιστοιχεί στην τιμή της i-οστής συντεταγμένης του p. Στο παράδειγμα με τα ξενοδοχεία, το ερώτημα skyline θα επέστρεφε τα σημεία p_1, p_4, p_6 και p_7.

Ένα βασικό πλεονέκτημα των ερωτημάτων skyline είναι πως δεν απαιτούν τον ορισμό κάποιας συνάρτησης βαθμολόγησης, καθώς τα αποτελέσματά τους εξαρτώνται μόνο από τα εσωτερικά χαρακτηριστικά των δεδομένων. Επιπλέον, η ύπαρξη διαφορετικής κλίμακας τιμών σε κάθε διάσταση δεν αποτελεί πρόβλημα, αφού μόνο η διάταξη των ανά διάσταση προβολών των σημείων είναι αυτή που έχει σημασία.

Από την άλλη πλευρά, το μέγεθος του αποτελέσματος ενός skyline ερωτήματος δεν ορίζεται από το χρήστη, και στη χειρότερη περίπτωση μπορεί να είναι τόσο μεγάλο όσο και το σύνολο δεδομένων στο οποίο εκτελείται το ερώτημα. Κατά συνέπεια, πολλές φορές ο χρήστης δεν μπορεί να αξιοποιήσει το αποτέλεσμα λόγω μεγέθους, καθώς μπορεί να χρειαστεί να εξετάσει πολλά σημεία skyline προκειμένου να εντοπίσει σημεία ενδιαφέροντος. Επιπλέον, υπάρχουν περιπτώσεις που τα ερωτήματα skyline δεν μπορούν να δώσουν καμία σημαντική πληροφορία για το dataset, όπως για παράδειγμα, στα correlated σύνολα δεδομένων, όπου το σημείο skyline είναι μόνο ένα, ή στα τελείως anti-correlated όπου κάθε σημείο του dataset είναι σημείο skyline. Και στις δύο περιπτώσεις δεν μπορεί να προκύψει κάποιο σημαντικό συμπέρασμα σχετικά με το dataset και τα σημαντικά σημεία του.

2.3 Σκορ Κυριαρχίας και Ερωτήματα Top-k Κυριαρχίας

Βάσει της προηγούμενης ενότητας, τα ερωτήματα top-k δεν παρέχουν μία αντικειμενική διάταξη των σημείων ως προς τη σημαντικότητά τους, καθώς τα αποτελέσματά τους επηρεάζονται άμεσα από τη συνάρτηση βαθμολόγησης που χρησιμοποιείται. Από την άλλη, τα ερωτήματα skyline παρέχουν ένα υποσύνολο των σημαντικών σημείων, του οποίου το μέγεθος είναι αυθαίρετο.

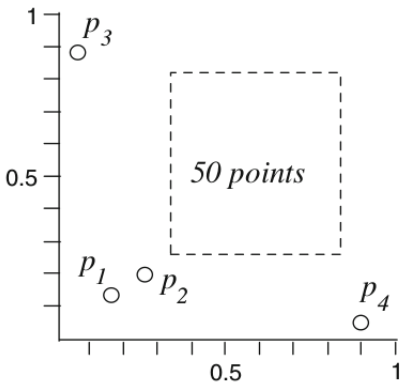
Για τον προσδιορισμό μιας φυσικής διάταξης σημαντικότητας, βάσει κυριαρχίας, ορίστηκε η έννοια του σκορ κυριαρχίας:

$$r(p) = |\{p' \in \mathcal{D} \mid p \succ p'\}|$$

Στην ουσία, το σκορ κυριαρχίας r ενός σημείου p ισούται με τον αριθμό των σημείων p' που κυριαρχεί το p. Επιπλέον, ισχύει η παρακάτω ιδιότητα για το σκορ κυριαρχίας r:

$$\forall p, p' \in \mathcal{D},\ p \succ p' \Rightarrow r(p) > r(p')$$

Βάσει του σκορ κυριαρχίας, είναι δυνατός ο προσδιορισμός μιας φυσικής διάταξης των σημείων ενός συνόλου δεδομένων. Τα ερωτήματα top-k κυριαρχίας επιστρέφουν τα k σημεία του $\mathcal{D}$ με το υψηλότερο σκορ. Για παράδειγμα, το top-2 ερώτημα κυριαρχίας πάνω στα δεδομένα του σχήματος 2.1 επιστρέφει το p_4 ($r(p_4) = 3$) και το p_5 ($r(p_5) = 2$). Αυτό το αποτέλεσμα μπορεί να υποδηλώνει σε έναν αναλυτή δεδομένων ποια είναι τα πιο δημοφιλή ξενοδοχεία στους συμμετέχοντες του συνεδρίου, με κριτήρια την τιμή και την απόσταση από το χώρο. Η δημοτικότητα ενός ξενοδοχείου p βασίζεται στο αριθμό των ξενοδοχείων από τα οποία το p είναι προτιμότερο, για οποιαδήποτε συνάρτηση προτίμησης.



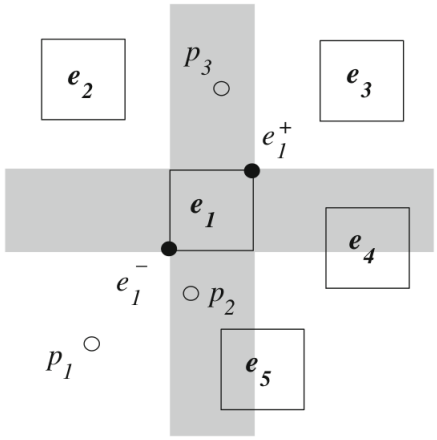
Σχημα 2.2: Σύνολο δεδομένων με 54 ξενοδοχεία [1]

Ως ένα άλλο παράδειγμα στο οποίο φαίνεται πως το σκορ προτίμησης σχετίζεται με τη δημοφιλία, υποθέτουμε πως έχουμε ένα σύνολο δεδομένων με 54 ξενοδοχεία (σχήμα 2.2). Σε αυτήν την περίπτωση, τα top-2 κυρίαρχα σημεία είναι τα p_1 ($r(p_1) = 51$) και p_2 ($r(p_2) = 50$). Παρόλο που το p_2 δεν είναι σημείο skyline, γίνεται ιδιαίτερα σημαντικό όταν το top-1 ξενοδοχείο p_1 γεμίσει. Ο λόγος είναι πως το p_2 είναι εγγυημένα καλύτερο από 50 άλλα ξενοδοχεία, για οποιαδήποτε μονότονη συνάρτηση προτίμησης και αν χρησιμοποιήσουν οι συμμετέχοντες. Αντιθέτως, το skyline σημείο p_3 δεν είναι ικανό να εγγυηθεί κάτι τέτοιο, καθώς στη χειρότερη περίπτωση οι συμμετέχοντες του συνεδρίου μπορεί να ενδιαφέρονται για φθηνά ξενοδοχεία, περίπτωση στην οποία το p_3 δεν είναι

καθόλου καλό. Αντίστοιχες παρατηρήσεις μπορούν να γίνουν και για το p_4.

Το παραπάνω παράδειγμα δείχνει πως τα ερωτήματα top-k κυριαρχίας αποτελούν ένα πολύ ισχυρό εργαλείο υποστήριξης λήψης αποφάσεων, καθώς εντοπίζουν τα σημαντικότερα αντικείμενα ενός συνόλου δεδομένων με έναν ευκολονόητο τρόπο. Στην ουσία τα ερωτήματα top-k κυριαρχίας συνδιάζουν τα πλεονεκτήματα των ερωτημάτων top-k και skyline, χωρίς να μοιράζονται τα μειονεκτήματά τους. Ο αριθμός των αποτελεσμάτων καθορίζεται από το χρήστη, χωρίς να ορίζεται κάποια συνάρτηση βαθμολόγησης. Επιπλέον, δεν απαιτείται κανονικοποίηση των δεδομένων.

2.4 Είδη Κυριαρχίας και Φράγματα Σκορ Κυριαρχίας



Σχημα 2.3: Σχέσεις κυριαρχίας μεταξύ σημείων και κελιών [1]

Σε αυτήν την ενότητα παρουσιάζεται κάποια σημειογραφία και κάποιες έννοιες οι οποίες χρησιμοποιούνται συχνά κατά το Κεφάλαιο 4 και με τις οποίες ο αναγνώστης θα πρέπει να είναι εξοικειωμένος. Για μία οντότητα e, η οποία μπορεί να είναι το κελί ενός πλέγματος που έχει εφαρμοστεί στο χώρο των δεδομένων (ή ένα minimum bounding box με σημεία του dataset), της οποίας η προβολή της i-οστής διάστασης είναι το διάστημα $[e[i]^-, e[i]^+]$, ορίζεται η κάτω γωνία e^- και η πάνω γωνία e^+ ως:

$$e^- = (e[1]^-, e[2]^-, \ldots, e[d]^-) \text{ και } e^- = (e[1]^+, e[2]^+, \ldots, e[d]^+)$$

Τα e^- και e^+ δεν αντιστοιχούν σε πραγματικά σημεία, επιτρέπουν όμως την εύκολη έκφραση σχέσεων κυριαρχίας μεταξύ κελιών και σημείων. Όπως φαίνεται και στο σχήμα 2.3, διακρίνονται τρεις περιπτώσεις σχέσεων κυριαρχίας μεταξύ ενός σημείου και ενός κελιού:

i) *πλήρης κυριαρχία* (π.χ. $p_1 \succ e_1^-$, άρα το p_1 κυριαρχεί όλα τα σημεία εντός του e_1)

ii) *μερική κυριαρχία* (π.χ. το p_2 κυριαρχεί το e_1^+ αλλά όχι το e_1^-, επομένως το p_2 κυριαρχεί κάποια αλλά όχι όλα από τα σημεία του e_1)

iii) *μη κυριαρχία* (π.χ. $p_3 \not\succ e_1^+$, άρα το p_3 δε γίνεται να κυριαρχεί κανένα σημείο εντός του e_1)

Αντίστοιχα, παρόμοιες σχέσεις κυριαρχίας ορίζονται και μεταξύ δύο κελιών:

- $e_1^+ \succ e_3^-$, άρα το e_1 κυριαρχεί πλήρως το e_3.

- $e_1^- \succ e_4^+ \wedge e_1^+ \not\succ e_4^-$, άρα το e_1 κυριαρχεί μερικώς το e_4.
- $e_1^- \not\succ e_2^+$, άρα το e_1 δεν κυριαρχεί καθόλου το e_2.

Για ένα κελί e, για του οποίου τα σημεία δε γνωρίζουμε το σκορ, οι τιμές $r(e^-)$ και $r(e^+)$ αντιστοιχούν σε κάτω και άνω φράγμα, αντίστοιχα, του σκορ κάθε σημείου εντός του e. Αυτά τα φράγματα, έστω $r^l(e)$ και $r^u(e)$, όπως θα δούμε και στο Κεφάλαιο 4, υπολογίζονται εύκολα και επιτρέπουν το κλάδεμα σημείων χωρίς να χρειάζεται να υπολογιστεί το ακριβές σκορ τους.

2.5 Επίλογος και Σύνοψη

Σε αυτό το κεφάλαιο παρουσιάστηκαν τα ερωτήματα top-k κυριαρχίας, τα οποία είναι αντικείμενο μελέτης της παρούσας εργασίας. Αρχικά έγινε μία αναφορά στα ερωτήματα top-k και skyline, κατά την οποία υπογραμμίστηκαν τα πλεονεκτήματα και τα μειονεκτήματά τους. Τα ερωτήματα top-k επιτρέπουν στο χρήστη να ορίσει το μέγεθος του αποτελέσματος, αλλά απαιτούν τον ορισμό συνάρτησης βαθμολόγησης. Από την άλλη πλευρά, τα ερωτήματα skyline δεν χρειάζονται συνάρτηση βαθμολόγησης, όμως το μέγεθος του αποτελέσματος είναι αυθαίρετο.

Τα ερωτήματα top-k κυριαρχίας συνδιάζουν τα πλεονεκτήματα των ερωτημάτων top-k και skyline, χωρίς να μοιράζονται τα μειονεκτήματά τους. Με τον ορισμό του σκορ κυριαρχίας $r(p)$, που είναι ο αριθμός των σημείων p' που κυριαρχεί το p, γίνεται δυνατή η φυσική διάταξη των αντικειμένων ενός συνόλου δεδομένων βάσει της σημαντικότητάς τους. Τα ερωτήματα top-k κυριαρχίας επιστρέφουν τα k σημεία με το υψηλότερο σκορ.

Στο τέλος του κεφαλαίου ορίστηκαν κάποιες έννοιες οι οποίες αξιοποιούνται από τους αλγορίθμους στο κεφάλαιο 4, ωστέ να επιτευχθεί κλάδεμα σημείων χωρίς να υπολογιστεί το σκορ τους.

Κεφάλαιο 3

Περιβάλλον Apache Spark

3.1 Εισαγωγή

Σε αυτό το κεφάλαιο γίνεται μια συνοπτική παρουσίαση του προγραμματιστικού περιβάλλοντος Apache Spark[6], το οποίο χρησιμοποιήθηκε για την πραγματοποίηση της παρούσας εργασίας. Πρόκειται για ένα ανοιχτού κώδικα cluster-computing framework που αναπτύχθηκε αρχικά στο πανεπιστήμιο Berkeley της Καλιφόρνια, αλλά στη συνέχεια παραχωρήθηκε στην Apache Software Foundation, η οποία το συντηρεί μέχρι σήμερα.

Σχημα 3.1: Το λογότυπο του Apache Spark[6]

Το *Apache Spark*[5] είναι μία πλατφόρμα υπολογισμών σε cluster, η οποία σχεδιάστηκε να είναι γρήγορη και γενικού σκοπού. Το πρώτο επιτυγχάνεται με την επέκταση του προγραμματιστικού μοντέλου *MapReduce*[8] της Google. Το Spark υποστηρίζει περισσότερα είδη υπολογισμών, όπως διαδραστικά ερωτήματα και επεξεργασία streams. Το μεγαλύτερο όμως πλεονέκτημα του έναντι στο MapReduce είναι η δυνατότητα υπολογισμών στην κύρια μνήμη, ενώ επιδεικνύει καλύτερες επιδόσεις και σε εφαρμογές οι οποίες εκτελούνται στη δευτερεύουσα. Τα παραπάνω φαίνονται καλύτερα στον πίνακα 3.1, στο οποίο γίνεται σύγκριση του Spark με ένα άλλο framework, το Apache Hadoop[7], το οποίο υλοποιεί το κλασσικό μοντέλο MapReduce.

Πέρα από την ταχύτητα, ένα άλλο πλεονέκτημα που προσφέρει το Apache Spark είναι ότι σχεδιάστηκε για μια πληθώρα διαφορετικών εργασιών, όπως υλοποίηση επαναληπτικών αλγορίθμων ή επεξεργασία streams, οι οποίες προηγουμένως απαιτούσαν ξεχωριστά κατανεμημένα συστήματα. Αυτό καθιστά το Spark έναν εύκολο και οικονομικό τρόπο συνδιασμού διαφορετικών ειδών επεξεργασίας, κάτι που συνήθως απαιτείται σε pipelines ανάλυσης δεδομένων, καθώς εξαφανίζει την ανάγκη συντήρησης ξεχωριστών εργαλείων. Βλέπουμε λοιπόν πως πράγματι πρόκειται για μία πλατφόρμα γενικού σκοπού.

Το Apache Spark είναι ιδιαίτερα προσιτό και εύκολο στη χρήση. Παρέχει στο χρήστη υψηλού επιπέδου APIs σε Scala, Java, Python και R, ενώ διαθέτει μια πλούσια ποικιλία από ενσωματωμένες βιβλιοθήκες. Τέλος, μπορεί να συνεργαστεί με άλλα εργαλεία

	Hadoop	Spark 100TB	Spark 1PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Πινακας 3.1: Apache Spark vs Apache Hadoop[14]

μεγάλων δεδομένων, όπως το σύστημα βάσεων δεδομένων Cassandra[10] και το κατανεμημένο σύστημα αρχείων HDFS[9], το οποίο και χρησιμοποιήθηκε στα πλαίσια αυτής της εργασίας.

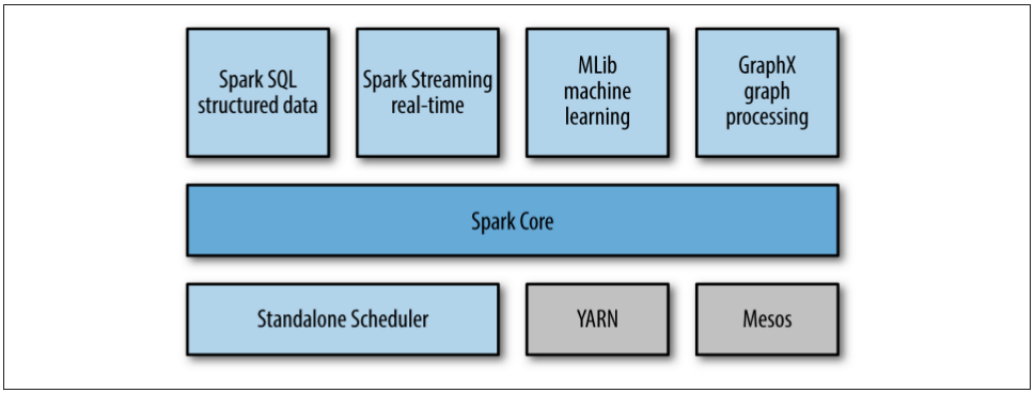
3.2 Συστατικά Μέρη του Apache Spark

Το Apache Spark αποτελείται από διάφορα συστατικά μέρη τα οποία ενσωματώνονται πάνω στον κύριο πυρήνα του. Ο πυρήνας του Spark είναι υπεύθυνος για το χρονοπρογραμματισμό, την κατανομή και την παρακολούθηση εφαρμογών που αποτελούνται από πολλαπλές υπολογιστικές εργασίες πάνω σε έναν cluster. Πάνω στον πυρήνα έρχονται και συνδέονται συστατικά μέρη που πραγματοποιούν εξειδικευμένους υπολογισμούς, όπως ερωτήματα SQL και μηχανική μάθηση.

