Πανεπιστημιο Πατρων
Πολυτεχνικη Σχολη
Τμημα Μηχανικων Ηλεκτρονικων Υπολογιστων και Πληροφορικης

Αποδοτική επεξεργασία χωρικών ερωτημάτων σε περιβάλλον SPARK με χρήση ευρετηρίου R-Tree

Διπλωματικη Εργασια

του

Ξενοφων Βουραιωτη

Επιβλέπων: Σπύρος Σιούτας
Καθηγητής

Πάτρα, Αύγουστος 2021

Ευχαριστίες

Θα ήθελα καταρχάς να ευχαριστήσω τον καθηγητή κ. Σπύρο Σιούτα για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να εμβαθύνω στον τομέα των Πολυδιάστατων Δομών Δεδομένων. Επίσης ευχαριστώ ιδιαίτερα τον κ. Κώστα Τσίχλα για την καθοδήγησή του. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Πάτρα, Αύγουστος 2021
Ξενοφών Βουρλιώτης

στην οικογένειά μου

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

1.1 Το πρόβλημα και η σημασία του

Ζούμε στην εποχή όπου η έννοια του Big Data είναι πλέον ευρέως γνωστή. Υπολογίζεται ότι το 2020, τα δεδομένα τα οποία δημιουργήθηκαν, καταγράφηκαν, αντιγράφηκαν και καταναλώθηκαν ξεπερνούν τα 44 Zettabytes [1], εκ των οποίων, το 90% παράχθηκαν τα τελευταία δύο χρόνια. Με τον συνεχόμενο αυξανόμενο ρυθμό παραγωγής δεδομένων γίνεται όλο και πιο σημαντική η ανάγκη για την αποδοτικότερη αποθήκευση και αναζήτηση αυτών. Η συνεχώς αυξανόμενη ζήτηση για υπολογισμούς χωρικών δεδομένων στην εποχή των μεγάλων δεδομένων απαιτεί πλέον, την επιλογή των αποδοτικότερων χωρικών ευρετηρίων ανάλογα με τις ανάγκες του χρήστη. Για αυτόν τον λόγο έχουν αναπτυχθεί τεχνολογίες οι οποίες καθιστούν την αντιμετώπιση αυτού του προβλήματος πιο εύκολες από ότι ήταν στο παρελθόν. Με την ανάπτυξη του μοντέλου MapReduce και των κατανεμημένων περιβάλλων υπολογισμού η επεξεργασία μεγάλου όγκου δεδομένων γίνεται όλο και πιο εύκολη.

1.2 Στόχοι διπλωματικής εργασίας

Στόχος της διπλωματικής εργασίας είναι η υλοποίηση του ευρετήριου R-Tree σε γλώσσα Python και η μετατροπή αυτού στο κατανεμημένο περιβάλλον Spark, ώστε να συγκρίνουμε το ποσοστό βελτίωσής του σχετικά με διάφορες παραμέτρους, όπως είναι ο όγκος των δεδομένων, ο τύπος των δεδομένων και ο τύπος των αναζητήσεων.

1.3 Συνεισφορά διπλωματικής εργασίας

Η συνεισφορά της διπλωματικής εργασίας είναι η υλοποίηση του ευρετηρίου R-Tree στο κατανεμημένο περιβάλλον Spark και η πειραματική αξιολόγηση των λειτουργιών του σε σχέση με μία τοπική υλοποίησή του. Η υλοποίηση του R-Tree περιγράφεται στο κεφάλαιο 3 και η υλοποίηση του σε κατανεμημένο περιβάλλον Spark στο κεφάλαιο 5. Επιπλέον των βασικών λειτουργιών (ένθεση, αναζήτηση στοιχείων), εξετάζεται και αυτή της αναζητησης κορυφογραμμής. Στόχος των πειραμάτων ήταν ο χρόνος ανταπόκρισης και αξιολόγηση κλιμάκωσης

του δέντρου. Τα αποτελέσματα και των δύο πειραματων φαίνονται στο κεφάλαιο 6 που φαίνεται η υπεροχή του αλγορίθμου στο Spark.

1.4 Οργάνωση του τόμου

Η εργασία αυτή ειναι οργανωμένη σε 7 κεφάλαια: Στο κεφάλαιο 2 δίνονται οι βασικες έννοιες των βασικών τεχνολογιών που σχετίζονται με τη διπλωματική αυτή. Αρχικά περιγράφονται τα γραφήματα και δέντρα, στην συνέχεια το R-Tree με τις βασικες λειτουργίες του, και τέλος το μοντέλο MapReduce. Στο κεφάλαιο 3 περιγράφεται η υλοποίηση των λειτουργιών του R-Tree στην γλώσσα προγραμματισμού Python. Στο κεφαλαιο 4 παρουσιάζεται το κατανεμημενο περιβάλλον Spark μαζί με κάποιες βασικές έννοιες και τεχνολογίες γύρω από αυτό. Στο κεφάλαιο 5 περιγράφεται η μετατροπή και η υλοποίηση του R-Tree στο περιβάλλον Spark. Στο κεφάλαιο 6 γίνεται η σύγκριση ανάμεσα στους δύο αλγορίθμους. Τέλος, στο κεφάλαιο 7, παραθέτονται τα συμπεράσματα της σύγκρισης που εξάγονται.

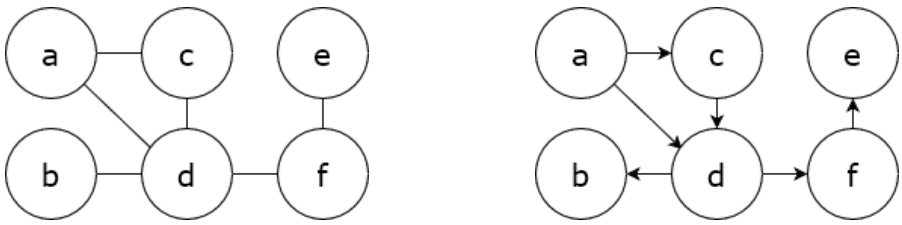
Κεφάλαιο 2

Βασικές Έννοιες

Σε αυτό το κεφάλαιο, θα αναφερθούν οι βασικές έννοιες που είναι σημαντικές για το θεωρητικό υπόβαθρο της διπλωματικής εργασίας. Συγκεκριμένα, θα αναλυθούν οι έννοιες των γραφημάτων και δέντρων, του R-Tree, και τέλος του μοντέλου MapReduce.

2.1 Γραφήματα και δέντρα

Ένα γράφημα θεωρείται η πιο γενική μορφή δομής δεδομένων, με την έννοια ότι όλες οι υπόλοιπες δομές δεδομένων μπορούν να θεωρηθούν περιπτώσεις γράφων. Ένα γράφημα αποτελείται από ένα σύνολο V (Vertices) κορυφών ή κόμβων, από ένα σύνολο E (Edges) ακμών και συμβολίζεται ώς $G = (V, E)$. Αν το σύνολο των ακμών E είναι διατεταγμένα ζεύγη κόμβων, τότε το γράφημα είναι κατευθυνόμενο και οι ακμές συμβολίζονται ως (a, b). Μη-κατευθυνόμενο γράφημα είναι εκείνο που τα ζεύγη κόμβων δεν είναι διατεταγμένα και οι ακμές συμβολίζονται ως a, b και υποδηλώνει ότι η φορά της ακμής είναι από το a στο b. Παράδειγμα τέτοιων γραφημάτων παρουσιάζετε στο σχήμα 2.1.

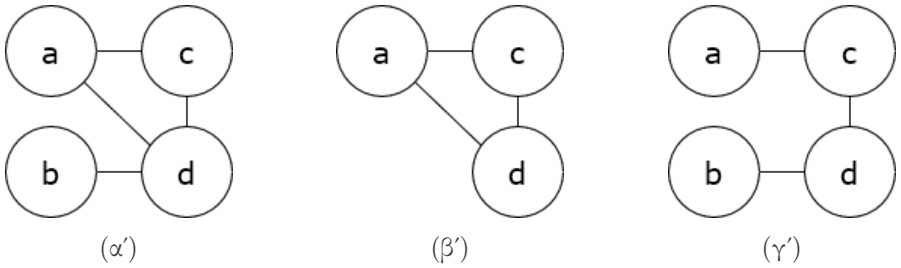


(α′) (β′)

Σχήμα 2.1: *(α′) Μη-κατευθυνόμενο γράφημα, (β′) Κατευθυνόμενο γράφημα*

Ο αριθμός των κορυφών ή κόμβων του γράφου ονομάζεται τάξη του γράφου και συμβολίζεται ως $n = |V|$, ενώ ο αριθμός των ακμών του γράφου ονομάζεται μέγεθος του γράφου και συμβολίζεται ως $m = |E|$.