Η λογική της ενσωμάτωσης νέων συστατικών στοιχείων πάνω στον πυρήνα έχει διάφορα οφέλη. Καταρχάς, βελτιώσεις στα χαμηλότερα επίπεδα επωφελούν τα υψηλότερα. Έτσι, μια βελτιστοποίηση που γίνεται στον πυρήνα του Spark έχει ως αποτέλεσμα οι βιβλιοθήκες της μηχανικής μάθησης να γίνονται αυτόματα πιο γρήγορες. Ένα άλλο πλεονέκτημα είναι πως περιορίζεται η ανάγκη χρήσης πολλαπλών εξειδικευμένων εργαλείων, καθώς το Spark αποτελεί μία all-in-one λύση. Κάτι τέτοιο μειώνει το οικονομικό κόστος σε μια επιχείρηση, καθώς χρειάζεται να συντηρείται μόνο μία πλατφόρμα. Επιπλέον, κάθε φορά που ενσωματώνεται κάποιο νέο συστατικό στοιχείο στον πυρήνα του, οι χρήστες του Spark μπορούν να το δοκιμάσουν αμέσως, γλιτώνοντας το χρόνο που θα χρειαζόταν να στήσουν και να μάθουν κάποιο άλλο λογισμικό.

Το μεγαλύτερο όμως πλεονέκτημα της ενσωμάτωσης πολλαπλών στοιχείων πάνω στον πυρήνα του Spark είναι η ευκολία συνδιασμού διαφορετικών ειδών εργασιών μέσα στην ίδια εφαρμογή. Για παράδειγμα, είναι δυνατόν μέσα σε μία εφαρμογή Spark να γίνεται κατηγοριοποίηση δεδομένων που καταφθάνουν σε πραγματικό χρόνο από live streams, ενώ ταυτόχρονα οι χρήστες πραγματοποιούν ερωτήματα SQL πάλι σε πραγματικό χρόνο.

Στις παρακάτω υποενότητες εξετάζονται τα υπάρχοντα συστατικά μέρη του Apache Spark, τα οποία φαίνονται και στο σχήμα 3.2.



Σχημα 3.2: Τα συστατικά μέρη του Spark [5]

3.2.1 Spark Core

Το Spark Core είναι υπεύθυνο για όλες τις βασικές λειτουργίες του Spark, όπως χρονοπρογραμματισμός εργασιών, διαχείριση μνήμης, αποκατάσταση σφαλμάτων και άλλα. Το Spark Core επίσης περιέχει το API που ορίζει τα resilient distributed datasets (RDDs). Τα RDDs αντιπροσωπεύουν μια συλλογή από αντικείμενα κατανεμημένα στους κόμβους ενός cluster, τα οποία μπορούν να χειριστούν παράλληλα μέσω APIs που επίσης παρέχει το Spark Core.

3.2.2 Spark SQL

Το Spark SQL είναι το πακέτο του Spark για επεξεργασία δομημένων δεδομένων. Επιτρέπει την υποβολή ερωτημάτων μέσω SQL, και HQL – την παραλλαγή της SQL του Apache Hive[11] – και υποστηρίζει πολλές πηγές δεδομένων, όπως πίνακες Hive, Parquet και JSON. Πέρα από την παροχή διεπαφής SQL στο Spark, το Spark SQL επιτρέπει στους προγραμματιστές να αναμιγνύουν ερωτήματα SQL με τις ενέργειες που υποστηρίζουν τα RDDs, σε μία ενιαία εφαρμογή, συνδιάζοντας έτσι την SQL με πιο περίπλοκες εργασίες.

Το Spark SQL προστέθηκε στο Spark στην έκδοση 1.0. Αντικατέστησε το Shark, ένα έργο που τροποποιούσε το Apache Hive ώστε να τρέχει σε Spark. Η αντικατάσταση έγινε για καλύτερη ενσωμάτωση στο Spark Core και μεγαλύτερη συμβατότητα με τα APIs του.

3.2.3 Spark Streaming

Το Spark Streaming είναι ένα πάκετο του Spark που επιτρέπει την επεξεργασία live streams δεδομένων. Παραδείγματα streams δεδομένων είναι τα logfiles που παράγονται από έναν server ή οι δημοσιεύσεις που κοινοποιούν οι χρήστες ενός κοινωνικού δικτύου. Το Spark Streaming παρέχει ένα API για το χειρισμό streams δεδομένων, αντίστοιχο του RDD API του Spark Core, κάθιστώντας εύκολο στους προγραμματιστές να μεταπηδούν από τη διαχείριση δεδομένων που βρίσκονται στη μνήμη, στη διαχείριση δεδομένων που καταφθάνουν σε πραγματικό χρόνο. Τέλος, το Spark Streaming σχεδιάστηκε να προσφέρει την ίδια ανοχή σε σφάλματα, ρυθμαπόδοση και κλιμακωσιμότητα με το Spark Core.

3.2.4 MLlib

Το Spark έρχεται με μια βιβλιοθήκη που περιέχει συνηθισμένες διαδικασίες μηχανικής μάθησης, την MLlib. Η MLlib παρέχει πολλών διαφορετικών ειδών αλγορίθμους μηχανικής μάθησης, ανάμεσα στους οποίους αλγορίθμους κατηγοριοποίησης, ομαδοποίησης και παλινδρόμησης, ενώ υποστηρίζει λειτουργίες όπως η αξιολόγηση μοντέλων. Όλα τα παραπάνω είναι σχεδιασμένα έτσι ώστε να κλιμακώνονται σε έναν cluster.

3.2.5 GraphX

Η GraphX είναι μία βιβλιοθήκη για τη διαχείριση και επεξεργασία γράφων, και την πραγματοποίηση παράλληλων υπολογισμών πάνω σε αυτούς. Όπως και τα Spark Streaming και Spark SQL, έτσι και η GraphX επεκτείνει το RDD API του Spark Core, παρέχοντας στο χρήστη διάφορους τελεστές για τη διαχείριση γράφων, ενώ προσφέρει και μία βιβλιοθήκη με συνηθισμένους αλγορίθμους (π.χ PageRank).

3.2.6 Διαχειριστές Cluster

Το Spark σχεδιάστηκε να κλιμακώνεται αποδοτικά από έναν σε χιλιάδες κόμβους. Για να επιτευχθεί αυτό, προσφέροντας παράλληλα ευελιξία, το Spark μπορεί να συνεργάζεται με μία πληθώρα διαχειριστών clusters όπως οι Hadoop YARN[12] και Apache Mesos[13], ενώ περιλαμβάνει και το ίδιο έναν απλό διαχειριστή, τον Standalone Scheduler.

3.3 Βασικές Έννοιες του Apache Spark

Σε αυτήν την ενότητα παρουσιάζονται κάποιες από τις βασικές προγραμματιστικές έννοιες του Apache Spark, οι οποίες χρησιμοποιήθηκαν στην παρούσα εργασία. Αρχικά γίνεται μία εκτενής περιγραφή της δομής των RDDs, ενώ στη συνέχεια εξετάζονται τα διάφορα είδη μοιραζόμενων μεταβλητών που υποστηρίζει το Spark.

3.3.1 Resilient Distributed Datasets

Ένα RDD πρόκειται για μία αμετάβλητη κατανεμημένη συλλογή από αντικείμενα. Κάθε RDD είναι χωρισμένο σε πολλαπλές κατατμήσεις (partitions), οι οποίες μπορεί να βρίσκονται και να επεξεργάζονται σε διαφορετικούς κόμβους του ίδιου cluster. Τα RDDs μπορεί να περιέχουν οποιουδήποτε είδους Scala, Java ή Python αντικείμενα, ακόμα και κλάσης ορισμένης από το χρήστη.

Οι χρήστες μπορούν να δημιουργήσουν RDDs με δύο τρόπους, είτε φορτώνοντας κάποιο εξωτερικό αρχείο, είτε διαμοιράζοντας μία συλλογή αντικειμένων από τον driver (βλέπε 3.4.1). Όταν δημιουργηθεί ένα RDD υπάρχουν δύο είδη ενεργειών που μπορούν να πραγματοποιηθούν με αυτό: οι μετασχηματισμοί (transformations) και οι πράξεις (actions). Οι μετασχηματισμοί κατασκευάζουν ένα νέο RDD βασισμένο στο αρχικό, ενώ οι πράξεις υπολογίζουν ένα αποτέλεσμα και είτε το επιστρέφουν στο πρόγραμμα driver, είτε το αποθηκεύουν σε κάποιο εξωτερικό σύστημα αποθήκευσης (π.χ. HDFS).

Η διαφοροποίηση μεταξύ μετασχηματισμών και πράξεων έχει να κάνει με τον τρόπο που το Spark υπολογίζει τα RDDs. Το RDD μπορεί να οριστεί οποιαδήποτε στιγμή, το Spark όμως θα το υπολογίσει *τεμπέλικα*, όταν θα το χρειαστεί σε κάποια πράξη. Κάτι τέτοιο μπορεί να φαίνεται ασυνήθιστο, όμως όπως θα δούμε στη συνέχεια είναι μια πολύ λογική τακτική, δεδομένου ότι έχουμε να κάνουμε με μεγάλα δεδομένα.

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Πινακας 3.2: Επιλογές διατήρησης RDD [5]

Τα RDDs υπολογίζονται εκ νέου κάθε φορά που εκτελείται μία πράξη με αυτά. Σε περίπτωση πολλαπλών χρήσεων του ίδιου RDD είναι δυνατή η διατήρηση αυτού μετά την πρώτη του χρήση, μέσω της εντολής RDD.persist(). Η διατήρηση μπορεί να γίνει σε διάφορα σημεία, ανάλογα με το όρισμα που θα δωθεί (οι διαθέσιμες επιλογές φαίνονται στον πίνακα 3.2). Το ότι η διατήρηση του RDD δεν είναι η προεπιλεγμένη συμπεριφορά του Spark μπορεί να φανεί και πάλι κάτι ασυνήθιστο, είναι όμως λογικό για δεδομένα μεγάλου όγκου: αν δεν υπάρχει πρόθεση επαναχρησιμοποίησης του RDD δεν υπάρχει λόγος να ξοδευτεί αποθηκευτικός χώρος για αυτό, όταν το αποτέλεσμα μπορεί να υπολογιστεί με ένα πέρασμα των δεδομένων εκείνη τη στιγμή.

Παρακάτω γίνεται μία πιο αναλυτική αναφορά στους μετασχηματισμούς και τις πράξεις που μπορούν να γίνουν με τα RDDs, ενώ δίνεται και ο λόγος που το Spark χρησιμοποιεί την τακτική του λεγόμενου lazy evaluation.

Μετασχηματισμοί (Transformations)

Οι μετασχηματισμοί είναι ενέργειες που πραγματοποιούνται σε ένα RDD και επιστρέφουν ένα νέο. Όπως αναφέρθηκε προηγουμένως, τα μετασχηματισμένα RDDs υπολογίζονται όταν χρησιμοποιούνται σε κάποια πράξη και όχι όταν ορίζονται. Η πλειοψηφία των μετασχηματισμών επιδρά σε κάθε εγγραφή του RDD ξεχωριστά, αλλά αυτό δεν ισχύει για όλους τους μετασχηματισμούς.

Ας υποθέσουμε ότι έχουμε ένα RDD, έστω inputRDD, το οποίο ως εγγραφές έχει τα μηνύματα ενός logfile, και στόχος μας είναι να βρούμε τα μηνύματα που περιέχουν τη λέξη error. Κάτι τέτοιο μπορεί εύκολα να επιτευχθεί με τη χρήση του μετασχηματισμού filter(). Η εφαρμογή αυτού του μετασχηματισμού δε θα αλλάξει το αρχικό RDD αλλά θα επιστρέψει ένα καινούριο, έστω errorRDD. Θυμίζουμε ότι το errorRDD δεν έχει δημιουργηθεί ακόμα, καθώς δεν πραγματοποιήθηκε κάποια πράξη που να το χρησιμοποιεί.

Καθώς παράγονται νέα RDDs από την εφαρμογή μετασχηματισμών πάνω σε άλλα, το Spark καταγράφει τις μεταξύ τους εξαρτήσεις. Στο παραπάνω παράδειγμα, το errorRDD είναι εξαρτημένο από το inputRDD. Κάθε φορά που χρειάζεται να υπολογιστεί το errorRDD εξαιτίας κάποιας πράξης που το χρησιμοποιεί, χρειάζεται να υπολογιστεί πρώτα το inputRDD, εφόσον αυτό δε διατηρείται κάπου.

Μετασχηματισμός	Αποτέλεσμα
map(func)	Επιστρέφει ένα νέο RDD, το οποίο προέκυψε από την εφαρμογή της συνάρτησης func σε κάθε μία από τις εγγραφές του αρχικού.
filter(func)	Επιστρέφει ένα νέο RDD, το οποίο αποτελείται από τις εγγραφές του αρχικού για τις οποίες η func επιστρέφει true.
flatMap(func)	Παρόμοια με τη map, με τη διαφορά ότι κάθε εγγραφή του αρχικού RDD μπορεί να χαρτογραφηθεί σε 0 ή παραπάνω νέες εγγραφές (δηλ. η func επιστρέφει λίστα).
mapPartitions(func)	Παρόμοια με τη map, με τη διαφορά ότι η func εφαρμόζεται ανά κατάτμηση αντί ανά εγγραφή.
distinct()	Επιστρέφει ένα νέο σύνολο δεδομένων το οποίο περιέχει τις εγγραφές του αρχικού χωρίς τα διπλότυπα.
reduceByKey(func)	Όταν εφαρμόζεται σε δεδομένα ζευγαριών (key, value), οι τιμές για κάθε κλειδί συναθροίζονται όπως ορίζει η συνάρτηση func.

Πινακας 3.3: Μετασχηματισμοί RDDs

Στον πίνακα 3.3 φαίνονται μερικοί από τους πιο κοινούς μετασχηματισμούς, καθώς και οι μετασχηματισμοί που χρησιμοποιήθηκαν στην παρούσα εργασία.

Πράξεις (Actions)

Παραπάνω είδαμε πως δημιουργούνται RDDs από άλλα RDDs με τη χρήση μετασχηματισμών. Οι πράξεις είναι το δεύτερο είδος ενεργειών RDD με τις οποίες παίρνουμε κάποιο χρήσιμο αποτέλεσμα από τα υπάρχοντα δεδομένα. Το αποτέλεσμα αυτό είτε επιστρέφεται στο πρόγραμμα driver (βλέπε 3.4.1), είτε αποθηκεύεται σε κάποιο εξωτερικό σύστημα αποθήκευσης. Οι πράξεις επιβάλουν την εφαρμογή των μετασχηματισμών πάνω στο RDD που χρησιμοποιούν, καθώς πρέπει να επιστρέψουν κάποιο αποτέλεσμα τη στιγμή που καλούνται.

Συνεχίζοντας με το προηγούμενο παράδειγμα, έστω ότι θέλουμε να μάθουμε πόσα μηνύματα είχαν τη λέξη error και ποια είναι αυτά. Το πρώτο μπορεί εύκολα να επιτευχθεί εφαρμόζοντας την πράξη count() πάνω στο errorRDD, καθώς αυτή επιστρέφει τον αριθμό των εγγραφών του RDD στο οποίο εφαρμόζεται. Όσο για το δεύτερο, αρκεί να εφαρμοστεί η πράξη collect() πάνω στο errorRDD, η οποία ανακτά τα περιεχόμενά του στον driver. Σε περίπτωση που τα περιεχόμενα του RDD δε χωράνε στη μνήμη, υπάρχουν πράξεις όπως η saveAsTextFile(), οι οποίες αποθηκεύουν τα περιεχόμενα του RDD σε κάποιο κατανεμημένο σύστημα αποθήκευσης, όπως το HDFS.

Είναι σημαντικό να σημειωθεί πως στο παραπάνω παράδειγμα το errorRDD (και κατ' επέκταση το inputRDD) δημιουργήθηκε από την αρχή δύο φορές: μία για την εκτέλεση της count και μία για την εκτέλεση της collect. Σε αυτή την περίπτωση θα ήταν επιθυμητή η χρήση της persist στο errorRDD. Αυτό θα είχε ως αποτέλεσμα το errorRDD να δημιουργηθεί εκ νέου μόνο κατά την εκτέλεση της count και να διατηρηθεί για την

Πράξη	Αποτέλεσμα
collect()	Επιστρέφει όλες τις εγγραφές του RDD ως πίνακα στον driver.
count()	Επιστρέφει τον αριθμό εγγραφών του RDD.
foreach(func)	Εφαρμόζει τη συνάρτηση func σε κάθε μία εγγραφή του RDD.
saveAsTextFile(path)	Αποθηκεύει τις εγγραφές του RDD στο path ως αρχείο κειμένου.
reduce(func)	Συναθροίζει τις εγγραφές του RDD χρησιμοποιώντας τη συνάρτηση func.

ΠΙΝΑΚΑΣ 3.4: Πράξεις RDDs

εκτέλεση της collect και οποιωνδήποτε άλλων πιθανών πράξεων.

Στον πίνακα 3.4 φαίνονται μερικές από τις πιο κοινές πράξεις, καθώς και οι πράξεις που χρησιμοποιήθηκαν στην παρούσα εργασία.

Lazy Evaluation

Όπως προαναφέρθηκε, οι μετασχηματισμοί των RDDs πραγματοποιούνται τεμπέλικα, που σημαίνει πως το Spark δε θα τους πραγματοποιήσει αν δε κληθεί μία πράξη που να χρησιμοποιεί τα συγκεκριμένα RDDs. Ο λόγος που γίνεται αυτό από το Spark είναι για να αποφευχθούν άσκοπα περάσματα των δεδομένων, κάτι που όταν έχουμε να κάνουμε με μεγάλα δεδομένα είναι ιδιαίτερα χρονοβόρο. Ας υποθέσουμε πως θέλουμε να:

i) φορτώσουμε ένα εξωτερικό αρχείο ως RDD.

ii) εφαρμόσουμε έναν μετασχηματισμό filter πάνω σε αυτό.

iii) εφαρμόσουμε ένα μετασχηματισμό map στις εγγραφές που πέρασαν το filter.

iv) συλλέξουμε τα τελικά αποτελέσματα στον driver.

Παραπάνω έχουμε τρεις μετασχηματισμούς και μία πράξη. Άμα το Spark εκτελούσε κάθε μετασχηματισμό τη στιγμή που καλούταν, τότε θα είχαμε τρία περάσματα των δεδομένων. Περιμένωντας όμως να κληθεί μία πράξη, το Spark μπορεί να πραγματοποιήσει βελτιστοποιήσεις στο πλάνο εκτέλεσης, έτσι ώστε οι μετασχηματισμοί να πραγματοποιηθούν σε ένα μόνο πέρασμα, κερδίζοντας έτσι πολύτιμο χρόνο.

Η τακτική του lazy evaluation δίνει το πλεονέκτημα στο Spark να μπορεί να ομαδοποιεί ενέργειες ώστε να μειώνει τα συνολικά περάσματα πάνω στα δεδομένα. Σε συστήματα όπως το Hadoop οι προγραμματιστές έπρεπε να σχεδιάσουν προσεκτικά πως να ομαδοποιήσουν οι ίδιοι διάφορες ενέργειες, ώστε να περιοριστούν οι φάσεις MapReduce. Με το Spark και το lazy evaluation οι χρήστες μπορούν να επιτύχουν το ίδιο αποτέλεσμα οργανώνοντας το προγραμμά τους σε μικρότερες και πιο διαχειρίσιμες ενέργειες.

3.3.2 Μοιραζόμενες Μεταβλητές

Κατά την εκτέλεση μία συνάρτησης που δώθηκε ως όρισμα σε μία ενέργεια του Spark σε έναν executor (βλέπε 3.4.2), ο executor αυτός δουλεύει με αντίγραφα των μεταβλητών που χρησιμοποιεί η εν λόγω συνάρτηση. Αυτές οι μεταβλητές αντιγράφονται σε κάθε

κόμβο, αλλά οποιαδήποτε ενημέρωσή τους δεν επιστρέφει πίσω στον driver. Για την αντιμετώπιση αυτού του περιορισμού το Spark παρέχει δύο είδών μοιραζόμενες μεταβλητές, τις μεταβλητές broadcast και τους accumulators.

Μεταβλητές Broadcast

Οι μεταβλητές Broadcast επιτρέπουν σε μία εφαρμογή να στείλει αποδοτικά μία μεγάλου μεγέθους τιμή, που προορίζεται μόνο για ανάγνωση, σε όλους τους κόμβους του cluster, ώστε να χρησιμοποιηθεί από μία ή περισσότερες ενέργειες του Spark. Κάτι τέτοιο είναι ιδιαίτερα χρήσιμο, και όπως θα δούμε χρησιμοποιείται και από τον αλγόριθμο που προτείνεται σε αυτήν την εργασία.

Το Spark όπως αναφέρθηκε, στέλνει αντίγραφα των μεταβλητών που χρησιμοποιούνται από κάποια συνάρτηση σε όλους τους κόμβους. Αυτή η συμπεριφορά, αν και βολική, είναι μη αποδοτική, καθώς κάποια μεταβλητή μπορεί να χρησιμοποιείται από πολλές εργασίες, με αποτέλεσμα να αποστέλεται κάθε φορά εκ νέου. Επιπλέον, δεν έχουν γίνει οι κατάλληλες βελτιστοποιήσεις για την αποστολή μεγάλου μεγέθους μεταβλητών, καθώς ο μηχανισμός εκκίνησης εργασιών προορίζεται για μικρού μεγέθους εργασίες.

Οι μεταβλητές Broadcast δίνουν λύση και στα δύο παραπάνω προβλήματα, καθώς μετά τη μετάδοση των τιμών στους κόμβους, αυτές διατηρούνται για πολλαπλές χρήσεις, ενώ η μετάδοση πραγματοποιείται βάσει αποδοτικού αλγορίθμου για αυτό το σκοπό.

Accumulators

Οι Accumulators είναι μοιραζόμενες μεταβλητές οι οποίες υποστηρίζουν μόνο πρόσθεση σε αυτές. Συνήθως χρησιμοποιούνται για την υλοποίηση μετρητών ή για τον υπολογισμό αθροισμάτων. Η τιμή ενός accumulator είναι προσβάσιμη μόνο από το driver, καθώς οι κόμβοι μπορούν μόνο να την ενημερώσουν.

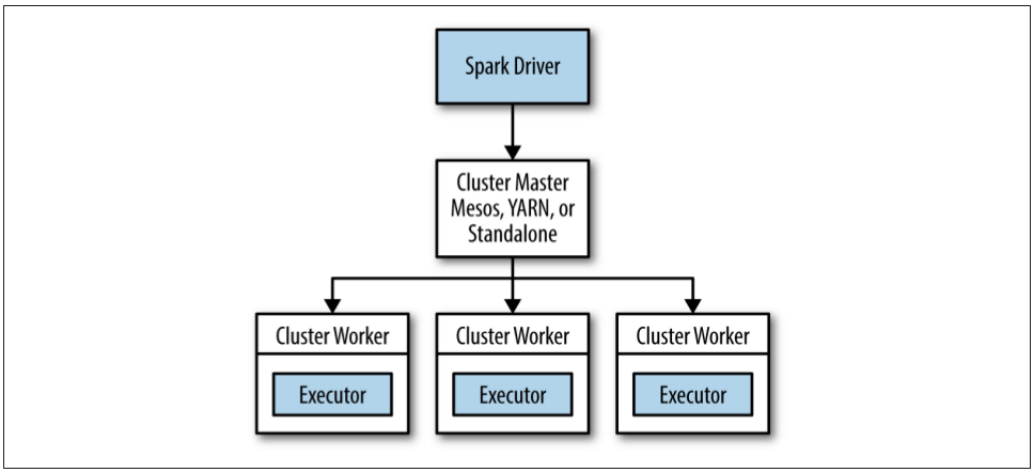
Το Spark υποστηρίζει accumulators με τιμές αριθμητικού τύπου. Παρόλαυτα, παρέχει τη δυνατότητα στους χρήστες να προσθέσουν υποστήριξη και άλλων τύπων, ακόμα και ορισμένων από τους ίδιους.

3.4 Αρχιτεκτονική Εκτέλεσης Apache Spark

Όταν εκτελείται κατανεμημένα, το Apache Spark χρησιμοποιεί μία αρχιτεκτονική αφέντη/σκλάβου, με έναν κεντρικό συντονιστή και πολλούς κατανεμημένους εργάτες. Ο κεντρικός συντονιστής ονομάζεται driver. Ο driver επικοινωνεί με ένα δυνητικά μεγάλο αριθμό από εργάτες, οι οποίοι ονομάζονται executors. Ο driver εκτελείται μέσα στη δικιά του Java διεργασία, ενώ κάθε executor αποτελεί από μόνος του μία ανεξάρτητη διεργασία Java. Ο driver μαζί με τους executors του αποτελούν μία εφαρμογή Spark.

Μία εφαρμογή Spark εκκινείται σε ένα σύνολο από υπολογιστικές μονάδες με τη χρήση μιας εξωτερικής υπηρεσίας που ονομάζεται *διαχειριστής cluster*. Όπως αναφέρθηκε σε προηγούμενη ενότητα, το Spark διαθέτει το δικό του ενσωματωμένο διαχειριστή, τον Standalone, ενώ συνεργάζεται και με δύο δημοφιλείς διαχειριστές ανοιχτού κώδικα, τον Hadoop YARN και τον Apache Mesos.

Στη συνέχεια της ενότητας δίνονται περαιτέρω πληροφορίες για τη λειτουργία του driver, των executors και του διαχειριστή cluster, ενώ παρουσιάζεται σε απλά βήματα



Σχημα 3.3: Τα μέρη μιας κατανεμημένης εφαρμογής Spark [5]

ο κύκλος ζωής μιας τυπικής εφαρμογής Spark. Στο σχήμα 3.3 φαίνονται τα μέρη μιας κατανεμημένης εφαρμογής Spark.

3.4.1 Driver

Ο driver είναι η διεργασία στην οποία εκτελείται η μέθοδος main() μιας εφαρμογής Spark. Πρόκειται για τη διεργασία που εκτελεί τον κώδικα του χρήστη, ο οποίος δημιουργεί το SparkContext, δημιουργεί RDDs, και πραγματοποιεί μετασχηματισμούς και πράξεις με αυτά. Όταν ο driver τερματίσει, η εφαρμογή Spark έχει τελειώσει.

Κατά την εκτέλεσή του, ο driver είναι υπεύθυνος για δύο πράγματα, τη μετατροπή του κώδικα του χρήστη σε εργασίες και τη δρομολόγηση των εργασιών αυτών στους executors. Ξεκινώντας με το πρώτο, ο driver είναι υπεύθυνος να μετατρέψει ένα πρόγραμμα του χρήστη σε ένα σύνολο μονάδων εκτέλεσης που ονομάζονται εργασίες. Κάθε πρόγραμμα Spark ακολουθεί την ίδια δομή: δημιουργεί RDDs από κάποια είσοδο, παράγει νέα με τη χρήση μετασχηματισμών και εκτελεί πράξεις για τη συλλογή ή αποθήκευση δεδομένων. Αυτή η διαδικασία αναπαριστάται από το Spark με ένα λογικό κατευθυνόμενο ακυκλικό γράφο, ο οποίος κατά την εκτέλεση του driver μετατρέπεται σε ένα φυσικό σχέδιο εκτέλεσης. Το Spark πραγματοποιεί διάφορες βελτιστοποιήσεις, όπως τη διασωλήνωση διαδοχικών μετασχηματισμών με σκοπό την ένωση τους, και μετατρέπει το γράφο εκτέλεσης σε ένα σύνολο από βήματα. Κάθε βήμα αποτελείται από πολλαπλές εργασίες. Οι εργασίες είναι η μικρότερη μονάδα εκτέλεσης στο Spark και μια εφαρμογή μπορεί να αποτελείται από χιλιάδες τέτοιες.

Όσον αφορά τη δρομολόγηση των εργασιών, ο driver καλείται να συντονίσει τη δρομολόγηση καθεμίας εργασίας στους executors. Όταν οι executors εκκινούνται, καταγράφονται στον driver, ώστε αυτός να έχει πλήρη εικόνα της κατάστασής τους κατά την εκτέλεση της εφαρμογής. Κάθε executor αποτελεί μία διεργασία ικανή να εκτελεί εργασίες και να αποθηκεύει δεδομένα RDD.

Ο driver ελέγχει το τρέχον σύνολο των executors και προσπαθεί να δρομολογήσει κάθε εργασία στον καταλληλότερο, βάσει της τοποθεσίας των δεδομένων. Όταν οι εργασίες εκτελούνται μπορεί να αφήσουν δεδομένα στην cache του αντίστοιχου executor. Ο driver το λαμβάνει υπόψιν αυτό και φροντίζει να στείλει στο συγκεκριμένο executor εργασίες που αξιοποιούν αυτά τα δεδομένα.

3.4.2 Executors

Οι executors του Spark είναι διεργασίες υπεύθυνες για την εκτέλεση μεμονομένων εργασιών σε μία εφαρμογή Spark. Εκκινούνται στο ξεκίνημα της εφαρμογής και συνήθως τρέχουν καθόλη τη διάρκεια της, αν και η εφαρμογή συνεχίζει να εκτελείται και σε περίπτωση αποτυχίας τους. Οι executors έχουν δύο ρόλους. Πρώτον, εκτελούν τις εργασίες που αποτελούν την εφαρμογή και επιστρέφουν τα αποτελέσματα στο driver. Δεύτερον, παρέχουν εντός μνήμης αποθήκευση των RDDs τα οποία η εφαρμογή του χρήστη ορίζει πως πρέπει να διατηρηθούν. Αυτό επιτυγχάνεται μέσω της υπηρεσίας Block Manager που υπάρχει μέσα σε κάθε executor.

3.4.3 Διαχειριστής Cluster

Ο ρόλος του διαχειριστή cluster είναι να εκκινεί τους executors, και σε ορισμένες περιπτώσεις τον ίδιο το driver. Ο διαχειριστής cluster είναι αποσπώμενο στοιχείο του Spark, κάτι που κάνει πολύ εύκολη την εκτέλεση του Spark πάνω σε εξωτερικούς διαχειριστές, όπως οι YARN και Mesos, πέρα από τον ενσωματωμένο Standalone Scheduler που διαθέτει.

3.4.4 Εκκίνηση Προγράματος

Το Spark παρέχει ένα σενάριο ενεργειών για την υποβολή προγραμμάτων σε αυτό με όνομα spark-submit. Μέσω διάφορων επιλογών που διαθέτει, το spark-submit μπορεί να συνδεθεί με διάφορους διαχειριστές cluster και να ελέγχει πόσοι πόροι θα δωθούν στην εφαρμογή.

3.4.5 Κύκλος Ζωής Εφαρμογής Spark

Παρακάτω φαίνονται τα βήματα που εκτελούνται κάτα τη διάρκεια μιας εφαρμογής Spark σε έναν cluster:

1. Ο χρήστης υποβάλει την εφαρμογή του στο Spark χρησιμοποιώντας το spark-submit.
2. Το spark-submit εκκινεί τον driver και καλεί τη μέθοδο main() που έχει ορίσει ο χρήστης.
3. Ο driver επικοινωνεί με το διαχειριστή cluster και ζητάει πόρους για να εκκινήσει τους executors.
4. Ο διαχειριστής cluster εκκινεί τους executors εκ μέρους του driver.
5. Ο driver εκτελεί την εφαρφμογή του χρήστη, και ανάλογα με τους μετασχηματισμούς και πράξεις στα RDD, στέλνει δουλειά στους executors με τη μορφη εργασιών.
6. Οι εργασίες εκτελούνται στους executors για τον υπολογισμό και την αποθήκευση αποτελεσμάτων.
7. Εάν η μέθοδος main() ολοκληρωθεί ή καλέσει τη μέθοδο SparkContext.stop(), οι executors τερματίζουν και απελευθερώνονται οι πόροι.

3.5 Επίλογος και Σύνοψη

Σε αυτό το κεφάλαιο έγινε μια παρουσίαση του προγραμματιστικού περιβάλλοντος Apache Spark, το οποίο χρησιμοποιήθηκε στα πλαίσια αυτής της εργασίας. Πρόκειται για μία πλατφόρμα υπολογισμών σε cluster, η οποία σχεδιάστηκε να είναι γρήγορη και γενικού σκοπού.

Το Apache Spark έρχεται με ενσωματωμένες βιβλιοθήκες που πραγματοποιούν εξειδικευμένους υπολογισμούς, όπως λειτουργίες μηχανικής μάθησης και ερωτήματα SQL. Ένα από τα μεγαλύτερα πλεονεκτήματα του Spark είναι η δυνατότητα συνδιασμού των λειτουργιών αυτών μέσα στο ίδιο πρόγραμμα, κάτι που συνήθως απαιτείται σε pipelines ανάλυσης δεδομένων.

Η διαχείρηση δεδομένων στο Spark γίνεται με τα resilient distributed datasets (RDDs). Ένα RDD είναι μία αμετάβλητη κατανεμημένη συλλογή από αντικείμενα, και μέσω των μετασχηματισμών και πράξεων που είναι διαθέσιμες, ο χρήστης μπορεί να τα επεξεργαστεί και να πάρει κάποιο αποτέλεσμα.

Το Spark κατά την εκτέλεση του σε cluster χρησιμοποιεί μία αρχιτεκτονική αφέντη-/σκλάβου. Υπάρχει ένας κεντρικός συντονιστής που ονομάζεται driver, και ένας δυνητικά μεγάλος αριθμός από εργάτες που ονομάζονται executors. Μια εφαρμογή Spark εκκινείται από το διαχειριστή cluster, ο οποίος εξασφαλίζει πόρους εκ μέρους του driver και εκκινεί τους executors.

Κεφάλαιο 4

Αλγόριθμοι Επεξεργασίας Ερωτημάτων Top-k Κυριαρχίας

4.1 Εισαγωγή

Σκοπός της παρούσας εργασίας είναι η ανάπτυξη αλγορίθμου για την αποδοτική εκτέλεση ερωτημάτων top-k κυριαρχίας πάνω σε δεδομένα μεγάλου όγκου για τα οποία δεν υπάρχει κάποιος κατάλογος. Αυτό γίνεται εφικτό με τη χρήση της πλατφόρμας Apache Spark που παρουσιάστηκε στο Κεφάλαιο 3, η οποία δίνει τη δυνατότητα πραγματοποίησης παράλληλων υπολογισμών σε διαφορετικούς κόμβους ενός cluster. Ο προτεινόμενος αλγόριθμος λοιπόν είναι ένας κατανεμημένος αλγόριθμος.

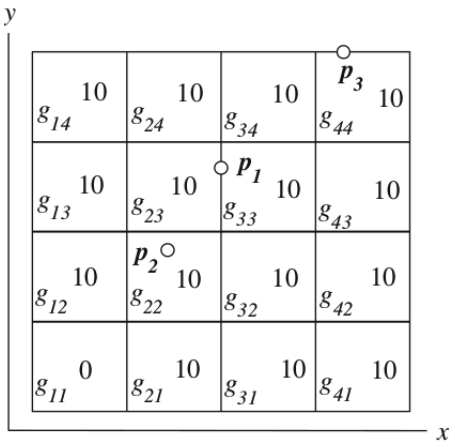
Όπως αναφέρθηκε στο Κεφάλαιο 2, ένα ερώτημα top-k κυριαρχίας επιστρέφει τα k αντικείμενα ενός συνόλου δεδομένων με το μεγαλύτερο σκορ κυριαρχίας. Ως σκορ κυριαρχίας ενός αντικειμένου ορίζεται ο αριθμός των αντικειμένων που κυριαρχούνται από αυτό. Είναι προφανές πως ακόμα και με τον παραλληλισμό που προσφέρει το Spark, ο υπολογισμός του σκορ κυριαρχίας για καθένα από τα αντικείμενα ενός μεγάλου συνόλου δεδομένων είναι μία ιδιαίτερα χρονοβόρα διαδικασία.