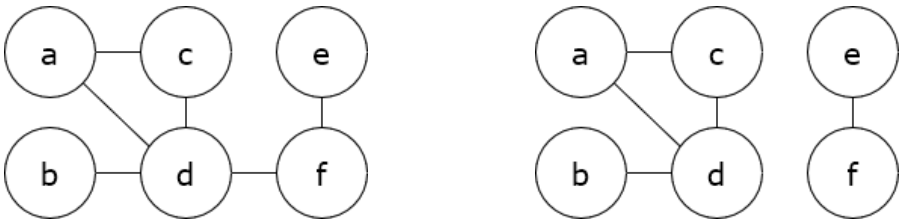
Ένα γράφημα $G' = (V', E')$, είναι υπογράφημα του γραφήματος $G = (V, E)$, αν $V' \subseteq V$ και $E' \subseteq E$. Στην περίπτωση όπου $V' = V$ και $E' \subseteq E$, τότε ονομάζεται γεννητικό υπογράφημα του G. Το σχήμα 2.2 δίνει ένα παράδειγμα για αυτούς τους όρους.



(α′) (β′) (γ′)

Σχήμα 2.2: *(α′) Γράφημα G, (β′) Υπογράφημα του G, (γ′) Γεννητικό υπογράφημα του G*

Διαδρομή (path) σε ένα γράφημα $G = (V, E)$ καλείται μία διάταξη κόμβων οι οποίοι ενώνονται με ακμές. Δύο κόμβοι a και b είναι συνδεδεμένοι αν το G περιέχει τουλάχιστον μία διαδρομή από το a στο b ή το αντίστροφο. Ένα γράφημα ονομάζεται συνεκτικό όταν κάθε ζεύγος κόμβων είναι συνδεδεμένο, αλλιώς ονομάζεται μη συνεκτικό. Κύκλος καλείται η διαδρομή δύο ή περισσοτέρων κόμβων που καταλήγει στον κόμβο αρχής του γραφήματος G. Το σχήμα 2.3 παρουσιάζει παράδειγμα για τους παραπάνω όρους.



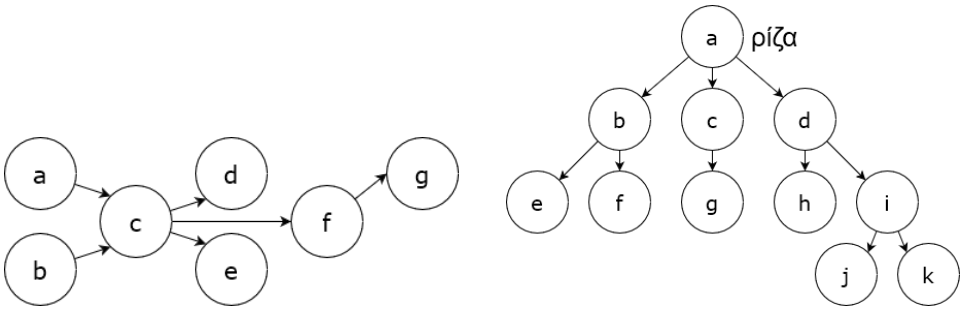
(α′) (β′)

Σχήμα 2.3: *(α′) Συνεκτικό γράφημα, (β′) Μη-συνεκτικό γράφημα*

Συνεκτικά γραφήματα τα οποία δεν περιέχουν κύκλους ονομάζονται **δέντρα**. Στην επιστήμη των υπολογιστών, τα δέντρα χρησιμοποιούνται ως δομές δεδομένων. Κάποιες από τις βασικές ιδιότητες ενός γραφήματος G το οποίο είναι δέντρο είναι οι εξής:

- Υπάρχει μοναδική διαδρομή μεταξύ κάθε ζεύγους κόμβων του γραφήματος G.
- Το γράφημα G είναι συνεκτικό.
- Το γράφημα G είναι άκυκλο.
- Το γράφημα G έχει n-1 ακμές.
- Όταν ενωθούν δύο τυχαίοι μη γειτονικοί κόμβοι του γραφήματος G με ακμή, το γράφημα G' που θα προκύψει έχει ένα μοναδικό κύκλο.

Ένα δέντρο ονομάζεται ριζωμένο όταν κάποιος διακεκριμένος κόμβος έχει οριστεί ως αφετηρία ή ρίζα. Στα ριζωμένα δέντρα σε κάθε ακμή του καλούμε τον κόμβο που βρίσκεται πλησιέστερα στην ρίζα γονέα και τον κόμβο που βρίσκεται μακρύτερα παιδί. Κάθε παιδί έχει μοναδικό πατέρα ενώ το αντίστροφο δεν ισχύει πάντα. Ως επίπεδο κάθε κόμβου ορίζεται ο αριθμός των ακμών του μονοπατιού από την ρίζα εών τον κόμβο. Στο σχήμα 2.4 παρουσιάζεται ένα μη ριζωμένο και ένα ριζωμένο δέντρο.



(α′) (β′)

Σχήμα 2.4: *(α′) Μη-ριζωμένο δέντρο, (β′) Ριζωμένο δέντρο*

2.2 R-Tree

Το R-Tree [2] είναι ένα ισοζυγισμένο δέντρο, παρόμοιο με το B^+-Tree, το οποίο χρησιμοποιείται κυρίως ως ευρετήριο πολυδιάστατων δομών δεδομένων όπως συντεταγμένες, ορθογώνια ή πολύγωνα. Αρχικά παρουσιάστηκε από τον Guttman το 1984 και έχει βρει σημαντική χρήση τόσο σε θεωρητικά όσο και σε εφαρμοσμένα πλαίσια. Η βασική ιδέα του R-Tree, είναι η ομαδοποίηση των κοντινότερων μεταξύ τους δεδομένων σε ένα ελάχιστο περιγεγραμμένο ορθογώνιο του χώρου αναζήτησης ή αλλιώς Minimum Bounding Rectangle (MBR) και η αναπαράσταση τους μέσω του MRB στο επόμενο υψηλότερο επίπεδο του δέντρου. Έτσι, τα MBR κάθε εσωτερικού κόμβου περιέχουν τα αντίστοιχα MBR, ενώ τα φύλα περιέχουν τις εγγραφές τους ευρετηρίου.

2.2.1 Δομή R-Tree

Έστω ότι έχουμε ένα σύνολο S από αντικείμενα στο $\mathbb{R}^2$. Τα αντικείμενα μπορεί να είναι πολύγωνα, γραμμές και σημεία. Όλα αυτά τα αντικείμενα, τα αντιπροσωπεύουν τα MBR τους. Το ευρετήριο στην συνέχεια κατασκευάζετε πάνω σε αυτά τα MBR. Με αυτήν την λογική, θεωρούμε πλέον τα αντικείμενα ως ορθογώνια στο $\mathbb{R}^2$. Στην πραγματικότητα τα αντικείμενα μπορούν να είναι πιο πολύπλοκα, και να ανήκουν σε μεγαλύτερες διαστάσεις. Σε αυτήν την περίπτωση λέμε ότι τα αντικείμενα τα αντιπροσωπεύουν τα Minimum Bounding Region τους όπου ανήκουν στην ίδια διάσταση με αυτά. Στην περίπτωση που εξετάζουμε τα φύλλα του δέντρου έχουν την μορφή (MBR, object_id), ενώ οι εσωτερικοί κόμβοι την μορφή (MBR, child_node_id), όπου το MBR των εσωτερικών κόμβων είναι το ορθογώνιο που καλύπτει όλα τα στοιχεία που έχουν ως ρίζα τον κόμβο που δείχνει το child_node_id. Μιας και το R-Tree είναι ισοζυγισμένο, όλα τα φύλλα βρίσκονται πάντα στο ίδιο επίπεδο. Αν M είναι ο μέγιστος αριθμός στοιχείων που αποθηκεύονται ανά κόμβο και $m \leq \frac{M}{2}$ ο ελάχιστος αριθμός στοιχείων ανά κόμβο, τότε σύμφωνα με τον Guttman το R-Tree πληρεί τις παρακάτω ιδιότητες:

1. Κάθε φύλλο του δέντρου περιέχει μεταξύ m και M αντικείμενα, εκτός εάν είναι η ρίζα του δέντρου.