Ο αλγόριθμος που προτείνεται σε αυτήν την εργασία βασίζεται σε έναν υπάρχων σειριακό. Ο σειριακός αλγόριθμος εφαρμόζει κάποιες έξυπνες τεχνικές κλαδέματος με τη χρήση πλέγματος, μειώνοντας σημαντικά των αριθμό των αντικειμένων για τα οποία πρέπει να υπολογιστεί το σκορ κυριαρχίας, και κατ΄ επέκταση το υπολογιστικό κόστος. Συνδιάζοντας αυτές τις τεχνικές με τον παραλληλισμό που προσφέρει το Spark, ο προτεινόμενος αλγόριθμος αποτελεί μία ιδιαίτερα αποδοτική υλοποίηση για την εκτέλεση ερωτημάτων top-k κυριαρχίας πάνω σε μεγάλα δεδομένα.

Στις επόμενες ενότητες αρχικά γίνεται μία αναφορά στο σειριακό αλγόριθμο που αποτέλεσε τη βάση του προτεινόμενου κατανεμημένου. Στη συνέχεια παρουσιάζεται ο αλγόριθμος που αναπτύχθηκε, μαζί με κάποιες βελτιώσεις που προστέθηκαν για την επίλυση bottlenecks. Τέλος, γίνεται μία σύνοψη του κεφαλαίου.

4.2 Σειριακός Αλγόριθμος

Ο σειριακός αλγόριθμος[1] κάνει την υπόθεση πως τα δεδομένα βρίσκονται αποθηκευμένα με τυχαία σειρά σε ένα αρχείο στο δίσκο, έστω $\mathcal{D}$. Στόχος του είναι ο υπολογισμός των top-k κυρίαρχων σημείων χρησιμοποιώντας ένα σταθερό αριθμό (3) περασμάτων του αρχείου με τα δεδομένα. Το πρώτο πέρασμα, το *πέρασμα καταμέτρησης*, χρησιμοποιεί μία δομή πλέγματος και κατά τη σάρωση των δεδομένων καταγράφει των αριθμό των σημείων που περιέχει κάθε κελί. Αυτή η δομή χρησιμοποιείται για τον ορισμό άνω



Σχημα 4.1: Παράδειγμα πλέγματος a [1]

/κάτω φραγμάτων του σκορ των σημείων για το επόμενο πέρασμα. Το δεύτερο πέρασμα, το *πέρασμα φιλτραρίσματος*, εφαρμόζει κανόνες κλαδέματος, αξιοποιώντας τις τιμές που υπολογίστηκαν προηγουμένως, ώστε να αποκλείσει σημεία που δεν ανήκουν στο αποτέλεσμα και να κρατήσει τα υπόλοιπα σε ένα σύνολο υποψηφίων. Στο τελικό πέρασμα, το *πέρασμα διύλισης*, υπολογίζεται το ακριβές σκορ κυριαρχίας r για καθένα από τα σημεία που ανήκουν στο σύνολο των υποψηφίων και επιλέγονται τα k με τη μεγαλύτερη τιμή.

Στις παρακάτω υποενότητες περιγράφεται λεπτομερώς το κάθε πέρασμα-φάση του αλγορίθμου.

4.2.1 Πέρασμα Καταμέτρησης

Το πρώτο βήμα του αλγορίθμου ορίζει ένα κανονικό πολυδιάστατο πλέγμα πάνω στο χώρο και πραγματοποιεί γραμμική σάρωση των δεδομένων για να υπολογίσει τον αριθμό των σημείων μέσα σε κάθε κελί του. Ένα τέτοιο πλέγμα δύο διαστάσεων (4×4) φαίνεται στο σχήμα 4.1. Κατά τη σάρωση των σημείων αυξάνονται οι μετρητές των κελιών που τα περιέχουν, τα ίδια τα σημεία όμως δε διατηρούνται στην κύρια μνήμη. Έτσι, όπως φαίνεται στο πλέγμα που χρησιμοποιούμε για παράδειγμα, μετά το τέλος του πρώτου περάσματος έχουμε $count(g_{11}) = 0$ και $count(g_{12}) = 10$. Ένα σημείο μπορεί να ανήκει μόνο σε ένα κελί. Κατά σύμβαση, όταν ένα σημείο (π.χ. p_1) πέφτει στα σύνορα δύο κελιών (π.χ. g_{23} και g_{33}) τότε ανήκει στο κελί με τις μεγαλύτερες συντεταγμένες (π.χ. g_{33}).

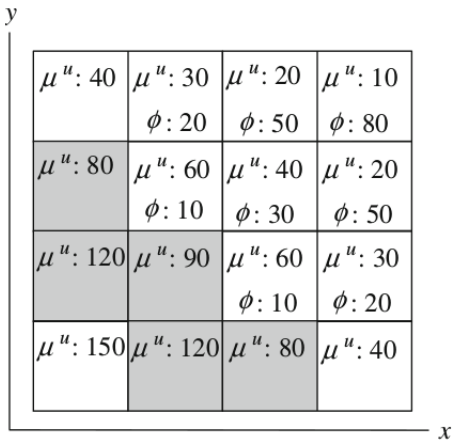
Μετά το πέρασμα της καταμέτρησης, και πριν ξεκινήσει το πέρασμα του φιλτραρίσματος, είναι δυνατός ο ορισμός άνω/κάτω φράγματος του σκορ των σημείων καθενός από τα κελιά, χρησιμοποιώντας το πλήθος των σημείων σε καθένα από αυτά. Αυτό επιτρέπει το γρήγορο εντοπισμό κελιών που δεν μπορεί να περιέχουν top-k κυρίαρχα σημεία.

Έχοντας ένα κελί g, ως άνω φράγμα του σκορ των σημείων που περιέχει ορίζεται ο συνολικός αριθμός των σημείων στα κελιά τα οποία κυριαρχεί μερικώς ή πλήρως:

$$r^u(g) = \sum_{g_y \in \mathcal{G} \wedge g^- \succ g_y^+} count(g_y)$$

Στο παράδειγμά μας, το κελί g_{33} κυριαρχεί τα g_{33}, g_{43}, g_{34} και g_{44}, επομένως όπως φαίνεται και στο σχήμα 4.2, $r^u(g_{33}) = 40$.

Ως κάτω φράγμα του σκορ των σημείων του κελιού g ορίζεται το πλήθος των σημείων εντός των κελιών τα οποία κυριαρχεί πλήρως:



Σχημα 4.2: Παράδειγμα πλέγματος b [1]

$$r^l(g) = \sum_{g_y \in \mathcal{G} \wedge g^+ \succ g_y^-} count(g_y)$$

Για παράδειγμα, το g_{33} κυριαρχεί πλήρως μόνο το g_{44} και έτσι $r^l(g_{33}) = 10$.

Πέρα από τα φράγματα των σκορ, κλάδεμα μπορεί να επιτευχθεί και με τη χρήση της ιδιότητας κυριαρχίας, από την οποία προκύπτει πως ένα σημείο δεν μπορεί να ανήκει στη λύση άμα κυριαρχείται από k άλλα σημεία. Για αυτόν το λόγο ορίζεται και ο μετρητής κυρίευσης $g.\phi$ του κελιού g, ο οποίος αντιστοιχεί στο πλήθος των σημείων των κελιών που κυριαρχούν πλήρως το g:

$$g.\phi = \sum_{g_y \in \mathcal{G} \wedge g_y^+ \succ g^-} count(g_y)$$

Στο παράδειγμά μας, το κελί g_{32} κυριαρχείται πλήρως από τα g_{11} και g_{21}, επομένως $g_{32}.\phi = 0 + 10 = 10$. Είναι ξεκάθαρο πως ένα κελί με $g_{ij}.\phi \geq k$ δε γίνεται να περιέχει κάποιο από τα top-k αποτελέσματα.

Ας υποθέσουμε πως στο παράδειγμά μας $k = 2$. Πριν την εκκίνηση του επόμενου περάσματος, προσδιορίζονται τα κελιά που δεν περιέχουν κανένα αποτέλεσμα. Σε αυτά συμπεριλαμβάνονται τα άδεια κελιά (π.χ. g_{11}) και τα κελιά με $g_{ij}.\phi \geq k$ (π.χ. g_{23}). Σειρά έχει ο προσδιορισμός του γ, το οποίο αποτελεί κάτω φράγμα του σκορ των top-k σημείων. Για να πάρουμε την τιμή του, απαριθμούνται με φθίνουσα σειρά ως προς το r^l τα υπόλοιπα κελιά, μέχρι ο αριθμός των συνολικών σημείων να φτάσει το k. Η τιμή του γ είναι το r^l του κελιού στο οποίο σταμάτησε η απαρίθμηση. Στο παράδειγμά μας το κελί g_{12} περιέχει 10 ($\geq k$) σημεία, και αφού $r^l(g_{12}) = 60$ ορίζουμε $\gamma = 60$. Προφανώς τα κελιά με $r^u < \gamma$ αποκλείται να περιέχουν κάποιο σημείο που ανήκει στο αποτέλεσμα και επομένως μπορούν να κλαδευτούν. Στο σχήμα 4.2 τα κελιά που παραμένουν, καθώς μπορεί να περιέχουν top-k κυρίαρχα σημεία, είναι μαρκαρισμένα με γκρι χρώμα.

4.2.2 Πέρασμα Φιλτραρίσματος

Κατά το δεύτερο πέρασμα, ο αλγόριθμος σαρώνει τα δεδομένα ξανά και καθορίζει ένα σύνολο από σημεία υποψήφια για το ερώτημα top-k κυριαρχίας. Ο αλγόριθμος αυτός λέγεται *coarse-grained filter* (CRS) και χρησιμοποιεί τις τιμές κυρίευσης $g.\phi$ των κελιών σε συνδιασμό με την ιδιότητα κυριαρχίας για να κλαδέψει σημεία. Ο CRS περιγράφεται στον αλγόριθμο 1.

Algorithm 1 Αλγόριθμος Coarse-Grained Filter

```
1: function CRS-Filter(Dataset 𝒟, Integer k, Grid 𝒢)
2:     for all cell g ∈ 𝒢 do
3:         g.C := new set;                                  ▷ candidate set of the cell
4:     end for
5:     for all p ∈ 𝒟 do                                     ▷ filter scan
6:         let g_p be the grid cell of p;
7:         p.φ := g_p.φ;
8:         for all cells g_z ∈ 𝒢 such that g_z^- ≻ g_p^+ ∧ g_z^+ ⊁ g_p^- do
9:             for all p' ∈ g_z.C such that p' ≻ p do
10:                p.φ := p.φ + 1;
11:                if p.φ ≥ k then
12:                    ignore further processing for the point p;
13:                end if
14:            end for
15:        end for
16:        for all cells g_z ∈ 𝒢 such that g_p^- ≻ g_z^+ ∧ g_p^+ ⊁ g_z^- do
17:            for all p' ∈ g_z.C such that p ≻ p' do
18:                p'.φ := p'.φ + 1;
19:                if p'.φ ≥ k then
20:                    remove p' from g_z.C;
21:                end if
22:            end for
23:        end for
24:        if p.φ < k then
25:            insert p into g_p.C;
26:        end if
27:    end for
28: end function
```

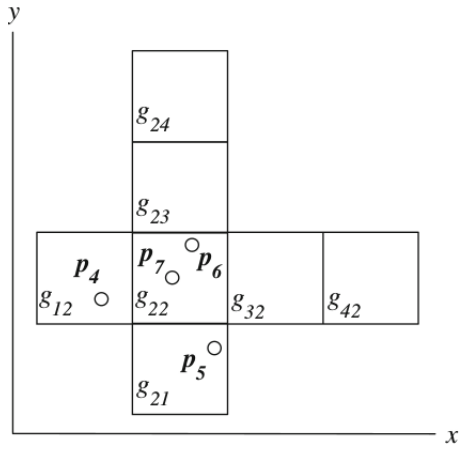
Για κάθε κελί g, το οποίο δεν κλαδεύτηκε μετά το πρώτο πέρασμα, ορίζεται ένα σύνολο υποψηφίων $g.C$ στο οποίο κρατάει τα υποψήφια σημεία που βρίσκονται μέσα σε αυτό. Στη συνέχεια ο αλγόριθμος πραγματοποιεί γραμμική σάρωση του συνόλου δεδομένων $\mathcal{D}$, στις γραμμές 5-27. Για το σημείο p που εξετάζεται κάθε φορά, αρχικοποιείται το $p.\phi$ με την αντίστοιχη τιμή του κελιού g_p στο οποίο ανήκει το p.

Στο βρόγχο των γραμμών 8-15, γίνεται αναζήτηση υποψήφιων σημείων p' που κυριαρχούν το p και έχουν ήδη διαβαστεί στη μνήμη. Για κάθε τέτοιο σημείο, η τιμή $p.\phi$ αυξάνεται κατά ένα. Λόγω του ότι στο $p.\phi$ έχει συνυπολογιστεί το $g_p.\phi$ του κελιού, αρκεί να εξεταστούν για σημεία p' μόνο τα κελιά που κυριαρχούν μερικώς το κελί του σημείου p (αντί για όλα). Σε περίπτωση που το $p.\phi$ φτάσει το k, δε χρειάζεται να αυξηθεί περαιτέρω και ο βρόγχος τερματίζει. Σε αυτήν την περίπτωση το p αποκλείεται να ανήκει στα top-k.

Στο βρόγχο των γραμμών 16-23, γίνεται αναζήτηση υποψήφιων σημείων p' τα οποία κυριαρχούναι από το p και έχουν ήδη διαβαστεί στη μνήμη. Το $p'.\phi$ κάθε τέτοιου σημείου p' αυξάνεται κατά ένα, και όταν φτάσει το k κλαδεύεται από το σύνολο υποψηφίων του κελιού του. Για τον ίδιο λόγω με προηγουμένως, εξετάζονται για σημεία p' κελιά τα οποία κυριαρχούνται μερικώς από το κελί του p (αντί για όλα). Να σημειωθεί πως αυτός ο βρόγχος δεν εκτελείται εάν το $p.\phi$ έχει φτάσει το k. Ο λόγος είναι πως οποιοσδήποτε υποψήφιος p' που κυριαρχείται από το p πρέπει να κυριαρχείται και από τα k σημεία που κυριαρχούν το p και άρα έχει ήδη κλαδευτεί σε κάποια προηγούμενη επανάληψη.

Στις γραμμές 24-26, εφόσον το $p.\phi$ του σημείου p είναι μικρότερο του k, το p τοποθετείται στο σύνολο υποψηφίων του κελιού του.

4.2.3 Πέρασμα Διύλισης



Σχημα 4.3: Παράδειγμα πλέγματος c [1]

Μετά το τέλος του περάσματος φιλτραρίσματος έχει προκύψει ένα σύνολο $\mathcal{C}$ με υποψήφια σημεία, μέσα στα οποία βρίσκονται τα top-k. Στο πέρασμα διύλισης πραγματοποιείται γραμμική σάρωση πάνω στο αρχείο $\mathcal{D}$, κατά την οποία κάθε σημείο $p' \in \mathcal{D}$ συγκρίνεται με κάθε υποψήφιο σημείο $p \in \mathcal{C}$, με το σκορ του p να αυξάνεται κατά ένα κάθε φορά που το p κυριαρχεί το p'. Η ευθύς υλοποίηση απαιτεί $|\mathcal{D}| \cdot |\mathcal{C}|$ ελέγχους κυριαρχίας, κάτι που ακόμα και για σύνολο υποψηφίων μετρίου μεγέθους είναι ιδιαίτερα χρονοβόρο.

Προκειμένου να επιταγχυνθεί το πέρασμα διύλισης, γίνεται χρήση του κάτω φράγματος του σκορ των κελιών του πλέγματος. Ας υποθέσουμε πως το σημείο p_7 του σχήματος 4.3 είναι υποψήφιο. Εφόσον πέφτει μέσα στο κελί g_{22}, το κάτω φράγμα του σκορ του ορίζεται ως $r^l(p_7) = r^l(g_{22}) = 40$. Πλέον, κατά τη σάρωση του $\mathcal{D}$ στο πέρασμα διύλισης, δε χρειάζεται να συγκριθεί κάθε σημείο $p' \in \mathcal{D}$ με το p_7, παρά μόνο τα σημεία p' τα οποία ανήκουν σε κελί που κυριαρχείται μερικώς από το g_{22} (π.χ. g_{22}, g_{23}, g_{24}, g_{32}, g_{42}).

Algorithm 2 Αλγόριθμος Διύλισης με Πλέγμα

```
1:  function GridRefinement(Dataset 𝒟, Integer k, Grid 𝒢)
2:      for all cell g ∈ 𝒢 do
3:          if ¬∃g_z ∈ 𝒢, (|g_z.C| > 0) ∧ (g_z^+ ⊁ g^- ∧ g_z^- ≻ g^+) then
4:              mark cell g as irrelevant;
5:          end if
6:          for all p ∈ g.C do
7:              r^l(p) := r^l(g);
8:          end for
9:      end for
10:     for all p' ∈ 𝒟 do
11:         let g_p' be the grid cell of p';
12:         if g_p' is irrelevant then
13:             ignore further processing for point p';
14:         end if
15:         for all cell g ∈ 𝒢 such that g^- ≻ g_p'^+ ∧ g^+ ⊁ g_p'^- do
16:             for all p ∈ g.C such that p ≻ p' do
17:                 r^l(p) := r^l(p) + 1;
18:             end for
19:         end for
20:     end for
21:     return k points in ∪_{g∈𝒢} g.C with the highest r^l scores;
22: end function
```

Στον αλγόριθμο 2 φαίνονται σε ψευδοκώδικα τα βήματα που πραγματοποιούνται κατά το τρίτο και τελευταίο πέρασμα. Το $\mathcal{G}$ αντιστοιχεί στο πλέγμα που προέκυψε από το πέρασμα καταμέτρησης. Κάθε κελί $g \in \mathcal{G}$ σχετίζεται με ένα σύνολο υποψηφίων $g.C$, στο οποίο διατηρεί τα υποψήφια σημεία (από το πέρασμα φιλτραρίσματος) τα οποία ανήκουν σε αυτό. Στη γραμμή 3 γίνεται έλεγχος άμα το κελί g δεν κυριαρχείται μερικώς από κανένα άλλο κελί g_z το οποίο περιέχει υποψήφια σημεία. Σε αυτήν την περίπτωση το κελί μαρκάρεται ως *αδιάφορο*, καθώς δεν επηρεάζει το τελικό αποτέλεσμα. Στη γραμμή 7, τα r^l των υποψηφίων παίρνουν ως τιμή το r^l του κελιού g στο οποίο ανήκουν.

Στη συνέχεια, πραγματοποιείται σάρωση του συνόλου δεδομένων $\mathcal{D}$. Στην περίπτωση που το κελί $g_{p'}$ του τρέχοντος σημείου $p' \in \mathcal{D}$ που εξετάζεται είναι αδιάφορο, τότε το σημείο p' απορρίπτεται κατευθείαν χωρίς καμία περαιτέρω επεξεργασία. Στις γραμμές 15-19, μόνο τα κελιά που κυριαρχούν μερικώς το κελί του p' χρειάζεται να ληφθούν υπόψιν. Κάθε υποψήφιος p που βρίσκεται σε τέτοιο κελί συγκρίνεται με το p', και το σκορ $r^l(p)$ του p αυξάνεται κατά ένα όταν αυτό κυριαρχεί το p'. Μετά το τέλος της σάρωσης, επιστρέφονται τα k υποψήφια σημεία με το υψηλότερο σκορ, ως το αποτέλεσμα του ερωτήματος top-k κυριαρχίας.

4.3 Κατανεμημένος αλγόριθμος

Η υλοποίηση εκτέλεσης ερωτημάτων top-k κυριαρχίας για Spark που προτείνουμε, στηρίζεται στις βασικές αρχές του σειριακού αλγορίθμου που παρουσιάστηκε στην προηγούμενη ενότητα. Κάνοντας την υπόθεση ότι εργαζόμαστε σε έναν cluster και τα δεδομένα είναι μοιρασμένα στους κόμβους αυτού, κύριο μέλημα είναι η ελαχιστοποίηση μεταφορών δεδομένων μεταξύ των κόμβων. Η προτεινόμενη υλοποίηση το καταφέρνει πολύ καλά αυτό, καθώς τα μόνα δεδομένα που μεταφέρονται κατά την εκτέλεση της είναι το πλέγμα και το σύνολο υποψηφίων, αντικείμενα μικρού μεγέθους σε σχέση με το συνολικό dataset.

Η προτεινόμενη υλοποίηση αποτελείται από τρεις φάσεις, με καθεμία να αντιστοιχεί σε ένα από τα περάσματα του σειριακού αλγορίθμου. Ως είσοδο δέχεται ένα αρχείο κειμένου το οποίο περιέχει το σύνολο δεδομένων. Το αρχείο αυτό μπορεί να βρίσκεται είτε τοπικά είτε στο κατανεμημένο σύστημα αρχείων HDFS, χωρισμένο σε blocks. Επιπλέον, δίνεται από το χρήστη η τιμή του k, ο αριθμός διαστάσεων d του συνόλου δεδομένων και ο επιθυμητός αριθμός κελιών ανά διάσταση n για το πλέγμα (δηλ. το πλέγμα αποτελείται από n^d κελιά). Τέλος δίνεται το μονοπάτι στο οποίο ο driver θα αποθηκεύσει το αποτέλεσμα του ερωτήματος top-k κυριαρχίας.

Στις τρεις υποενότητες που ακολουθούν περιγράφονται οι τρεις φάσεις της υλοποίησης. Σε κάθε υποενότητα, αρχικά δίνεται η βασική υλοποίηση της φάσης, ενώ στη συνέχεια προτείνεται κάποια βελτιστοποίηση που την κάνει αποδοτικότερη.

4.3.1 1η Φάση

Στόχος της πρώτης φάσης, όπως και του περάσματος καταμέτρησης, είναι η κατασκευή πλέγματος και το κλάδεμα κελιών του που δε γίνεται να περιέχουν σημεία τα οποία ανήκουν στο αποτέλεσμα. Πρώτο βήμα είναι η καταμέτρηση των σημείων που περιέχει κάθε κελί, κάτι που όπως θα δούμε δεν επηρεάζεται ιδιαίτερα από το γεγονός ότι τα σημεία βρίσκονται σε διαφορετικές τοποθεσίες μεταξύ τους στον cluster.

Με την εκκίνηση της εφαρμογής γίνεται broadcast κάποιων τιμών όπως το k, το d και το n στους executors. Στη συνέχεια, το αρχείο με τα σημεία φορτώνεται σε ένα RDD, στο οποίο εφαρμόζεται ένας μετασχηματισμός map που το φέρνει στη μορφή *(συντεταγμένες σημείου, id κελιού στο οποίο ανήκει)*. Υπενθυμίζουμε πως στην πραγματικότητα τα δεδομένα δεν έχουν φορτωθεί, ούτε έχει εφαρμοστεί ο μετασχηματισμός, καθώς δεν έγινε κάποια πράξη που να χρησιμοποιεί το μετασχηματισμένο RDD, έστω *data*. Επειδή όμως στη συνέχεια της υλοποίησης το *data* χρησιμοποείται πολύ συχνά, καλείται η persist() για αυτό.

Αφού φορτωθεί το αρχείο με τα σημεία και γίνει ανάθεση κάθε σημείου σε ένα κελί, σειρά έχει η καταμέτηση των σημείων ανά κελί. Αυτό γίνεται εύκολα με τους παρακάτω μετασχηματισμούς που προσφέρει το Spark:

```
val cellCounters = data.map(x => (x._2, 1: Long)).reduceByKey(_ + _)
```

Ουσιαστικά, ο πρώτος μετασχηματισμός που εφαρμόζεται στο *data* δίνει ένα RDD της μορφής *(id κελιού, 1)*, ενώ ο δεύτερος ομαδοποιεί τις εγγραφές ανά id και αθροίζει τις μονάδες. Έτσι, το RDD *cellCounters* που προκύπτει είναι της μορφής *(id κελιού, αριθμός σημείων που περιέχει)*. Επειδή για τον υπολογισμό άνω/κάτω φραγμάτος σκορ και τιμής

κυρίευσης των σημείων κάθε κελιού απαιτείται πληροφορία και για τα υπόλοιπα κελιά, καλείται collect() για το $cellCounters$ και οι τιμές αυτές υπολογίζονται τοπικά στον driver.

Ο υπολογισμός των άνω/κάτω φραγμάτων μπορεί να επιταγχυνθεί σημαντικά από το γεγονός ότι το r^l ενός κελιού $g_{i,j}$ ισούται με το r^u του κελιού $g_{i+1,j+1}$ (αντίστοιχα για τρεις διαστάσεις $r^l(g_{i,j,k}) = r^u(g_{i+1,j+1,k+1})$, κ.ο.κ.), μειώνοντας έτσι σημαντικά τον αριθμό των κελιών που επισκέπτονται. Προφανώς για να εκμεταλευτούμε αυτή τη σχέση θα πρέπει να έχει προηγηθεί ο υπολογισμός του $r^u(g_{i+1,j+1})$ και γι΄ αυτό δίνεται προτεραιότητα σε κελιά με μεγαλύτερες συντεταγμένες. Επιπλέον, για να υπολογιστεί το r^u ενός κελιού g έχοντας το r^l του, αρκεί να προσθέσουμε σε αυτό το πλήθος των σημειών που βρίσκονται στα κελιά που κυριαρχούνται μερικώς από το g. Για παράδειγμα, στο σχήμα 4.1, το r^l του κελιού g_{33} ισούται με το r^u του κελιού g_{44}, ενώ το r^u του ισούται με το άθροισμα του $r^l(g_{33})$ με τον αριθμό των σημείων στα κελιά g_{33}, g_{43} και g_{34}.

Όσον αφορά τον υπολογισμό τιμής κυρίευσης, τα πράγματα δε διαφέρουν και πολύ. Η τιμή κυρίευσης $g.\phi$ ενός κελιού, η οποία αντιστοιχεί στον αριθμό σημείων στα κελιά που κυριαρχούν πλήρως το κελί g, αποτελεί στην ουσία κάτω φράγμα του αριθμού των σημείων που κυριαρχούν τα σημεία μέσα στο g. Ορίζοντας ένα αντίστοιχο άνω φράγμα, έστω $g.\phi^u$, μπορούμε με παρόμοιο τρόπο με προηγουμένως να εκμεταλευτούμε το γεγονός πως το $g.\phi$ ενός κελιού $g_{i,j}$ ισούται με το $g.\phi^u$ του κελιού $g_{i-1,j-1}$. Οι διαφορές είναι πως τώρα προτεραιότητα έχουν κελιά με μικρότερες συντεταγμένες, ενώ για τον υπολογισμό του $g.\phi^u$ ενός κελιού αθροίζεται το $g.\phi$ του με τον αριθμό των σημείων στα κελιά που κυριαρχούν μερικώς το g.

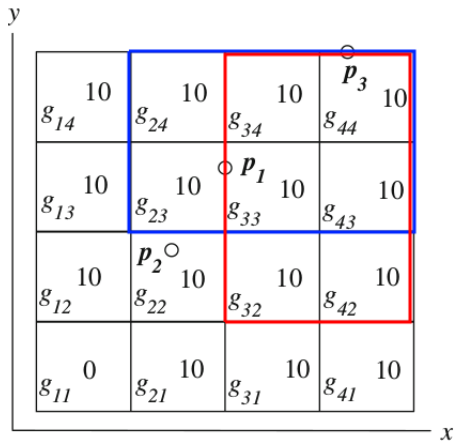
Έχοντας υπολογίσει όλες τις απαιτούμενες τιμές, κλαδεύονται τα κελιά τα οποία δεν έχουν καθόλου σημεία και τα κελιά με $g.\phi \geq k$. Τα κελιά που παραμένουν τοποθετούνται σε ένα σωρό μεγίστων ως προς το r^l τους. Στη συνέχεια αφαιρείται η ρίζα του σωρού όσες φορές χρειαστεί μέχρι το άθροισμα των σημείων των κελιών που αφαιρούνται να φτάσει το k. Το γ ισούται με το r^l τελευταίου κελιού που αφαιρέθηκε από το σωρό. Γυρνώντας στα κελιά που παρέμειναν μετά το πρώτο κλάδεμα, κλαδεύονται επιπλέον όσα έχουν $r^u < \gamma$. Τα ids των κελιών που δεν κλαδεύτηκαν, μαζί με τα r^l και $g.\phi$ τους, αναμεταδίδονται μέσω broadcast στους executors, και η 1η φάση ολοκληρώνεται.

Βελτιστοποίηση Υπολογισμού Άνω Φραγμάτων

Όπως είδαμε πιο πάνω, ο υπολογισμός του r^l (και του $g.\phi$) ενός κελιού g γίνεται με σταθερό αριθμό (1) επισκέψεων σε άλλα κελιά και είναι ιδιαίτερα φθηνός. Αντιθέτως, για τον υπολογισμό του r^u (και του $g.\phi^u$) ενός κελιού g απαιτείται η επίσκεψη σε όλα τα κελιά που κυριαρχούνται (ή κυριαρχούν) μερικώς από το g προκειμένου να αθροιστούν τα σημεία που περιέχουν. Μια τέτοια διαδικασία είναι ιδιαίτερα ακριβή, ειδικά όταν έχουμε να κάνουμε με μεγάλα δεδομένα, τα οποία όπως είναι λογικό χρειάζονται μεγαλύτερο πλέγμα.

Χρησιμοποιώντας την αρχή έγκλεισης-απόκλεισης (inclusion-exclusion principle), είναι δυνατόν να επιταγχύνουμε δραματικά τον υπολογισμό των άνω φραγμάτων, περιορίζοντας τον αριθμό των κελιών που πρέπει να επισκεφθούμε. Βάσει αυτής της αρχής, ο πληθάριθμος της ένωσης δύο συνόλων A και B ισούται με το άθροισμα των πληθαρίθμων των δύο συνόλων μείον τον πληθάριθμο της τομής τους ($|A \cup B| = |A| + |B| - |A \cap B|$), ενώ η αρχή αυτή επεκτείνεται και για μεγαλύτερο αριθμό συνόλων.

Ο τρόπος με τον οποίο μπορεί να αξιοποιηθεί η αρχή έγκλεισης-απόκλεισης για τον υπολογισμό άνω φράγματος φαίνεται στο σχήμα 4.4. Έστω ότι βρισκόμαστε στο κελί



Σχημα 4.4: Εφαρμογή Αρχής Έγκλεισης-Απόκλεισης στο Πλέγμα

g_{22} και θέλουμε να υπολογίσουμε το r^u του. Αν υποθέσουμε πως έχουμε δύο σύνολα. το κόκκινο και το μπλε, τα οποία περιέχουν τα σημεία των κελιών που περικλύουν, για να βρούμε το $r^u(g_{22})$ αρκεί να αθροίσουμε το πλήθος των σημείων του g_{22} με τον πληθάριθμο της ένωσης των δύο συνόλων. Εφαρμόζοντας την αρχής έγκλεισης-απόκλεισης, ο πληθάριθμος της ένωσης του κόκκινου συνόλου με το μπλε ισούται με τον πληθάριθμο του κόκκινου συν τον πληθάριθμου του μπλε μείον τον πληθάριθμο της τομής τους. Όμως και οι τρεις αυτές τιμές είναι γνωστές, καθώς ο πληθάριθμος του κόκκινου συνόλου ισούται με $r^u(g_{32})$, ο πληθάριθμος του μπλε ισούται με $r^u(g_{23})$ και ο πληθάριθμος της τομής τους ισούται με $r^u(g_{33})$. Έτσι, με τρεις μόνο επισκέψεις σε άλλα κελιά είναι δυνατός ο υπολογισμός του $r^u(g_{22})$.

Με αντίστοιχο τρόπο μπορεί να υπολογιστεί το $g.\phi^u$. Σε περιπτώσεις πλέγματος παραπάνω διαστάσεων αυξάνεται και ο αριθμός των νοητών συνόλων, και κατ΄ επέκταση ο αριθμός των κελιών που πρέπει να επισκεφθούμε για να πάρουμε τους πληθαρίθμους τους και τους πληθαρίθμους των μεταξύ τους τομών. Συγκριτικά όμως με την αρχική μέθοδο που χρησιμοποιήθηκε, το υπολογιστικό κόστος είναι κατά πολύ μικρότερο.

4.3.2 2η Φάση

Στόχος της δεύτερης φάσης είναι η εξαγωγή ενός συνόλου σημείων υποψήφιων για top-k, όπως ακριβώς και στο πέρασμα φιλτραρίσματος του σειριακού αλγορίθμου. Μετά το τέλος της πρώτης φάσης, κάθε executor διαθέτει μια λίστα με τα ids των κελιών που δεν κλαδεύτηκαν, μαζί με τα r^l και $g.\phi$ τους. Το πρώτο βήμα της δεύτερης φάσης είναι η εφαρμογή μετασχηματισμού filter πάνω στο RDD *data*, ώστε με τη χρήση αυτής της λίστας να προκύψει ένα RDD, έστω *intermediateCandidates*, που περιέχει μόνο τα σημεία που ανήκουν σε κελιά που δεν κλαδεύτηκαν.

Για τη συνέχεια της δεύτερης φάσης ορίζεται μια συνάρτηση, έστω *pruneDominated–Candidates*. Η συγκεκριμένη συνάρτηση δέχεται ένα σημείο p και μια λίστα από σημεία. Το $p.\phi$ του p αρχικοποιείται με την αντίστοιχη τιμή του κελιού στο οποίο ανήκει, έστω g_p, και στη συνέχεια κάθε ένα σημείο της λίστας που ανήκει σε κελί που κυριαρχεί μερικώς το g_p συγκρίνεται με το p. Για κάθε ένα τέτοιο σημείο από το οποίο το p κυριαρχείται, το $p.\phi$ αυξάνεται κατά ένα. Σε περίπτωση που στο τέλος της λίστας το $p.\phi$ έχει φτάσει το k, το σημείο p απορρίπτεται ως υποψήφιος, καθώς κυριαρχείται από τουλάχιστον k άλλα σημεία.

Έχοντας ορίσει αυτή τη συνάρτηση, στη συνέχεια εφαρμόζεται ένας μετασχηματισμός mapPartitions πάνω στο *intermediateCandidates*. Σε κάθε partition αυτού του RDD, για κάθε ένα σημείο που περιέχει εφαρμόζεται η *prunedDominatedCandidates*. Η λίστα που δίνεται κάθε φορά στην *prunedDominatedCandidates* είναι ολόκληρο το partition. Μετά την εφαρμογή του mapPartitions, για το RDD που προκύπτει, έστω *perPartitionCandidates*, καλείται collect().

Στα σημεία του *perPartitionCandidates* μπορεί να ανήκει κάποιο σημείο p, το οποίο κυριαρχείται από k ή παραπάνω άλλα σημεία, αλλά δεν απορρίφθηκε ως υποψήφιο κατά το μετασχηματισμό mapPartitions. Αυτό συμβαίνει στην περίπτωση που στο partition που ανήκει το p, τα σημεία που ανήκουν σε κελιά που κυριαρχούν μερικώς το g_p και τα ίδια κυριαρχούν το p, δεν είναι αρκετά ώστε να κάνουν το $p.\phi$ να φτάσει το k, αλλά υπάρχουν άλλα τέτοια σημεία σε άλλα partitions με τα οποία θα το έφτανε. Για αυτό το λόγο, όταν με το collect συγκεντρώνονται τα σημεία του *perPartitionCandidates* στον driver, ξαναεκτελείται για καθένα από αυτά η *prunedDominatedCandidates*, δίνοντας κάθε φορά για λίστα όλα τα σημεία που παρέμειναν μετά τη mapPartitions.