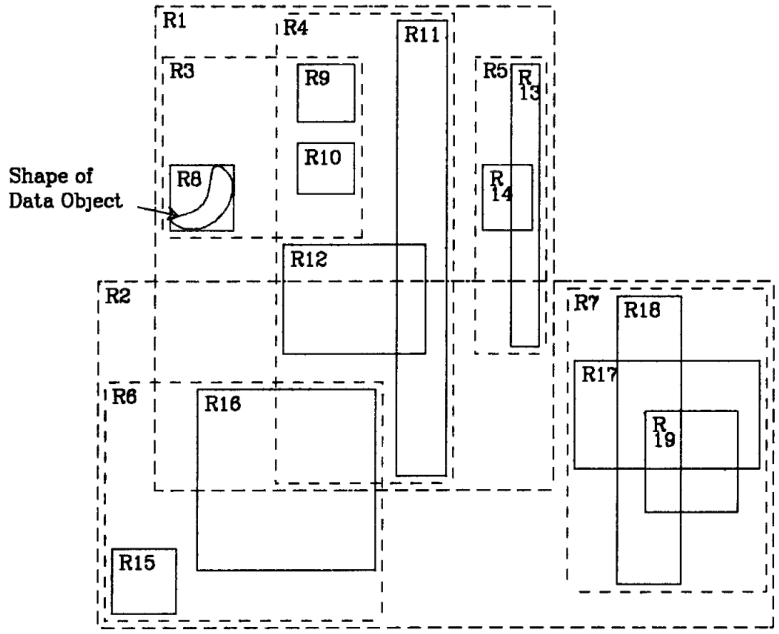
2. Κάθε κόμβος φύλλου που είναι της μορφής (MBR, object_id), το MBR του είναι το ελάχιστο περιγεγραμμένο ορθογώνιο που μπορεί να χωρέσει το 2-διάστατο αντικείμενο.

3. Κάθε κόμβος που δεν είναι φύλλο, περιέχει μεταξύ m και M παιδιά, εκτός εάν είναι η ρίζα του δέντρου.

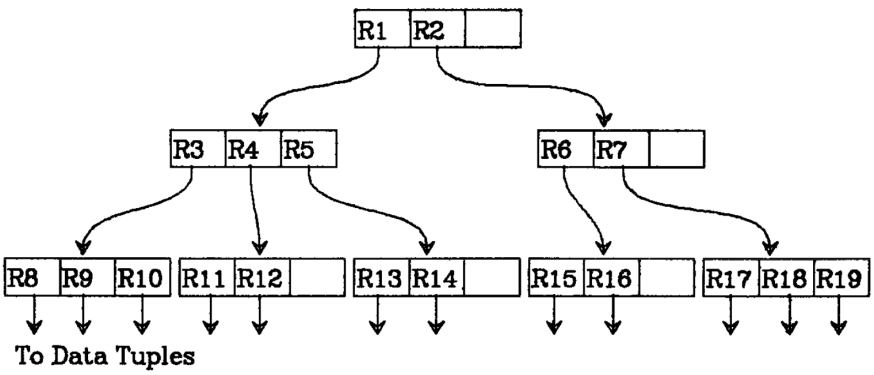
4. Για κάθε κόμβο (MBR, child_node_id) που δεν είναι φύλλο, το MBR είναι το ελάχιστο περιγεγραμμένο ορθογώνιο που χωράει τα ορθογώνια των παιδιών του.

5. Η ρίζα έχει τουλάχιστον δύο παιδιά, εκτός αν είναι και φύλλο.

6. Όλα τα φύλλα βρίσκονται πάντα στο ίδιο επίπεδο.



Εικόνα 2.1: *Παράδειγμα από MBR αντικειμένων και τα αντίστοιχα MBR τους. Πηγή: Gut84*



Εικόνα 2.2: *Το αντίστοιχο R-Tree. Πηγή: Gut84*

Έχοντας το R-Tree και δοθέντος ενός ορθογωνίου Q μπορούμε να βρούμε όλα τα αντικείμενα που περιέχονται μέσα στο Q. Αυτό ονομάζεται αναζήτηση εύρους. Ο αλγόριθμος 2.1 περιγράφει την διαδικασία του Range Query σε R-Tree.

Αλγοριθμος 2.1: *Αναζήτηση εύρους σε R-Tree*

```
1: function RangeQuery(Node N, Rectangle Q)
2:     if N is not a leaf node then
3:         examine each child c of N to find every c.mbr that overlaps with Q
4:         for all such children c do
5:             RangeQuery(c, Q)
6:         end for
7:     else                                                    ▷ N is a leaf node
8:         examine each child c of N to find every c.mbr that overlaps with Q
9:         add every child c that overlaps with Q to the answer set R
10:    end if
11: end function
```

Οι προσθήκες σε ένα R-Tree γίνονται με παρόμοιο τρόπο όπως με αυτές σε ένα B^+-Tree. Για να εισάγουμε κάποιο αντικείμενο στο R-Tree διατρέχουμε όλο το δέντρο μέχρι να βρούμε το κατάλληλο φύλλο για την προσθήκη. Το αντικείμενο εισάγετε στο φύλλο και στην συνέχεια όλοι οι κόμβοι που ήταν στην διαδρομή από την ρίζα μέχρι εκείνο το φύλλο ανανεώνονται με τις απαραίτητες αλλαγές. Σε περίπτωση που το φύλλο που βρήκαμε είναι γεμάτο (έχει M αντικείμενα) τότε διασπάται σε δύο φύλλα. Ο αλγόριθμος 2.2 περιγράφει την παραπάνω διαδικασία και ακολουθεί γραμμική διάσπαση κόμβων και φύλλων linear split.

Αλγοριθμος 2.2: *Εισαγωγή αντικειμένου σε R-Tree*

```
1: function Insert(Node N, Entry E)
2:     traverse the R-Tree from N to the appropriate leaf, selecting the node L whose
       MBR requires the minimum area enlargement to cover E.mbr
3:     if there is a tie between two nodes then
4:         select the node L which requires the minimum area
5:     end if
6:     if the selected leaf L has entries less than M then
7:         insert E into L
8:         update all MBRs in the path from N to L so that all cover E.mbr
9:     else                                                    ▷ L is full
10:        let C be the set consisting every child in L plus the new entry E
11:        select the two children C1 and C2 of which the distance between them is the
           greatest in the set C
12:        form two new leaf nodes L1 and L2, where the first contains C1 and the second
           contains C2
13:        assign the remaining children from C to the new nodes depending on which will
           require the minimum area enlargement
14:        if a tie occurs then
15:            assign the entry to the node with the least entries
16:        end if
17:        if one node is full and the other node contains M-r entries, where r are the
             remaining entries to be added then
18:            insert the remaining entries to that node without considering the criteria
               mentioned above
19:        end if
20:        update all MBRs in the path from N to L so that all cover L1.mbr and L2.mbr
21:        perform splits at the upper levels as well following the same algorithm if a node
           is full
22:        if the root has to be split then
23:            create a new root and increase the height of the tree by one
24:        end if
25:    end if
26: end function
```

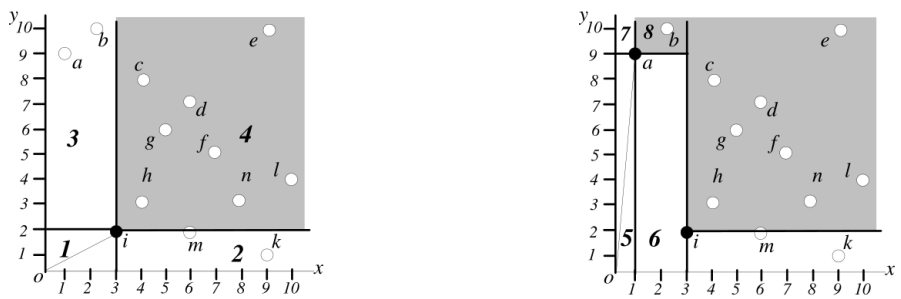
Ο **Linear Split** αλγόριθμος είναι ένας από τους τρεις τρόπους που είχε προτείνει ο Guttman. Οι άλλοι δύο είναι ο Quadratic Split αλγόριθμος και ο Exponential Split αλγόριθμος. Συγκεκριμένα ο Linear Split αλγόριθμος που παρουσιάζουμε σε αυτήν την περίπτωση, διαλέγει τα δύο αντικείμενα με την μεγαλύτερη απόσταση, και τα υπολειπόμενα αντικείμενα τα καταχωρεί με τυχαία σειρά στον κόμβο που χρειάζεται την ελάχιστη επέκταση.