Στην ουσία κατά το μετασχηματισμό mapPartitions γίνεται ένα πρώτο ξεκαθάρισμα, όσο αυτό είναι δυνατόν μόνο με τα σημεία του κάθε partition, αλλά το τελικό κλάδεμα σημείων γίνεται στον driver. Ο λόγος που παρεμβάλεται ο μετασχηματισμός mapPartitions είναι για να μειώθει όσο γίνεται ο αριθμός των σημείων που θα πάνε στον driver, καθώς για κάθε ένα σημείο ελέγχονται όλα τα άλλα, και με μερικά από αυτά γίνεται και σύγκριση κυριαρχίας, που είναι μία σχετικά ακριβή πράξη. Αφού γίνει και το τελικό κλάδεμα στον driver, το σύνολο των υποψηφίων που προκύπτει αναμεταδίδεται με broadcast στους executors και η δεύτερη φάση ολοκληρώνεται.

Κατανεμημένο Κλάδεμα Σημείων

Κατά τη βασική υλοποίηση κλαδέματος σημείων που παρουσιάστηκε, το κλάδεμα λαμβάνει χώρα τοπικά στον driver. Παρεμβάλεται βέβαια ο μετασχηματισμός mapPartitions ώστε να μειωθούν τα σημεία και κατ΄ επέκταση οι υπολογισμοί που θα χρειαστεί να γίνουν, όμως στην πράξη, όπως έδειξαν τα πειράματα, ο αριθμός σημείων που κλαδεύεται κατά το μετασχηματισμό είναι αμελητέος.

Για τους παραπάνω λόγους προτείνεται μία εναλλακτική υλοποίηση, κατά την οποία ο μετασχηματισμός mapPartitions παραλείπεται τελειώς και εκτελείται απευθείας collect για το *intermediateCandidates*. Στη συνέχεια, δημιουργείται στον driver μία μεταβλητή broadcast με τα σημεία του *intermediateCandidates*, έστω *bcIntermediateCandidates*, η οποία αναμεταδίδεται στους executors. Τέλος, για κάθε σημείο του *intermediateCandidates* εκτελείται η *prunedDominatedCandidates*, δίνοντας για λίστα κάθε φορά την τιμή του *bcIntermediateCandidates*. Το σύνολο των υποψηφίων που προκύπτει, συγκεντρώνεται μέσω collect στον driver, απ΄ όπου αναμεταδίδεται στους executors για την επόμενη φάση.

Το πλεονέκτημα αυτής της προσέγγισης είναι πως το κλάδεμα πραγματοποιείται παράλληλα. Κάθε executor διαθέτει όλα τα σημεία μέσω του *bcIntermediateCandidates*, καλεί όμως την *prunedDominatedCandidates* μόνο για τα σημεία που βρίσκονται στα δικά του partitions του *intermediateCandidates*. Ως μειονέκτημα μπορεί να θεωρηθεί το γεγονός ότι γίνεται μία επιλέον αναμετάδοση, όμως όπως αναφέρθηκε στο κεφάλαιο 3, το Spark φροντίζει αυτή η αναμετάδοση να γίνεται με το βέλτιστο και οικονομικότερο τρόπο.

4.3.3 3η Φάση

Στόχος της τρίτης φάσης της υλοποίησης, όπως και του περάσματος διύλισης στο σειριακό αλγόριθμο, είναι ο υπολογισμός ακριβούς σκορ για κάθε ένα από τα υποψήφια σημεία, ώστε να επιστραφούν τα k με το μεγαλύτερο σκορ ως αποτέλεσμα του ερωτήματος top-k κυριαρχίας. Μετά το τέλος και της δεύτερης φάσης, κάθε executor διαθέτει μια λίστα με τα υποψήφια σημεία και μια λίστα με τα ids των κελιών που δεν κλαδεύτηκαν, μαζί με τα r^l και $g.\phi$ τους.

Για την εκτέλεση της τρίτης φάσης ορίζεται μια συνάρτηση, έστω *scoreCalculationAssistant*. Η συνάρτηση αυτή δέχεται ένα σημείο p και μια λίστα από υποψήφια σημεία. Εάν το p ανήκει σε κελί το οποίο κυριαρχείται μερικώς από κελί υποψηφίου, τότε γίνεται σύγκριση κυριαρχίας μεταξύ των δύο. Σε περίπτωση που το υποψήφιο σημείο κυριαρχεί το p, τότε στην έξοδο της συνάρτησης προστίθεται τιμή της μορφής *(υποψήφιο σημείο, 1)*. Αυτή η διαδικασία επαναλαμβάνεται για όλους τους υποψήφιους της λίστας.

Κατά τη βασική υλοποίηση της τρίτης φάσης, η παραπάνω συνάρτηση εφαρμόζεται σε κάθε σημείο του *data* με τη λίστα που δίνεται κάθε φορά να είναι η λίστα υποψηφίων από τη δεύτερη φάση, που διαθέτουν όλοι οι executors. Μετά την εφαρμογή αυτής της συνάρτησης, γίνεται χρήση του μετασχηματισμού reduceByKey, με παρόμοιο τρόπο με την πρώτη φάση, ώστε να γίνει ομαδοποίηση ανά υποψήφιο και να αθροιστούν οι μονάδες. Σε αυτό το άθροισμα κάθε υποψηφίου προστίθεται και το r^l του κελιού του, και έτσι προκύπτει ένα RDD, έστω *candidateScores*, το οποίο περιέχει τα υποψήφια σημεία μαζί με τα σκορ τους. Στη συνέχεια, εκτελείται collect στο *candidateScores* ώστε να συγκεντρωθούν οι υποψήφιοι με τα σκορ τους στον driver και να τοποθετηθούν σε σωρό μεγίστων βάσει του σκορ τους. Τέλος, αφαιρείται από το σωρό η ρίζα k φορές. Τα k σημεία που αφαιρέθηκαν από το σωρό είναι το αποτέλεσμα του ερωτήματος top-k κυριαρχίας.

Κλάδεμα Αδιάφορων Σημείων του *data*

Τα σημεία του *data* τα οποία παρουσιάζουν ενδιαφέρον κατά την τρίτη φάση είναι αυτά που ανήκουν σε κάποιο κελί το οποίο κυριαρχείται μερικώς από το κελί κάποιου υποψηφίου. Για αυτό το λόγο, στην *scoreCalculationAssistant* για κάθε έναν υποψήφιο της λίστας γίνεται έλεγχος αν το κελί του κυριαρχεί μερικώς το κελί του τρέχοντος σημείου p που εξετάζεται. Είναι πολύ συχνό, αυτό το σημείο p να είναι σε κελί που κυριαρχείται μερικώς από το κελί ενός υποψήφιου αλλά να είναι αδιάφορο για κάποιον άλλον υποψήφιο. Ύπάρχουν όμως και σημεία, των οποίων τα κελιά δεν κυριαρχούνται μερικώς από το κελί κανενός υποψηφίου, και άρα δε συνεισφέρουν τίποτα κατά την τρίτη φάση. Αυτά τα σημεία δε συμμετέχουν σε καμία σύγκριση κυριαρχίας, όμως κάθε φορά γίνεται έλεγχος του κελιού τους για καθέναν από τους υποψηφίους. Κάτι τέτοιο πέρα από άσκοπο είναι και ιδιαίτερα χρονοβόρο.

Για τον παραπάνω λόγο, προτείνεται μία βελτιστοποίηση κατά την οποία αρχικά προσδιορίζονται τα κελιά που δεν κυριαρχούνται μερικώς από το κελί κανενός υποψηφίου. Αυτό γίνεται συγκεντρώνοντας το μεγαλύτερο δείκτη κελιού, στο οποίο υπάρχει υποψήφιος, ανά διάσταση. Αυτό σημαίνει πως αν τα υποψήφια σημεία βρίσκονται στα κελιά g_{11}, g_{12} και g_{21}, τότε οι μεγαλύτεροι δείκτες θα ήταν το 2 για την πρώτη διάσταση (λόγω του g_{21}) και το 2 για τη δεύτερη (λόγω του g_{12}). Κάθε κελί το οποίο έχει μεγαλύτερες τιμές σε όλους τους δείκτες είναι ένα κελί που δεν κυριαρχείται μερικώς από το κελί κανενός υποψήφιου σημείου (π.χ. g_{33}, g_{43}, g_{34}, g_{44}). Γνωρίζοντας αυτό είναι δυνατή η εφαρμογή μετασχηματισμού filter πάνω στο *data*, ώστε να προκύψει ένα RDD, έστω $filteredData$,

μόνο με τα απαιτούμενα για την τρίτη φάση σημεία.

Έχοντας το $filteredData$, η εφαρμογή της $scoreCalculationAssistant$ πραγματοποιείται στα σημεία του αντί τα σημεία του $data$, γλιτώνοντας έτσι άσκοπους και χρονοβόρους ελέγχους.

4.4 Επίλογος και Σύνοψη

Σε αυτό το κεφάλαιο έγινε παρουσίαση της προτεινόμενης υλοποίησης εκτέλεσης ερωτημάτων top-k κυριαρχίας σε Spark. Αρχικά παρουσιάστηκε ένας σειριακός αλγόριθμος στον οποίο στηρίζεται η υλοποίηση. Ο αλγόριθμος αυτός εργάζεται με ένα σύνολο δεδομένων $\mathcal{D}$, το οποίο βρίσκεται στο δίσκο, και πραγματοποιεί σταθερό αριθμό (3) περασμάτων για τον υπολογισμό του αποτελέσματος. Τα περάσματα αυτά είναι το πέρασμα καταμέτρησης, το πέρασμα φιλτραρίσματος και το πέρασμα διύλισης. Στο πρώτο γίνεται εφαρμογή ένος πλέγματος στο χώρο των δεδομένων με στόχο το κλάδεμα κελιών που τα σημεία τους δεν είναι μέρος της λύσης. Στο δεύτερο, με την εφαρμογή περαιτέρω κλαδέματος σε επίπεδο σημείων, εξάγεται ένα σύνολο υποψηφίων και στο τρίτο υπολογίζεται το ακριβές σκορ αυτών και επιστρέφονται οι k με το υψηλότερο.

Στη συνέχεια παρουσιάζεται αναλυτικά η υλοποίηση που προτείνουμε. Αποτελείται από τρεις φάσεις, με κάθε φάση να αντιστοιχεί και να έχει ίδιο σκοπό με τα περάσματα του σειριακού αλγορίθμου. Προφανώς γίνονται τροποποιήσεις αυτού για συμβατότητα σε κατανεμημένο περιβάλλον. Αρχικά δίνεται για κάθε φάση η βασική της υλοποίηση, ενώ στο τέλος δίνεται μια περαιτέρω βελτιστοποιήση που την κάνει αποδοτικότερη.

Κεφάλαιο 5

Εφαρμογή Υλοποίησης σε Δεδομένα

5.1 Εισαγωγή

Σε αυτό το κεφάλαιο παρατίθονται τα αποτελέσματα της εκτέλεσης της προτεινόμενης υλοποίησης, και των βελτιστοποιήσεων της, πάνω σε σύνολα δεδομένων διαφόρων μεγεθών και κατανομών, για διάφορες τιμές των παραμέτρων της. Η πλειοψηφία των πειραμάτων πραγματοποιήθηκε σε σύστημα αρχιτεκτονικής NUMA, με μνήμη 1TB και 40 physical cores, ενώ πραγματοποιήθηκαν και κάποια πειράματα σε πραγματικό cluster αποτελούμενο από 30 workstations.

Στις επόμενες ενότητες παρουσιάζονται δύο είδη πειραμάτων, πειράματα για τον προσδιορισμό του τρόπου που επηρεάζουν την εκτέλεση της υλοποίησης οι διάφορες παράμετροί της, και πειράματα που συγκρίνουν τη βασική υλοποίηση της κάθε φάσης με τη βελτιστοποιημένη της.

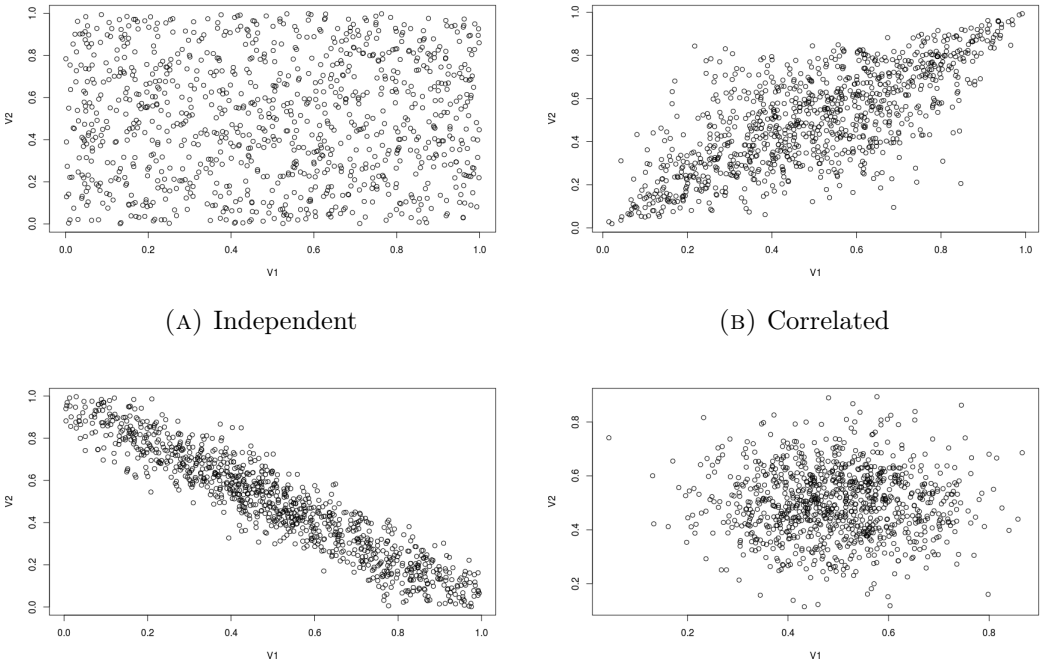
5.2 Πειράματα Παραμέτρων

Κάθε σειρά πειραμάτων που εξετάζεται σε αυτήν την ενότητα έχει στόχο να προσδιορίσει τον τρόπο που μία παράμετρος επηρεάζει την εκτέλεση του αλγορίθμου, όταν οι υπόλοιπες παραμένουν σταθερές. Οι παράμετροι αυτές είναι:

1. Ο τρόπος που παράγονται τα δεδομένα.
2. Το πλήθος των αντικειμένων του συνόλου δεδομένων $\mathcal{D}$, έστω $|\mathcal{D}|$.
3. Ο αριθμός των διαστάσεων του dataset, έστω d.
4. Η τιμή του k.
5. Ο αριθμός των κελιών του πλέγματος $\mathcal{G}$, έστω $|\mathcal{G}|$.
6. Ο αριθμός των executors που χρησιμοποιεί το Spark.

Ο τρόπος με τον οποίο αυτές οι παράμετροι μπορεί να επηρεάζουν την εκτέλεση είναι είτε επηρεάζοντας των αριθμό των σημείων που κλαδεύονται σε κάθε φάση, είτε επηρεάζοντας το χρόνο εκτέλεσης της κάθε φάσης.

Στα πειράματα αυτής της ενότητας χρησιμοποιούνται οι βελτιστοποιήσεις που προτάθηκαν για κάθε φάση. Σε κάθε σειρά πειραμάτων δοκιμάζονται διάφορες τιμές για την παράμετρο που εξετάζεται, ενώ οι υπόλοιπες παραμένουν σταθερές.



(A) Independent

(B) Correlated

(C) Anti-correlated

(D) Normally-Independent

Σχημα 5.1: Σύνολα δεδομένων παραγώμενα από διαφορετικές μεθόδους

5.2.1 Μέθοδος Παραγωγής Δεδομένων

Η πρώτη σειρά πειραμάτων εξετάζει διαφορετικούς τρόπους που παράγονται οι τιμές των χαρακτηριστικών κάθε αντικειμένου στα διάφορα σύνολα δεδομένων. Εξετάζονται τέσσερις διαφορετικές μέθοδοι:

- μέθοδος *independent*, στην οποία οι τιμές των χαρακτηριστικών ενός αντικειμένου είναι ανεξάρτητες μεταξύ τους και ακολουθούν ομοιόμορφη κατανομή.
- μέθοδος *correlated*, στην οποία ένα αντικείμενο με καλή τιμή σε κάποιο χαρακτηριστικό θα έχει αντίστοιχα καλές τιμές και στα υπόλοιπα.
- μέθοδος *anti-correlated*, στην οποία ένα αντικείμενο με καλή τιμή σε κάποιο χαρακτηριστικό θα έχει κακή τιμή σε κάποιο από ή και σε όλα τα υπόλοιπα.
- μέθοδος *normally-independent*, η οποία είναι παρόμοια με την independent, με τη διαφορά ότι οι τιμές των χαρακτηριστικών ακολουθούν κανονική κατανομή.

Στο σχήμα 5.1 φαίνονται κάποια αντιπροσωπευτικά datasets αυτών των μεθόδων για δύο διαστάσεις.

Τα πειράματα εκτελέστηκαν πάνω σε τέσσερα διαφορετικά σύνολα δεδομένων, ένα για κάθε μέθοδο. Οι υπόλοιπες παράμετροι είχαν τιμές $|\mathcal{D}| = 10^8$, $d = 3$, $k = 100$, $|\mathcal{G}| = 10^6$ και $\#ofExecutors = 16$. Τα αποτελέσματα των πειραμάτων φαίνονται στους πίνακες 5.1 και 5.2.