Η διαγραφή ενός αντικειμένου παρουσιάζετε στον αλγόριθμο 2.3 Να σημειωθεί ότι ένας underflowing κόμβος, έχει άλλη μεταχείριση στο R-Tree σε σχέση με το B^+-Tree μιας και το R-Tree διαχειρίζεται δεδομένα πολλών διαστάσεων.

Αλγοριθμος 2.3: *Διαγραφή αντικειμένου σε R-Tree*

```
1: function DELETE(Node N, Entry E)
2:     if N is a leaf node then
3:         search all entries of N to find E.mbr
4:     else                                          ▷ N is an internal node
5:         find all entries of N that overlap with E.mbr
6:         follow the corresponding subtrees until the leaf L that contains E is found
7:         remove E from L
8:     end if
9:     remove all empty nodes after the removal of E starting from the leaf node L
10:    if the root has only one child and is not a leaf then
11:        remove the root
12:        set as new root its child
13:    end if
14: end function
```

Τέλος, έχει υλοποιηθεί και ο αλγόριθμος εύρεσης κορυφογραμμής (**Skyline Search**) [3] στο R-Tree. Ένα αντικείμενο ανήκει στην κορυφογραμμή, εάν δεν κυριαρχείται από κανένα άλλο στοιχείο του R-Tree. Ένα αντικείμενο λέμε ότι κυριαρχείται από κάποιο άλλο όταν η απόσταση του από την άξονα x αλλά και τον άξονα y είναι μεγαλύτερη από το άλλο. Συγκεκριμένα έχει υλοποιηθεί ο **Branch and Bound Skyline Algorithm**. Ο αλγόριθμος ξεκινάει από την ρίζα του δέντρου και προσθέτει όλα τα παιδιά της ρίζας σε ένα heap. Στην συνέχεια τα ταξινομεί σύμφωνα την ελάχιστη ευκλείδεια απόσταση της κοντινότερης γωνίας των MBR τους στην αρχή του άξονα. Αυτήν την απόσταση την ονομάζουμε mindist.



Εικόνα 2.3: *Παράδειγμα mindist. Πηγή: TODS05*

Στην συνέχεια, το αντικείμενο μέσα στο heap με το μικρότερο mindist ¨επεκτείνεται¨, αφαιρώντας τον εαυτό του και προσθέτοντας τα δικά του παιδιά μέσα στο heap. Ο heap ξανά ταξινομείται με τον ίδιο τρόπο. Όταν φτάσουμε στο πρώτο entry μέσα στο heap, το αφαιρούμε από το heap και το προσθέτουμε στο σύνολο S, το οποίο κρατάει όλα τα στοιχεία της κορυφογραμμής. Πλέον επαναλαμβάνουμε την διαδικασία, και κάθε φορά που φτάνουμε σε κάποιο entry, προσπαθούμε να το προσθέσουμε στο σύνολο S. Εάν αυτό το στοιχείο κυριαρχείται από οποιοδήποτε entry μέσα στο S, τότε το αφαιρούμε, αλλιώς το προσθέτουμε κανονικά. Όταν αδειάσει εντελώς το heap, τα στοιχεία μέσα στο S, είναι τα στοιχεία της κορυφογραμμής. Ο αλγόριθμος 2.4 περιγράφει την παραπάνω διαδικασία.

Αλγοριθμος 2.4: *Εύρεση κορυφογραμμής σε R-Tree*

```
1: function BBSSkylineSearch(Node N)
2:     let S be the list of skyline points, where S=∅
3:     insert all entries of N into the heap
4:     while heap is not empty do
5:         sort heap by mindist
6:         remove top child C
7:         if C is dominated by some point in S then
8:             discard C
9:         else                                          ▷ C is not dominated
10:            if C is a leaf node then
11:                for all children C_i of C do
12:                    if C_i is not dominated by some point in S, insert C_i into the heap
13:                end for
14:            else                                      ▷ C is not a leaf node
15:                insert the children of C into S
16:            end if
17:        end if
18:    end while
19: end function
```

2.3 Μοντέλο MapReduce

Το MapReduce [4] είναι ένα προγραμματιστικό μοντέλο και έχει εφαρμογή στην επεξεργασία και δημιουργία μεγάλων συνόλων δεδομένων. Για την επεξεργασία των δεδομένων, χρησιμοποιεί πολλούς υπολογιστές οι οποίοι ονομάζονται είτε συστάδα (cluster) εάν βρίσκονται σε τοπικό δίκτυο, είτε πλέγμα (grid) εάν βρίσκονται σε απομακρυσμένο δίκτυο. Για τον συντονισμό των διαδικασιών χρησιμοποιείται ένας υπολογιστής της συστάδας που ονομάζεται master node, ενώ οι υπόλοιποι υπολογιστές που εκτελούν τις εργασίας που αναθέτει το master node ονομάζονται worker nodes.

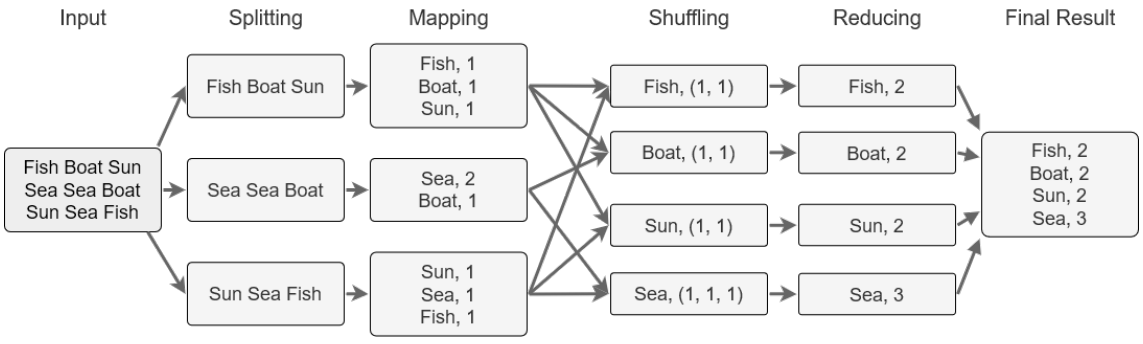
Προγράμματα που γράφονται με αυτόν τον συναρτησιακό τρόπο, παραλληλοποιούνται αυτόματα και εκτελούνται σε μεγάλο πλέγμα. Το προγραμματιστικό περιβάλλον λειτουργίας αναλαμβάνει τον διαμερισμό των δεδομένων εισόδου, την χρονοδρομολόγηση της εκτέλεσης του προγράμματος σε ένα σετ υπολογιστών, χειρίζεται τα πιθανά σφάλματα που μπορεί να

προκύψουν στους υπολογιστές ,και τέλος, χειρίζεται και την επικοινωνία μεταξύ των αυτών υπολογιστών. Αυτό επιτρέπει σε προγραμματιστές που δεν έχουν μεγάλη εμπειρία με κατανεμημένα περιβάλλοντα, να μπορούν να εκμεταλλευτούν τους πόρους που μπορεί να τους παρέχουν.

Υπάρχουν δύο κύριες εργασίες στο MapReduce από τις οποίες παίρνει και το όνομά του. Ο χρήστης προσδιορίζει μία συνάρτηση map, η οποία επεξεργάζεται ένα ζεύγος τιμής/κλειδιού για να δημιουργήσει ενδιάμεσα ζεύγη από τιμές/κλειδιά, και μία reduce συνάρτηση η οποία ενώνει όλες τις ενδιάμεσες τιμές με το ίδιο ενδιάμεσο κλειδί. Ο πιο συνηθισμένος διαχωρισμός των βημάτων του MapReduce είναι ο εξής:

- **Split:** Σε αυτό το βήμα διαμοιράζονται τα δεδομένα εισόδου για να επεξεργαστούν από τον Mapper του κάθε εργάτη.
- **Map:** Στην συνέχεια, κάθε εργάτης δημιουργέι τα key/value pairs από τα δεδομένα που πήρε από το split.
- **Shuffle:** Τα δεδομένα οργανώνονται εκ νέου, σύμφωνα με τα key/values που παράχθηκαν στο προηγούμενο βήμα.
- **Reduce:** Ο reducer παίρνει τα δεδομένα που προέκυψαν από το shuffle και συνδυάζει τα values, επιστρέφοντας μία μοναδική τιμή.
- **Final Result:** Τέλος ακολουθεί η αποθήκευση των δεδομένων σε κατανεμημένο σύστημα αρχείων.

Η αρχιτεκτονική του μοντέλου MapReduce φαίνεται στο παράδειγμα του σχήματος 2.5



Σχήμα 2.5: *Παράδειγμα αρχιτεκτονικής μοντέλου MapReduce*

Κεφάλαιο 3

Υλοποίηση R-Tree στην γλώσσα προγραμματισμού Python

Στο κεφάλαιο αυτό θα περιγραφεί ο τρόπος που υλοποιήθηκε το χωρικό ευρετήριο R-Tree στην γλώσσα προγραμματισμού Python. Η παρούσα υλοποίηση υποστηρίζει δισδιάστατα σημεία στον χώρο όπου η κύρια εφαρμογή της είναι η αποθήκευση συντεταγμένων.

3.1 Λειτουργίες

Στην παρούσα διπλωματική εργασία υλοποιήθηκαν οι παρακάτω λειτουργίες:

- Κατασκευή R-Tree
- Προσθήκη σημείου
- Αναζήτηση εύρους (Range Query)
- Αναζήτηση κορυφογραμμής (Skyline Search)
- Αναζήτηση κορυφογραμμής μέσα σε συγκεκριμένο εύρος
- Απεικόνιση του R-Tree (R-Tree Visualization)

3.2 Περιγραφή κλάσεων και μεθόδων

Στην ενότητα αυτή δίνεται μία περιγραφή των κλάσεων, των ορισμάτων και των βασικών μεθόδων που χρειάστηκαν για να υλοποιηθούν οι παραπάνω λειτουργίες.

3.2.1 class Point(x: int, y: int)

Η παρούσα υλοποίηση του R-Tree υποστηρίζει δισδιάστατα σημεία στον χώρο. Για αυτόν τον λόγο δημιουργήθηκε η κλάση Point, η οποία κρατάει τα σήμεια (x, y). Στην Point υπάρχει μία βοηθητική μέθοδος που χρησιμοποιείται για την εύρεση κορυφογραμμής στο ευρετήριο η οποία είναι η **dominates(point: Point)**. Παίρνει σαν όρισμα ένα άλλο αντικείμενο τύπου

Point και επιστρέφει True στην περίπτωση όπου το πρώτο αντικείμενο μας τύπου Point κυριαρχεί το το αντικείμενο Point που περάστηκε σαν όρισμα. Εάν δεν κυριαρχεί, τότε επιστρέφει False. Παρακάτω δίνεται ένα παράδειγμα της χρήσης της μεθόδου.

```
pointA = Point(5, 6)
pointB = Point(3, 2)
pointA.dominates(pointB)
>>True #pointA.x > pointB.x and pointA.y > pointB.y
```

3.2.2 class Rectangle(lower_left: Point, upper_right: Point)

Η κλάση Rectangle χρησιμοποιείται για τα MBRs του R-Tree. Παίρνει ως ορίσματα δύο αντικείμενα τύπου Point, όπου το πρώτο είναι το κάτω αριστερά σημείο του ορθογωνίου και το δεύτερο το πάνω δεξιά σημείο αντίστοιχα. Αυτά τα δύο σημεία είναι αρκετά για να φτιαχτεί ένα ορθογώνιο. Με την αρχικοποίηση του αντικειμένου τρέχει η μέθοδος **calculate_properties()** στο αντικείμενο η οποία υπολογίζει το πλάτος, το μήκος και το εμβαδό του ορθογωνίου που χρειάζονται αργότερα στην υλοποίηση. Οι βασικές μέθοδοι της Rectangle είναι οι εξείς:

- **overlaps(rect: Rectangle)**
  Η μέθοδος overlaps ελέγχει εάν το αντικείμενο Rectangle που την καλεί, επικαλύπτετε με το αντικείμενο Rectangle που περνάει ως όρισμα. Οι τιμές που επιστρέφει είναι True ή False.

- **contains(rect: Rectangle)**
  Η contains ελέγχει εάν το Rectangle που την καλεί επικαλύπτει πλήρως το Rectangle που περνάει ως όρισμα. Οι τιμές επιστροφής είναι True ή False.

- **expand_to_contain(rect: Rectangle)**
  Η μέθοδος expand_to_contain επεκτείνει τις αρχικές διαστάσεις του Rectangle που την καλεί ώστε να επικαλύπτει πλήρως το Rectangle που περνάει ως όρισμα.

- **create_mbr_for_entry(rect: Rectangle)**
  Η create_mbr_for_entry είναι μία classmethod, στην Python οι classmethods δεν χρειάζονται κάποιο αντικείμενο τύπου Rectangle για να εκτελεστούν γιατί είναι δεσμευμένες στην κλάση που είναι δηλωμένες. Σε αυτήν την περίπτωση, η μέθοδος παίρνει ως όρισμα ένα Rectangle και ως έξοδο επιστρέφει ένα αντικείμενο Rectangle το οποίο υπερκαλύπτει το Rectangle του ορίσματος. Παρακάτω δίνεται παράδειγμα εκτέλεσης της μεθόδου.

```
RectangleA = Rectangle(Point(1, 2), Point(4, 4))
RectangleB = Rectangle.create_mbr_for_entry(RectangleA)
RectangleB.overlaps(RectangleA)
>>True
```

- **expansions_area_cost(rect: Rectangle)**
  Αυτή η μέθοδος χρησιμοποιείται για να υπολογισθεί το κόστος επέκτασης ενός MBR για να χωρέσει κάποιο άλλο MBR. Χρησιμοποιείτε στις μεθόδους εισαγωγής για να βρεθούν τα κατάλληλα MBR για να γίνει το insertion και το split. Η τιμή επιστροφής της είναι η αύξηση του εμβαδού σε σχέση με το εμβαδό πριν την επέκταση.

- **dominates_rec(rect: Rectangle) & is_dominated(rect: Rectangle)**
  Οι δύο μέθοδοι αυτοί υπολογίζουν εάν ένα Rectangle κυριαρχεί κάποιο άλλο ή κυριαρχείτε από κάποιο άλλο αντίστοιχα. Οι τιμές επιστροφής είναι είτε True, είτε False. Ένα Rectangle κυριαρχεί κάποιο άλλο όταν το πάνω δεξιά σημείο του κυριαρχεί το κάτω αριστερά σημείο του άλλου.

- **mindist(point: Point)**
  Τέλος η μέθοδος mindist, υπολογίζει την ελάχιστη απόσταση ενός MBR από το σημείο που του δίνεται σαν όρισμα. Το προκαθορισμένο σημείο σε αυτήν την υλοποίηση είναι το Point(0, 0), δηλαδή η αρχή των αξόνων. Η απόσταση υπολογίζεται από την κοντινότερη γωνία του MBR στην αρχή των αξόνων.