Από τα πειράματα συμπεραίνουμε πως η προτεινόμενη υλοποίηση εκτελείται αποδοτικότερα σε correlated δεδομένα και καθόλου αποδοτικά σε anti-correlated δεδομένα. Κάτι τέτοιο ίσως ήταν αναμενόμενο. Αυτό όμως που προκαλεί ενδιαφέρον είναι πως

	Ind	*Corr*	*Anti-corr*	*Norm-Ind*
κελιά μετά το κλάδεμα	20	7	421	172
υποψήφιοι μετά το κλάδεμα κελιών	2036	1028	436	180
υποψήφιοι μετά τη δεύτερη φάση	1138	257	436	180
μη αδιάφορα σημεία στο $\mathcal{D}$	11529366	14648	99985490	5515947

Πινακας 5.1: Στατιστικά κλαδέματος για διαφορετικές κατανομές δεδομένων

	Ind	*Corr*	*Anti-corr*	*Norm-Ind*
χρόνος φορτώματος δεδομένων	$216s$	$214s$	$211s$	$219s$
χρόνος εκτέλεσης πρώτης φάσης	$317s$	$270s$	$288s$	$259s$
χρόνος εκτέλεσης δεύτερης φάσης	$183s$	$163s$	$173s$	$176s$
χρόνος κλαδέματος αδιάφορων σημείων	$188s$	$171s$	$204s$	$175s$
χρόνος εκτέλεσης τρίτης φάσης	$3046s$	$22s$	$9375s$	$287s$
συνολικός χρόνος εκτέλεσης	$3950s$	$840s$	$10251s$	$1116s$

Πινακας 5.2: Στατιστικά χρόνου εκτέλεσης για διαφορετικές κατανομές δεδομένων

αυτή η διαφορά στην απόδοση δεν έχει τόσο να κάνει με τον αριθμό των υποψηφίων που προέκυπταν σε κάθε περίπτωση, αλλά με των αριθμό σημείων στο $\mathcal{D}$ τα οποία ήταν *αδιάφορα* και δε λήφθηκαν υπόψιν στην τρίτη φάση. Η απόδοση του αλγορίθμου σε independent δεδομένα είναι κάπου στη μέση συγκριτικά με τα άλλα δύο είδη, ενώ για normally-independent δεδομένα είναι ιδιαίτερα ικανοποιητική.

5.2.2 Παράμετρος $|\mathcal{D}|$

Στην επόμενη σειρά πειραμάτων, η παράμετρος που εξετάζεται είναι ο αριθμός των αντικειμένων $|\mathcal{D}|$ του dataset. Για τα πειράματα χρησιμοποιήθηκαν 3-διάστατα independent δεδομένα, αποτελούμενα από 10^8 ($\sim 5.5GB$) και 10^9 ($\sim 55GB$) σημεία. Όσον αφορά τις υπόλοιπες παραμέτρους, $k = 100$, $|\mathcal{G}| = 10^6$ και $\#ofExecutors = 16$. Τα αποτελέσματα των πειραμάτων φαίνονται στους πίνακες 5.3 και 5.4.

Το πρώτο πράγμα που παρατηρούμε είναι πως στο μεγαλύτερο dataset, μετά την πρώτη φάση, ο αριθμός των υποψηφίων είναι αρκετά μεγαλύτερος σε σχέση με το μικρότερο, παρόλο που το k είναι κοινό. Κάτι τέτοιο είναι απολύτως λογικό, καθώς το πλέγμα παραμένει κοινό, με αποτέλεσμα τα κελιά στο μεγαλύτερο dataset να είναι πιο πυκνοκατοικημένα, δυσκολεύοντας την εφαρμογή κανόνων κλαδέματος. Παρόλαυτα, μετά τη δεύτερη φάση, όπου το κλάδεμα γίνεται σε επίπεδο σημείων και όχι κελιών, ο αριθμός υποψηφίων στο μεγαλύτερο dataset πλησιάζει αυτόν του μικρότερου. Γενικά ο

	$\|\mathcal{D}\| = 10^8$	$\|\mathcal{D}\| = 10^9$
κελιά μετά το κλάδεμα	20	10
υποψήφιοι μετά το κλάδεμα κελιών	2036	10023
υποψήφιοι μετά τη δεύτερη φάση	1138	1995
μη αδιάφορα σημεία στο $\mathcal{D}$	11529366	87327373

Πινακας 5.3: Στατιστικά κλαδέματος για διαφορετικού μεγέθους σύνολα δεδομένων

	$\|\mathcal{D}\| = 10^8$	$\|\mathcal{D}\| = 10^9$
χρόνος φορτώματος δεδομένων	$216s$	$1586s$
χρόνος εκτέλεσης πρώτης φάσης	$317s$	$2570s$
χρόνος εκτέλεσης δεύτερης φάσης	$183s$	$2423s$
χρόνος κλαδέματος αδιάφορων σημείων	$188s$	$1880s$
χρόνος εκτέλεσης τρίτης φάσης	$3046s$	$43504s$
συνολικός χρόνος εκτέλεσης	$3950s$	$51963s$

Πινακας 5.4: Στατιστικά χρόνου εκτέλεσης για διαφορετικού μεγέθους σύνολα δεδομένων

αλγόριθμος κλιμακώνεται σχετικά ικανοποιητικά (αν και όχι γραμμικά) στο μέγεθος των δεδομένων, ακόμα και με σταθερό μέγεθος πλέγματος.

5.2.3 Παράμετρος d

Η επόμενη παράμετρος που εξετάζεται είναι ο αριθμός των διαστάσεων d. Χρησιμοποιήθηκαν 2-διάστατα, 3-διάστατα και 4-διάστατα σύνολα δεδομένων μεγέθους 10^8 σημείων και ομοιόμορφης κατανομής. Οι υπόλοιπες παράμετροι ήταν $k = 100$, $|\mathcal{G}| \simeq 10^6$ και $\#ofExecutors = 48$. Χρησιμοποιήθηκαν 48 executors αντί για 16 που χρησιμοποιούνται συνήθως γιατί, όπως θα δούμε, η εκτέλεση σε δεδομένα πολλών διαστάσεων είναι ιδιαίτερα χρονοβόρα. Τα αποτελέσματα των πειραμάτων φαίνονται στους πίνακες 5.5 και 5.6.

	$d = 2$	$d = 3$	$d = 4$
κελιά μετά το κλάδεμα	4	20	70
υποψήφιοι μετά το κλάδεμα κελιών	419	2036	6604
υποψήφιοι μετά τη δεύτερη φάση	286	1138	3869
μη αδιάφορα σημεία στο $\mathcal{D}$	498808	11529366	49317058

Πινακας 5.5: Στατιστικά κλαδέματος για διαφορετικών διαστάσεων σύνολα δεδομένων

	$d = 2$	$d = 3$	$d = 4$
χρόνος φορτώματος δεδομένων	$144s$	$166s$	$190s$
χρόνος εκτέλεσης πρώτης φάσης	$177s$	$199s$	$277s$
χρόνος εκτέλεσης δεύτερης φάσης	$95s$	$89s$	$105s$
χρόνος κλαδέματος αδιάφορων σημείων	$84s$	$85s$	$100s$
χρόνος εκτέλεσης τρίτης φάσης	$39s$	$1890s$	28619
συνολικός χρόνος εκτέλεσης	$539s$	$2429s$	29291

Πινακας 5.6: Στατιστικά χρόνου εκτέλεσης για διαφορετικών διαστάσεων σύνολα δεδομένων

Όπως φαίνεται από τα αποτελέσματα των πειραμάτων, η απόδοση του αλγορίθμου πέφτει ραγδαία καθώς μεγαλώνει ο αριθμός των διαστάσεων, και αυτό δεν οφείλεται μόνο στον αυξημένο όγκο δεδομένων. Το πρώτο πράγμα που παρατηρούμε είναι πως ο αριθμός των υποψηφίων αυξάνεται όσο αυξάνουν οι διαστάσεις. Αυτό είναι λογικό, καθώς όσο πιο πολλά είναι τα διάφορα χαρακτηριστικά ενός αντικειμένου, τόσο πιο δύσκολο γίνεται να βρεθούν k άλλα αντικείμενα καλύτερα από το αρχικό σε κάθε ένα από τα

χαρακτηριστικά, ώστε αυτό να κλαδευτεί. Επιπλέον, παρατηρούμε ότι ο αριθμός των αδιάφορων σημείων που δεν συμμετέχουν στην τρίτη φάση μειώνεται καθώς μεγαλώνει ο αριθμός των διαστάσεων. Όλα τα παραπάνω, σε συνδιασμό με το ότι το κόστος του ελέγχου κυριαρχίας είναι ανάλογο με τον αριθμό των διαστάσεων, αποτελούν την αιτία που ο αλγόριθμος δεν τα καταφέρνει καλά σε δεδομένα μεγάλων διαστάσεων.

5.2.4 Παράμετρος k

Σε αυτή τη σειρά πειραμάτων εξετάζεται η παράμετρος k. Οι τιμές που χρησιμοποιήθηκαν είναι $k = 10$, $k = 100$ και $k = 1000$, ενώ το dataset είναι 3-διάστατο, μεγέθους $|\mathcal{D}| = 10^8$ σημείων και ομοιόμορφης. Το πλέγμα αποτελείται από 10^6 κελιά, ενώ οι executors που χρησιμοποιήθηκαν ήταν 16. Τα αποτελέσματα των πειραμάτων φαίνονται στους πίνακες 5.7 και 5.8.

	$k = 10$	$k = 100$	$k = 1000$
κελιά μετά το κλάδεμα	10	20	41
υποψήφιοι μετά το κλάδεμα κελιών	1017	2036	4155
υποψήφιοι μετά τη δεύτερη φάση	185	1138	3976
μη αδιάφορα σημεία στο $\mathcal{D}$	8736965	11529366	14266468

Πινακας 5.7: Στατιστικά κλαδέματος για διαφορετικές τιμές του k

	$k = 10$	$k = 100$	$k = 1000$
χρόνος φορτώματος δεδομένων	$209s$	$216s$	$221s$
χρόνος εκτέλεσης πρώτης φάσης	$315s$	$317s$	$320s$
χρόνος εκτέλεσης δεύτερης φάσης	$165s$	$183s$	$187s$
χρόνος κλαδέματος αδιάφορων σημείων	$184s$	$188s$	$188s$
χρόνος εκτέλεσης τρίτης φάσης	$445s$	$3046s$	$13517s$
συνολικός χρόνος εκτέλεσης	$1318s$	$3950s$	$14533s$

Πινακας 5.8: Στατιστικά χρόνου εκτέλεσης για διαφορετικές τιμές του k

Τα συμπεράσματα που προκύπτουν από τα αποτελέσματα των πειραμάτων είναι πως καθώς αυξάνεται το k, αυξάνεται και ο χρόνος εκτέλεσης. Αυτό οφείλεται στο ότι όσο μεγαλύτερο είναι το k, τόσο περισσότεροι θα είναι και οι υποψήφιοι, κάτι πολύ λογικό και αναμενόμενο. Επιπλέον, λόγω των περισσότερων υποψηφίων, μειώνεται ελαφρώς και ο αριθμός των αδιάφορων σημείων που μπορούν να κλαδευτούν, κάτι που αυξάνει περαιτέρω τους υπολογισμούς που πρέπει να γίνουν κατά την τρίτη φάση. Αυτό που είναι ιδιαίτερα ενδιαφέρον είναι πως καθώς αυξάνεται το k, μειώνεται η αποτελεσματικότητα της δεύτερης φάσης.

5.2.5 Παράμετρος $|\mathcal{G}|$

Στη συνέχεια εξετάζεται ο αριθμός των κελιών $|\mathcal{G}|$ του πλέγματος. Εξετάστηκαν πλέγματα αποτελούμενα από 10^5, 10^6 και 10^7 κελία, τα οποία εφαρμόστηκαν πάνω σε 3-διάστατο independent σύνολο δεδομένων με 10^8 σημεία, για $k = 100$ και με 16 executors. Τα αποτελέσματα των πειραμάτων φαίνονται στα σχήματα 5.9 και 5.10.

Βάσει των αποτελεσμάτων, ο αριθμός των κελιών του πλέγματος παρουσιάζει ιδιαίτερο ενδιαφέρον. Καθώς αυτός αυξάνεται, μειώνεται ο αριθμός των υποψηφίων, και κατ'

	$\|\mathcal{G}\| = 10^5$	$\|\mathcal{G}\| = 10^6$	$\|\mathcal{G}\| = 10^7$
κελιά μετά το κλάδεμα	10	20	57
υποψήφιοι μετά το κλάδεμα κελιών	9778	2036	567
υποψήφιοι μετά τη δεύτερη φάση	1975	1138	539
μη αδιάφορα σημεία στο $\mathcal{D}$	17954349	11529366	8106727

Πινακας 5.9: Στατιστικά κλαδέματος για διαφορετικά πλέγματα

	$\|\mathcal{G}\| = 10^5$	$\|\mathcal{G}\| = 10^6$	$\|\mathcal{G}\| = 10^7$
χρόνος φορτώματος δεδομένων	$224s$	$216s$	$217s$
χρόνος εκτέλεσης πρώτης φάσης	$188s$	$317s$	$1249s$
χρόνος εκτέλεσης δεύτερης φάσης	$197s$	$183s$	$119s$
χρόνος κλαδέματος αδιάφορων σημείων	$184s$	$188s$	$131s$
χρόνος εκτέλεσης τρίτης φάσης	$11982s$	$3046s$	$899s$
συνολικός χρόνος εκτέλεσης	$12775s$	$3950s$	$2615s$

Πινακας 5.10: Στατιστικά χρόνου εκτέλεσης για διαφορετικά πλέγματα

επέκταση αυξάνεται ο αριθμός των αδιάφορων σημείων που μπορούν να κλαδευτούν. Έτσι, μειώνεται ο αριθμός των συνολικών υπολογισμών που πρέπει να γίνουν κατά την τρίτη φάση, κάτι που φαίνεται και από τους χρόνους εκτέλεσης της. Παρόλο όμως που μεγαλύτερα πλέγματα επιταγχύνουν την τρίτη φάση, ο επιπλέον αριθμός κελιών σημαίνει περισσότεροι υπολογισμοί άνω/κάτω φραγμάτων κατά την πρώτη, οι οποίοι να θυμήσουμε γίνονται τοπικά στον driver. Αυτό σημαίνει πως το μέγεθος του πλέγματος θα πρέπει να επιλέγεται με προσοχή, καθώς είναι δυνατόν ο χρόνος εκτέλεσης της πρώτης φάσης να είναι τέτοιος που να αναιρεί τα οφέλη που έχει ένα μεγαλύτερο πλέγμα στην τρίτη φάση. Επιπλέον, πρέπει να λαμβάνεται υπόψιν ότι επειδή οι υπολογισμοί κατά την πρώτη φάση γίνονται τοπικά, υπάρχει και περιορισμός μνήμης.

5.2.6 Αριθμός Executors

Στην τελευταία σειρά πειραμάτων εξετάζεται πως ο αριθμός executors που γίνονται διαθέσιμοι στο Spark επηρεάζει τους χρόνους εκτέλεσης κάθε φάσης του αλγορίθμου. Στην ουσία πρόκειται για ένα πείραμα με 3-διάστατο independent σύνολο δεδομένων από 10^8 σημεία, με $k = 1000$ και $|\mathcal{G}| = 10^6$, το οποίο επαναλαμβάνεται με 2, 4 και 8 workers nodes. Κάθε worker διαθέτει 4 cores και εκτελεί έναν executor στον καθένα, επομένως έχουμε 8, 16 και 32 executors. Τα αποτελέσματα φαίνονται στον πίνακα 5.11.

	8 executors	16 executors	32 executors
χρόνος φορτώματος δεδομένων	$312s$	$221s$	$162s$
χρόνος εκτέλεσης πρώτης φάσης	$458s$	$320s$	$222s$
χρόνος εκτέλεσης δεύτερης φάσης	$373s$	$187s$	$108s$
χρόνος κλαδέματος αδιάφορων σημείων	$299s$	$188s$	$107s$
χρόνος εκτέλεσης τρίτης φάσης	$28381s$	$13517s$	$9277s$
συνολικός χρόνος εκτέλεσης	$29823s$	$14533s$	$9876s$

Πινακας 5.11: Στατιστικά χρόνου εκτέλεσης για διαφορετικό αριθμό executors

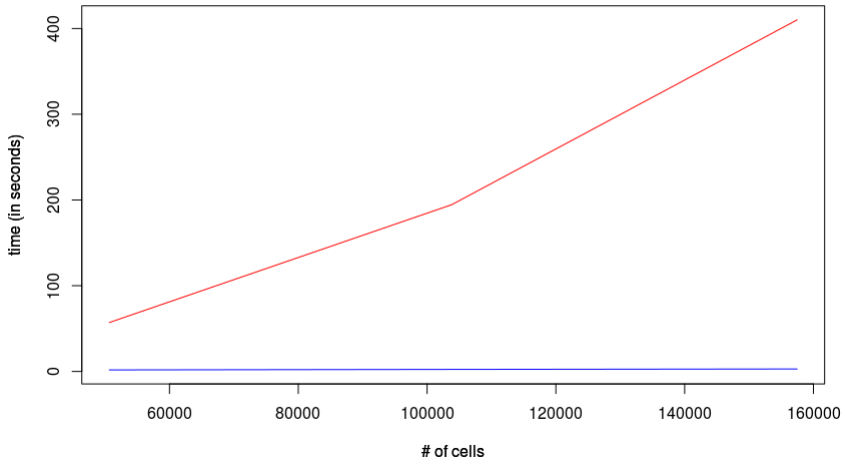
Από τα αποτελέσματα φαίνεται πως ο προτεινόμενος αλγόριθμος κλιμακώνεται αρκετά καλά σε κατανεμημένο περιβάλλον. Αυτό φαίνεται καλύτερα στους χρόνους εκτέλεσης της τρίτης φάσης, η οποία εκτελείται κατά κύριο λόγο στους executors.

5.3 Πειράματα Βελτιστοποιήσεων

Τα πειράματα αυτής της ενότητας συγκρίνουν τη βασική υλοποίηση της κάθε φάσης με τη βελτιστοποιημένη της.