Παρακάτω δίνεται κώδικας με παραδείγματα χρήσης των παραπάνω μεθόδων.

```
RectangleA = Rectangle(Point(2, 2), Point(7, 7))
RectangleB = Rectangle(Point(4, 4), Point(5, 5))
RectangleA.overlaps(RectangleB)
>>False

RectangleA.contains(RectangleB)
>>True

RectangleC = Rectangle.create_mbr_for_entry(RectangleB)
RectangleC.lower_left
>>Point(3, 3)
RectangleC.upper_right
>>Point(6, 6)

RectangleB.expand_to_contain(RectangleA)
RectangleB.lower_left
>>Point(1, 1)
RectangleB.upper_right
>>Point(8, 8)

RectangleA.dominates(RectangleB)
>>False

RectangleA.mindist(Point(0, 0))
>>2
```

3.2.3 class Entry(letter: str, mbr: Rectangle)

Η κλάση Entry είναι τα φύλλα του R-Tree. Η διαφορά τους με τους απλούς τους κόμβους που θα παρουσιαστούν πιο κάτω είναι ότι αποτελούνται μόνο από το MBR τους, και ένα string το οποίο μπορεί να είναι κάποια πληροφορία για το εκάστοτε Entry. Αυτά τα δύο στοιχεία είναι και αυτά που παίρνει ως όρισμα αυτή η κλάση. Η μόνο μέθοδος που την απαρτίζει είναι η **mindist**, η οποία όπως είδαμε και πριν υπολογίζει την απόσταση του MBR του Entry από ένα δοθέν Point. Στην υλοποίηση της διπλωματικής χρησιμοποιείται μόνο η αρχή των αξόνων.

3.2.4 class minList()

Η κλάση minList παίζει τον ρόλο της heap memory που περιγράφηκε στον αλγόριθμο 2.4. Είναι η κλάση στην οποία αποθηκεύονται όλα τα στοιχεία που ανήκουν στην κορυφογραμμή. Περιέχει τρεις λίστες που ενημερώνονται κατά την διάρκεια της εύρεσης της κορυφογραμμής. Οι λίστες είναι οι εξής:

- **toDo_list & mindist_list**
  Οι λίστες toDo_list και mindist_list είναι συνδεδεμένες μεταξύ τους. Στην toDo_list υπάρχει το περιεχόμενο της heap μνήμης που περιγράφετε στον αλγόριθμο 2.4. Είναι η λίστα με τα στοιχεία που θα αναλυθούν από την μέθοδο process για το αν ανήκουν στην κορυφογραμμή. Η λίστα mindist_list είναι οι mindist των στοιχείων της toDo_list. Έχουν αντιστοιχία καθ'όλη τη διάρκεια της εκτέλεσης του αλγορίθμου. Όταν γίνεται κάποιο sort στο mindist_list τότε ταξινομείτε και ο toDo_list σύμφωνα με αυτό.

- **skyline**
  Στην λίστα skyline ενημερώνονται τα στοιχεία της κορυφογραμμής που θα επιστραφούν μετά την εκτέλεση της μεθόδου process.

Οι βασικές μέθοδοι που απαρτίζουν την κλάση είναι οι εξής:

- **insert(node: Node, entry: Entry, point: Point)**
  Η μέθοδος insert παίρνει ως όρισμα είτε έναν κόμβο (Node), είτε ένα φύλλο (Entry) και ένα προαιρετικό Point. Το αντικείμενο Point χρησιμοποιείται για τον υπολογισμό του mindist. Αν δεν περάσει κάποιο αντικείμενο Point, θεωρείται ότι το mindist υπολογίζεται από την αρχή των αξόνων. Σε περίπτωση που θέλουμε να βρούμε κορυφογραμμή μέσα σε συγκεκριμένο range, πρέπει να περαστεί σαν όρισμα το Point της αρχής του range. Αφού υπολογιστεί το mindist του Node ή του Entry το προσθέτει στο mindist_list και τα περιεχόμενα του κόμβου ή του φύλλου προσθέτονται στο toDo_list. Στην συνέχεια κάνει ταξινόμηση και τις δύο λίστες με βάση το mindist_list.

- **isEmpty()**
  Η μέθοδος isEmpty επιστρέφει True εάν οι λίστες mindist_list και toDo_list είναι κενές. Αλλιώς επιστρέφεται η τιμή Fale.

- **process(rect: Rectangle)**
  Η μέθοδος process είναι η κύρια μέθοδος της κλάσης. Παίρνει ένα προαιρετικό αντικείμε-

νο Rectangle το οποίο είναι το range query στο οποίο μπορεί να εφαρμοστεί ένα skyline search. Κάθε εκτέλεση της process είναι μία εκτέλεση της while του αλγορίθμου 2.4.

3.2.5 class Node(mbr: Rectangle)

Η κλάση Node αντιπροσωπεύει όλους τους κόμβους στο R-Tree που δεν είναι φύλλα. Η κλάση παίρνει ως όρισμα ένα αντικείμενο Rectangle για να το ορίσει ως το MBR του κόμβου. Οι τρεις κύριες μεταβλητές που κρατάει είναι οι:

- **mbr: Rectangle**
  Το αντικείμενο αυτό κρατάει το MBR του κόμβου.

- **children: [Node]**
  Η λίστα children περιέχει αντικείμενα τύπου Node. Περιέχει στοιχεία μόνο όταν ο κόμβος βρίσκεται δύο επίπεδα πάνω και περισσότερο από τα φύλλα. Σε άλλη περίπτωση, ο κόμβος που είναι ένα επίπεδο πάνω από τα φύλλα γεμίζει την επόμενη λίστα.

- **entries: [Entry]**
  Αυτή τη λίστα την γεμίζουν μόνο οι κόμβοι που έχουν κόμβους φύλλα ως παιδιά. Στην υλοποίηση αυτή θα περιέχει στοιχεία είτε η λίστα children, είτε αυτή, δεν γίνεται να έχουν και οι δύο στοιχεία.

Οι βασικές μέθοδοι της κλάσης είναι οι παρακάτω:

- **insert_entry(entry: Entry)**
  Η μέθοδος insert_entry δέχεται σαν όρισμα ένα αντικείμενο τύπου Entry και το εισάγει στην λίστα entries του αντικείμένου που την κάλεσε. Στην συνέχεια επεκτείνει το mbr του αρχικού αντικειμένου σε περίπτωση που δεν καλύπτει ήδη το καινούργιο entry. Τέλος, ελέγχει αν η λίστα entries ξεπερνάει το επιτρεπτό όριο των αντικειμένον που έχει οριστεί. Σε αυτήν την περίπτωση καλεί την μέθοδο linear_split_entries η οποία διασπά το αντικείμενο όπως έχει περιγραφεί τον αλγόριθμο 2.2, και επιστρέφει δύο νέα Nodes. Σε κάθε άλλη περίπτωση επιστρέφει None.

- **chose_seeds_entries()**
  Η μέθοδος αυτή χρησιμοποιείται για να επιλεγούν τα δύο Entry Nodes που θα χρησιμοποιηθούν σε τυχόν διάσπαση του κόμβου. Επιλέγει δύο entries. Το πρώτο είναι αυτό του οποίου η κάτω αριστερή γωνία του mbr του έχει το μικρότερο y, ενω το δεύτερο είναι αυτό που η πάνω δεξιά γωνία του έχει το μεγαλύτερο y. Η τιμή επιστροφής της είναι οι index αριθμοί που δείχνουν την θέση των δύο επιλεγμένων entry στην λίστα entries του κόμβου που κάλεσε την μέθοδο. Αυτή η μέθοδος χρησιμοποιείται από την linear_split_entries που θα αναλυθεί στην συνέχεια.

- **linear_split_entries()**
  Η linear_split_entries διασπά έναν κόμβο με την linear split μέθοδο και επιστρέφει δύο νέους κόμβους όπου τα entries του αρχικού είναι διαχωρισμένα σε αυτά. Αρχικά καλεί την μέθοδο linear_split_entries για να επιλέξει τα αρχικά entries τα οποία θα

απαρτίζουν τα δύο καινούργια αντικείμενα. Στην συνέχεια μοιράζει τα υπολειπόμενα entries σε αυτά τα δύο καινούργια αντικείμενα, βλέποντας πιο θα έχει το μικρότερο κόστος επέκτασης του mbr του. Σε περίπτωση ισοβαθμίας, βάζει το entry στον πρώτο κόμβο.

- **insert(entry: Entry, level: int = 0)**
  Η μέθοδος insert εκτελείται αναδρομικά. Είναι η κύρια μέθοδος εισαγωγής αντικειμένου στο R-Tree και παίρνει δύο ορίσματα. Ένα αντικείμενο τύπου Entry για το οποίο γίνεται το insertion και μία μεταβλητή τύπου int η οποία χρησιμοποιείται στην αναδρομή για να προσδιορίσουμε το βάθος της κατά την εκτέλεση και να γίνει η σωστή διάσπασει τον κόμβων στα επίπεδα του R-Tree που χρειάζεται. Η αρχική τιμή της μεταβλητής level ξεκινάει πάντα από το μηδέν. Ο αλγόριθμος 2.2 περιγράφη την διαδικασία του insertion.