5.3.1 1η Φάση

Η βελτιστοποίηση που προτάθηκε για την πρώτη φάση έχει να κάνει με τον υπολογισμό άνω/κάτω φραγμάτων των κελιών του πλέγματος. Οι παράμετροι που επηρεάζουν τους υπολογισμούς αυτούς, είναι το πλήθος των κελιών και ο αριθμός διαστάσεων του πλέγματος. Για το λόγο αυτό πραγματοποιήθηκαν δύο σειρές πειραμάτων που συγκρίνουν τους χρόνους εκτέλεσης της βασικής και τις βελτιστοποιημένης υλοποίησης, μία για κάθε παράμετρο. Τα αποτελέσματα των πειραμάτων για την παράμετρο $|\mathcal{G}|$ φαίνονται στο σχήμα 5.2 και για την παράμετρο d στο σχήμα 5.3 (η μπλε γραμμή αντιπροσωπεύει τη βελτιστοποίηση).

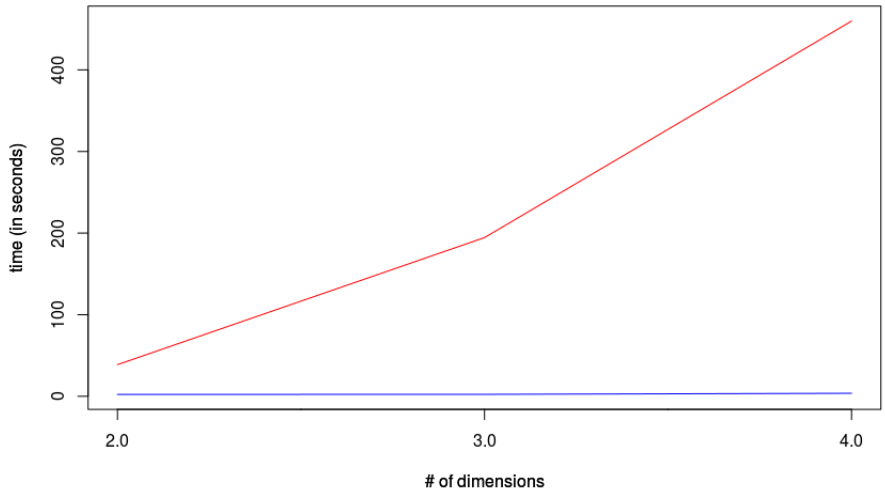


Σχημα 5.2: Απόδοση υπολογισμού φραγμάτων ως προς τον αριθμό κελιών

Όπως φαίνεται και από τα αποτελέσματα, η προτεινόμενη βελτιστοποίηση αξιοποιώντας την αρχή έγκλεισης-απόκλεισης, είναι κατά πολύ ανώτερη της βασικής υλοποίησης για τον υπολογισμό άνω/κάτω φραγμάτων των κελιών, και κλιμακώνεται καλύτερα τόσο για αυξημένο αριθμό κελιών όσο και για αυξημένο αριθμό διαστάσεων.

5.3.2 2η Φάση

Η βελτιστοποίηση που προτάθηκε για τη δεύτερη φάση πραγματοποιεί το κλάδεμα σημείων για την εξαγωγή υποψηφιών στους executors, σε αντίθεση με τη βασική υλοποίηση που το πραγματοποιεί τοπικά στον driver. Το μειονέκτημα που έχει η βελτιστοποίση συγκριτικά με τη βασική υλοποίηση, είναι πως απαιτεί μία επιπλέον αναμετάδοση δεδομένων στους executors. Υπάρχει λοιπόν ένα επιπλέον κόστος κατά τη βελτιστοποιημένη

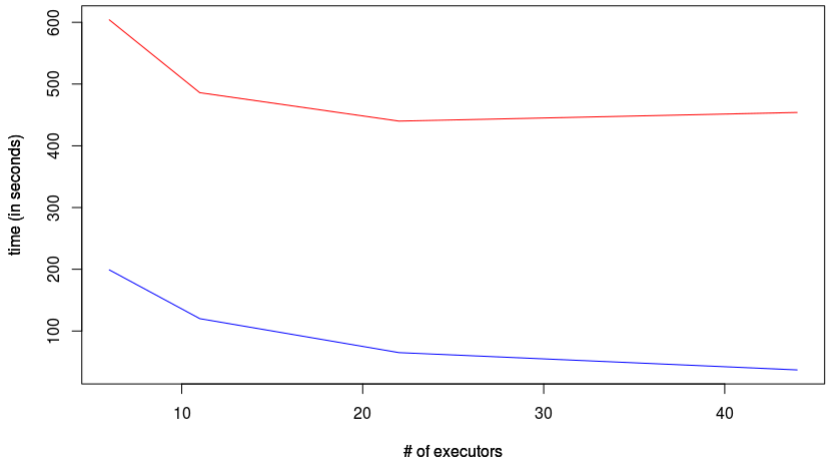


Σχημα 5.3: Απόδοση υπολογισμού φραγμάτων ως προς τον αριθμό διαστάσεων

εκδοχή της δεύτερης φάσης, το οποίο μένει να δούμε πόσο μικρό ή μεγάλο είναι σε σχέση με την επιτάγχυνση που θα επιφέρει ο κατανεμημένος υπολογισμός των υποψηφίων.

Οι δύο παράγοντες που κρίνουν την απόδοση των δύο εκδοχών είναι πρώτον ο αριθμός των υποψηφίων μετά το κλάδεμα των κελιών, και δεύτερον ο αριθμός των διαθέσιμων executors. Για αυτό το λόγο, πραγματοποιήθηκαν δύο σειρές πειραμάτων, με καθεμία να εξετάζει πως οι διάφορες τιμές της αντίστοιχης παραμέτρου επηρεάζουν τους χρόνους εκτέλεσης των δύο εκδοχών. Επειδή μας ενδιαφέρει το κόστος της επιπλέον αναμετάδοσης δεδομένων, τα πειράματα αυτά πραγματοποιήθηκαν σε πραγματικό cluster. Για τα πειράματα χρησιμοποιήθηκε ένα 3-διάστατο independent σύνολο δεδομένων από 10^8 σημεία (44 partitions), με πλέγμα μεγέθους $|\mathcal{G}| = 10^6$. Επειδή ο αριθμός των σημείων μετά το κλάδεμα των κελιών δεν είναι μια παράμετρος που ορίζεται από το χρήστη, στα πειράματα που μελετούν την παράμετρο αυτή έγινε έμμεσος καθορισμός της μέσω διαφορετικών τιμών του k. Τα αποτελέσματα των πειραμάτων φαίνονται στα σχήματα 5.4 και 5.5 (με μπλε γραμμή είναι η βελτιστοποιημένη εκδοχή).

Από τα αποτελέσματα των πειραμάτων προκύπτει πως τελικά το κόστος της επιπλέον αναμετάδοσης είναι αμελητέο σε σχέση με την επιτάγχυνση που επιφέρει ο υπολογισμός των υποψηφίων στους executors έναντι του driver. Συγκεκριμένα βλέπουμε πως όσο περισσότερα είναι τα σημεία από τα οποία θα προκύψουν οι υποψήφιοι, τόσο πιο χρονοβόρος είναι ο υπολογισμός αυτών τοπικά στον driver, ενώ παρατηρούμε πως ακόμα και για μικρό αριθμό executors η βελτιστοποίηση υπερτερεί. Γενικά η βελτιστοποιημένη εκδοχή, όπως φαίνεται και από τα αποτελέσματα των πειραμάτων, είναι προτιμότερη από τη βασική υλοποίηση της δεύτερης φάσης, καθώς οδηγεί σε ταχύτερους χρόνους εκτέλεσης. Πρέπει να σημειωθεί βέβαια πως γενικά η δεύτερη φάση δεν αποσπά μεγάλο ποσοστό του συνολικού χρόνου εκτέλεσης, και έτσι αυτό το κέρδος στο χρόνο εκτέλεσης που φέρνει η βελτιστοποίηση θα μπορούσε να χαρακτηριστεί μικρό.



Σχημα 5.4: Σύγκριση των χρόνων εκτέλεσης των δύο εκδοχών της 2ης φάσης ως προς τον αριθμό των executors

5.3.3 3η Φάση

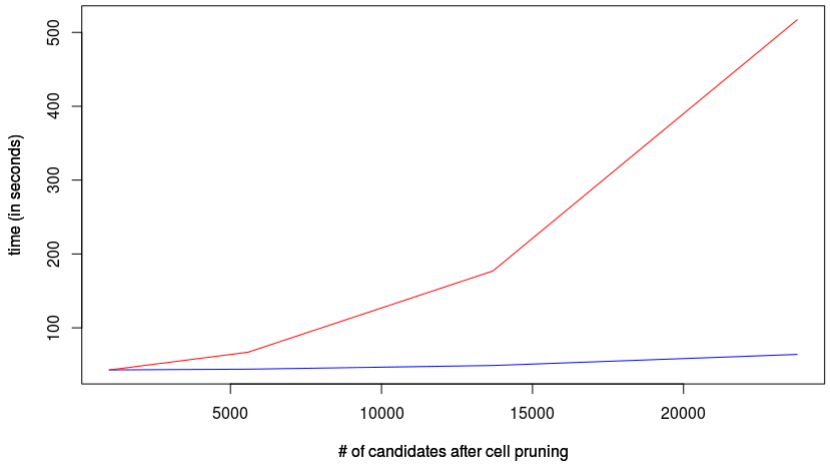
Η βελτιστοποίηση που προτάθηκε για την τρίτη φάση είναι ένα φιλτράρισμα των αδιάφορων σημείων του $\mathcal{D}$, τα οποία δεν απαιτούνται για τον υπολογισμό του σκορ κυριαρχίας των υποψηφίων. Αυτό το φιλτράρισμα απαιτεί ένα πέρασμα των υποψηφίων στον driver για τον προσδιορισμό των αδιάφορων κελιών, και ένα πέρασμα του $\mathcal{D}$, το οποίο γίνεται παράλληλα στους executors, για το κλάδεμα των σημείων που ανήκουν στα κελιά αυτά.

Μελετώντας τα αποτελέσματα των πειραμάτων που παρουσιάστηκαν στην προηγούμενη ενότητα, είναι προφανές πως αυτό το επιπλέον φιλτράρισμα στις περισσότερες περιπτώσεις κλαδεύει ένα πολύ μεγάλο ποσοστό του $\mathcal{D}$, μειώνοντας δραστικά το χρόνο που απαιτείται για τον υπολογισμό του σκορ των υποψηφίων, ενώ το ίδιο πραγματοποιείται σε σχετικά μικρό χρονικό διάστημα. Από όλες τις παραμέτρους που μελετήθηκαν στην προηγούμενη ενότητα, αυτή η οποία επηρεάζει σε μεγαλύτερο βαθμό την απόδοση του φιλτραρίσματος ήταν η κατανομή των δεδομένων. Για αυτό το λόγο, έγινε μια σειρά πειραμάτων ($|\mathcal{D}| = 10^8$, $d = 3$, $k = 100$, $|\mathcal{G}| = 10^6$, 16 executors) που συγκρίνει τους χρόνους εκτέλεσης της τρίτης φάσης με και χωρίς το φιλτράρισμα, για δεδομένα κάθε κατανομής (πίνακας 5.12).

	αδιάφορα σημεία	χρόνος βασικής υλοποίησης	χρόνος βελτ. υλοποίησης
Ind	88470634	24582s	3234s
Corr	99985352	6241s	193s
Anti	14510	9366s	9579s
Norm	94484053	4327s	462s

Πινακας 5.12: Στατιστικά εκτέλεσης των δύο εκδοχών της τρίτης φάσης

Τα anti-correlated δεδομένα είναι ίσως η μόνη περίπτωση στην οποία δε συμφέρει η εφαρμογή του φιλτραρίσματος, καθώς ο αριθμός των αδιάφορων σημείων είναι τέτοιος που δε δικαιολογεί επιπλέον περάσματα για το κλάδεμα τους. Ακόμα και σε αυτήν την περίπτωση όμως, η διαφορά των χρόνων εκτέλεσης της βασικής και της βελτιστοποιημένης εκδοχής της τρίτης φάσης είναι σχεδόν αμελητέα. Γενικά, με την εξαίρεση των



Σχημα 5.5: Σύγκριση των χρόνων εκτέλεσης των δύο εκδοχών της 2ης φάσης ως προς τον αριθμό των σημείων μετά το κλάδεμα κελιών

anti-correlated δεδομένων, το κλάδεμα των αδιάφορων σημείων είναι μία προσθήκη που συμβάλει σημαντικά στην επιτάχυνση της εκτέλεσης ερωτημάτων κυριαρχίας.

5.4 Επίλογος και Σύνοψη

Από την εκτέλεση των πειραμάτων προέκυψαν πολλά χρήσιμα συμπεράσματα σχετικά με την προτεινόμενη υλοποίηση. Το σημαντικότερο ίσως είναι πως οι τεχνικές κλαδέματος που χρησιμοποιούνται παίζουν καταλυτικό ρόσο στη γρήγορη εκτέλεση ερωτημάτων top-k κυριαρχίας. Αυτό γίνεται ιδιαίτερα εμφανές σε περιπτώσεις όπως αυτή των δεδομένων μεγάλων διαστάσεων, όπου λόγω μικρότερης αποτελεσματικότητας των τεχνικών αυτών αυξάνεται κατά πολύ ο συνόλικός χρόνος εκτέλεσης.

Πέραν όμως των παραπάνω, διαπιστώθηκε πως η προτεινόμενη υλοποίηση κλιμακώνεται αρκετά καλά σε μεγάλο αριθμό από executors, αλλά δεν συμπεριφέρεται το ίδιο σε όλες τις κατανομές των δεδομένων. Επιπλέον, θα πρέπει να γίνεται προσεκτική επιλογή του μεγέθους του πλέγματος, καθώς υπάρχει ένα tradeoff μεταξύ του αριθμού υποψηφίων και του χρόνου εκτέλεσης της πρώτης φάσης.

Τέλος, οι βελτιστοποιήσεις της πρώτης και τρίτης φάσης κρίνονται απαραίτητες για την αποφυγή bottlenecks κατά την εκτέλεση ερωτημάτων, ενώ αυτή της δεύτερης είναι προτιμότερη από τη βασική σε περιπτώσεις όπου είναι διαθέσιμος μεγάλος αριθμός executors ή/και μένει μεγάλος αριθμός υποψηφίων μετά το κλάδεμα κελιών.

Κεφάλαιο 6

Επίλογος

Στόχος της παρούσας εργασίας ήταν η αποδοτική επεξεργασία ερωτημάτων top-k κυριαρχίας πάνω σε μεγάλα δεδομένα, αξιοποιώντας τη δύναμη του παραλληλισμού που προσφέρει η πλατφόρμα Apache Spark. Μπορούμε να πούμε πως σε ένα μεγάλο βαθμό αυτός ο στόχος επετεύχθει.

Στηριζόμενοι σε έναν υπάρχων σειριακό αλγόριθμο, συνδιάσαμε τις βασικές του αρχές με τις λειτουργίες που προσφέρει το Spark, ώστε να προκύψει μία βασική κατανεμημένη υλοποίηση επεξεργασίας ερωτημάτων top-k κυριαρχίας. Στη συνέχεια, έγιναν περαιτέρω βελτιώσεις σε σημεία τα οποία αποτελούσαν bottlenecks κατά την εκτέλεση, με τις οποίες επιτυγχάνεται σημαντική μείωση του συνολικού χρόνου εκτέλεσης.

Η δύναμη του προτεινόμενου αλγορίθμου, πέραν από το κατανεμημένο κομμάτι, βρίσκεται στις τεχνικές κλαδέματος που χρησιμοποιούνται, οι οποίες μειώνουν σημαντικά τους υπολογισμούς που απαιτούνται και κατ΄ επέκταση το υπολογιστικό κόστος. Επιβεβαίωση σε αυτόν τον ισχυρισμό αποτελεί το γεγονός πως σε περιπτώσεις όπως τα anticorrelated ή τα πολυδιάστατα δεδομένα, στα οποία οι τεχνικές κλαδέματος που χρησιμοποιούνται δεν είναι ιδιαίτερα αποτελεσματικές, ο συνολικός χρόνος εκτέλεσης ήταν πολύ μεγαλύτερος.

Στο μέλλον θα γίνει προσπάθεια αποδοτικότερου κλαδέματος, ιδιαίτερα στις περιπτώσεις που αναφέρθηκαν, για ακόμα καλύτερες επιδόσεις. Πιθανό σημείο εκκίνησης θα μπορούσε να είναι η εφαρμογή μη ομοιόμορφου πλέγματος στα δεδομένα. Γενικά, όσο περισσότερα κελιά τόσο το καλύτερο, όμως ο αριθμός κελιών του πλέγματος περιορίζεται από τη μνήμη του driver, καθώς για κάθε κελί θα πρέπει να διατηρούνται άνω/κάτω φράγματα, κ.α. Ίσως η εφαρμογή περισσότερων μικρών κελιών σε χώρους με πολλά σημεία και λιγότερων, πιο μεγάλων κελιών σε χώρους με ελάχιστα σημεία, διατηρώντας το συνολικό αριθμό κελιών σταθερό, να είναι η απάντηση που ψάχνουμε.

Βιβλιογραφία

[1] Man Lung Yiu, Nikos Mamoulis: Multi-dimensional top-k dominating queries. In: VLDB (2009)

[2] Börzsönyi S., Kossmann D., Stocker K.: The skyline operator. In: ICDE (2001)

[3] Fagin R., Lotem A., Naor M.: Optimal aggregation algorithms for middleware. In: PODS (2001)

[4] Hristidis V., Koudas N., Papakonstantinou Y.: PREFER: a system for the efficient execution of multiparametric ranked queries. In: SIGMOD (2001)

[5] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia: Learning Spark, Lighting-Fast Data Analysis. O'Reilly 2015

[6] Apache Spark Website

[7] Apache Hadoop Website

[8] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI (2004)

[9] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler: The Hadoop Distributed File System. In: MSST (2010)

[10] Apache Cassandra Website

[11] Apache Hive Website

[12] Vavilapalli et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: SoCC (2013)

[13] Benjamin Hindman et al.: Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI (2011)

[14] Databricks