- **range_query(rec: Rectangle, range_entries: [Entry])**
  Τέλος η μέθοδος range_query επιστρέφει όλα τα entries που ανήκουν μέσα σε ένα δοθέν ορθογώνιο που περνάει ως όρισμα. Τα δύο ορίσματα που παίρνει η μέθοδος είναι το ορθογώνιο στο οποίο θέλουμε να βρούμε όλα τα entries που ανήκουν μέσα σε αυτό, και μία λίστα range_entries στην οποία αποθηκεύονται τα αποτελέσματα του query. Η μέθοδος εκτελείτε αναδρομικά και εκμεταλλεύεται πλήρως την αρχιτεκτονική του R-Tree. Ξεκινάει από τον κόμβο που την καλεί που συνήθως είναι η ρίζα του δέντρου και ελέγχει πιο mbr των παιδιών της διατέμνει το αντικείμενο Rectangle που έχει δοθεί ως όρισμα. Μόλις βρεθεί ένα τέτοιο παιδί, εκτελείτε εκ νέου η range_query περνώντας ως όρισμα το ίδιο rec και την ίδια λίστα που αποθηκεύονται τα αποτελέσματα. Με αυτήν την αναδρομή ο αλγόριθμος αποφεύγει όλα τα μονοπάτια που δεν χρειάζεται να πάει.

3.2.6 class RTree()

Η κλάση RTree λειτουργεί ως ο συντονιστής των Nodes και Entries. Το μόνο Node που κρατάει είναι η ρίζα, και χρησιμοποιεί τις μεθόδος της κλάσης Node ως αφετηρία την ρίζα. Για παράδειγμα οι μέθοδοι insert και range_query που υπάρχουν την κλάση RTree καλούν απλά τις μεθόδους της κλάσης Node πάνω στην ρίζα. Οι δύο μέθοδοι που υλοποιεί που δεν βρίσκονται στην κλάση Node είναι οι εξής:

- **bbs_skyline()**
  Η μέθοδος bbs_skyline υλοποιεί τον αλγόριθμο branch and bound skyline που παρουσιάζετε στον αλγόριθμο 2.4. Ο τρόπου που το επιτυγχάνει είναι προσθέτοντας τα παιδιά της ρίζας του δέντρου σε ένα αντικείμενο minList όπως είδαμε παραπάνω. Σε περίπτωση όπου η ρίζα δεν έχει παιδία, τότε κάνει insert μόνο την ρίζα. Στην συνέχεια χρησιμοποιεί την μέθοδο minList.isEmpty() και εκτελεί έναν βρόγχο while με την μέθοδο skyline = minList.proccess() όσο επιστρέφει False. Στο τέλος της εκτέλεσής του βρόγχου, η λίστα skyline περιέχει τα στοιχεία της κορυφογραμμής, τα οποία είναι και η τιμή επιστροφής της μεθόδου.

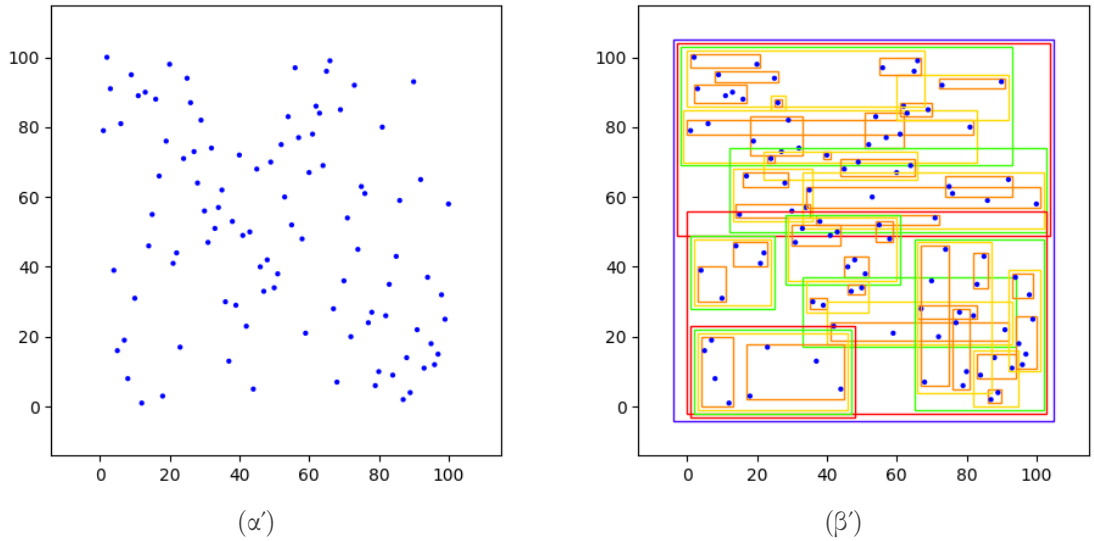
- **bbs_skyline_range_query(rec: Rectangle)**
  Η εκτέλεση της bbs_skyline_range_query είναι η ίδια με την μέθοδο που παρουσιάστη-

κε πριν, με την μόνη διαφορά ότι παίρνει ως όρισμα ένα αντικείμενο τύπου Rectangle στο οποίο θέλουμε εκεί μέσα να βρούμε τα στοιχεία της κορυφογραμμής. Η διαφορά στην υλοποίηση είναι ότι στην minList.insert() περνάει ως όρισμα την κάτω αριστερή γωνία του Rectangle ώστε το sorting του mindist να γίνει σύμφωνα με εκείνο το σημείο και επίσης στην μέθοδο minList.proccess() περνάει ως όρισμα όλο το Rectangle για να βρεθούν εκεί μέσα τα στοιχεία. Η περιγραφή λειτουργίας της κλάσης minList βρίσκεται στην υποενότητα 3.2.4.

3.3 R-Tree Visualization

Μαζί με την κύριο αλγόριθμο του R-Tree, έχουν υλοποιηθεί και μέθοδοι απεικόνισης του δέντρου. Για να γίνονται plot τα σημεία στο επίπεδο έχει χρησιμοποιηθεί η βιβλιοθήκη matplotlib [5] της Python. Στις κλάσεις main και RTree, επιπλέον από τις βασικές μεθόδους που παρουσιάστηκαν στην υποενότητα 3.2.6, έχουν υλοποιηθεί και οι παρακάτω λειτουργίες:

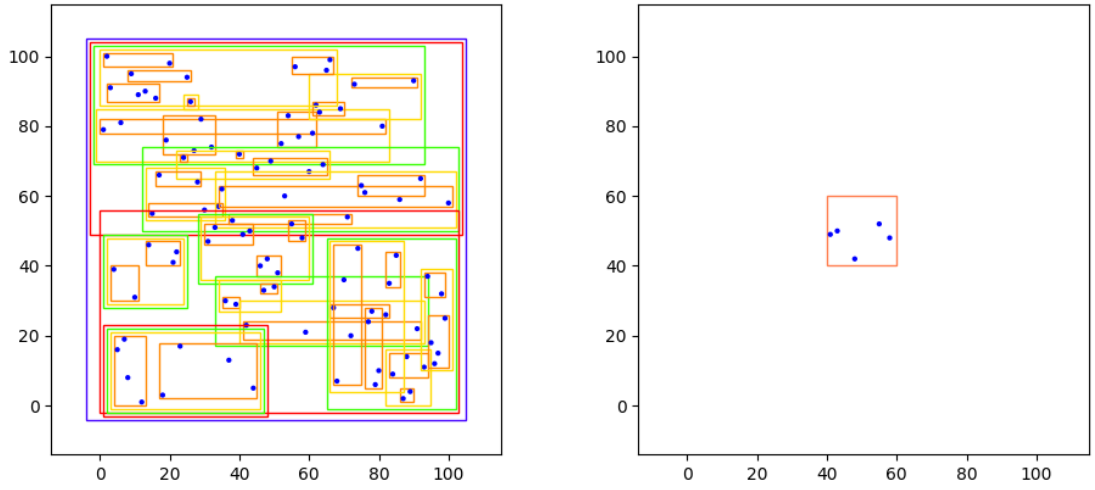
- **Απεικόνιση dataset και R-Tree**
  Η μέθοδος show_points της κλάσης RTree σχεδιάζει το R-Tree πάνω στον άξονα. Ξεκινάει από την ρίζα του δέντρου και καλείται αναδρομικά. Όταν προσθέσει όλα τα points και MBRs των παιδιών της ρίζας, καλείται ξεχωριστά στην συνέχεια για κάθε ένα παιδί. Στο τέλος καλείται η μέθοδος show πάνω στο αντικείμενο plt της matplotlib για να εμφανιστεί το δέντρο. Στην main κλάση του προγράμματος έχει υλοποιηθεί και λειτουργία για να εμφανιστούν όλα τα points πριν γίνουν εισαγωγή στο R-Tree. Παρακάτω δίνεται παράδειγμα της εκτέλεσης αυτών των μεθόδων σε τυχαίο dataset εκατό σημείων στο περιβάλλον ανάπτυξης λογισμικού Visual Studio Code.



Σχήμα 3.1: *(α′) Σημεία στον άξονα, (β′) Το R-Tree που προκύπτει*

- **Απεικόνιση αναζήτησης εύρους**
  Η μέθοδος RTree.range_query(Rectangle) επιστρέφει σε μία λίστα όλα τα στοιχεία που ανήκουν μέσα στο range. Η main παίρνει τα στοιχεία αυτά και τα απεικονίζει στον

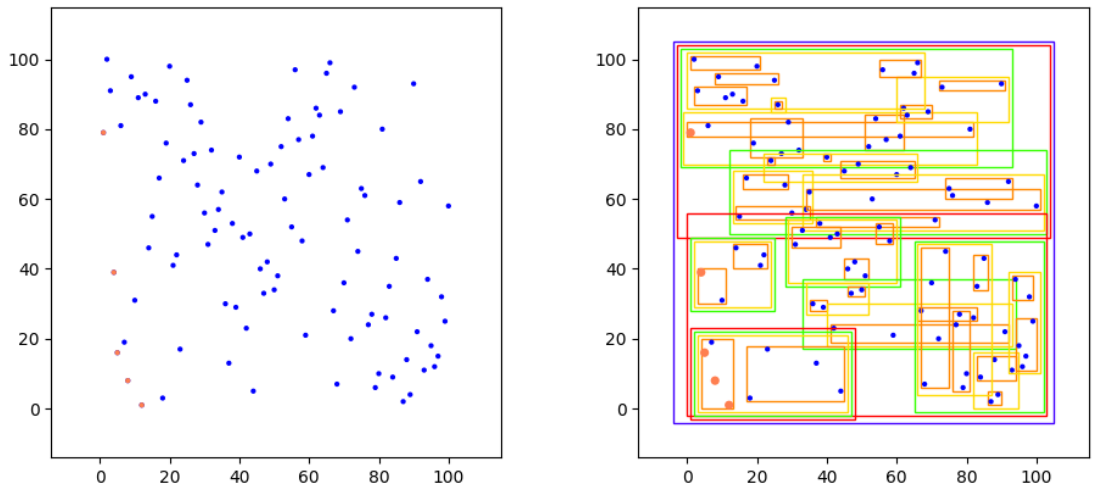
άξονα μαζί με το Rectangle που έχει δοθεί για το query. Το παρακάτω σχήμα δίνει ένα παράδειγμα απεικόνισης ενός τυχαίου range query.



(α′) (β′)

Σχήμα 3.2: *(α′) R-Tree, (β′) Το range query πάνω στο R-Tree*

- **Απεικόνιση αναζήτησης κορυφογραμμής**
  Όπως και με την RTree.range_query(Rectangle), έτσι και η μέθοδος RTree.skyline() επιστρέφει τα στοιχεία της κορυφογραμμής σε μία λίστα. Η main έχει μέθοδο για να απεικονίζει τα στοιχεία αυτά χωρίς το R-Tree, ενώ στην κλάση RTree έχει υλοποιηθεί μέθοδος show_skyline(skyline) η οποία απεικονίζει τα σημεία πάνω στο R-Tree. Στο σχήμα 3.3 δίνεται ένα τέτοιο παράδειγμα.

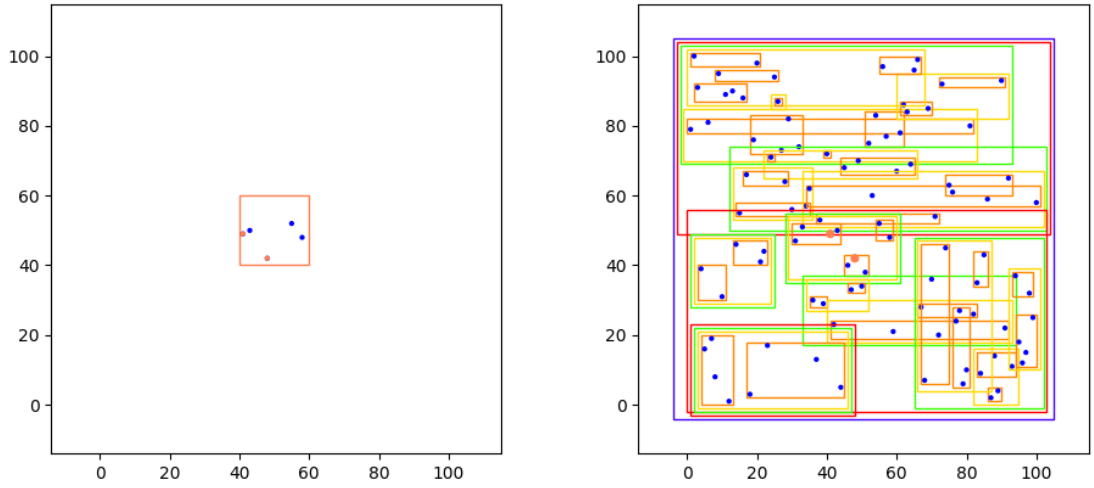


(α′) (β′)

Σχήμα 3.3: *(α′) Skyline Query, (β′) Skyline Query με το R-Tree*

- **Απεικόνιση στοιχείων κορυφογραμμής μέσα σε συγκεκριμένο εύρος**
  Τέλος, έχει υλοποιηθεί η δυνατότητα απεικόνισης κορυφογραμμής μέσα σε συγκεκριμένο εύρος. Η μέθοδος RTree.bbs_skyline_range_query(Rectangle) επιστρέφει το

αποτέλεσμα της με τα στοιχεία σε μία λίστα. Η main απεικονίζει τα στοιχεία αυτά χωρίς το R-Tree στο επίπεδο, ενώ στην κλάση R-Tree καλείται ξανά η μέθοδος show_skyline(skyline) όπως είχαμε δει στο προηγούμενο παράδειγμα. Το σχήμα 3.4 δείχνει την εκτέλεση αυτών των μεθόδων.



Σχήμα 3.4: *(α′) Skyline Range Query, (β′) Skyline Range Query με το R-Tree*

Κεφάλαιο 4

Το κατανεμημένο περιβάλλον Spark

Στο κεφάλαιο αυτό θα παρουσιάσουμε το κατανεμημένο περιβάλλον Spark. Θα δούμε τι ακριβώς είναι το Spark, πως ξεκίνησε και που χρησιμοποιείται. Τέλος θα δούμε κάποιες βασικές συναρτήσεις που υποστηρίζει και πως μπορούμε να το αξιοποιήσουμε.

4.1 Τι είναι το Apache Spark

Το Apache Spark [6] είναι ένα κατανεμημένο και εξαιρετικά επεκτάσιμο σύστημα ανάλυσης δεδομένων στη μνήμη, παρέχοντας την δυνατότητα να αναπτυχθούν εφαρμογές σε Scala, Python, Java και R. Έχει ένα από τα υψηλότερα ποσοστά συνεισφοράς μεταξύ των project της Apache αυτήν τη στιγμή.

Εσωτερικά το Apache Spark δουλεύει πάνω στο μοντέλο του MapReduce αλλά είναι εξαιρετικά βελτιστοποιημένο. Αντί για απλές λειτουργίες Map και Reduce, το Spark διαθέτει ένα μεγάλο σύνολο λειτουργιών που ονομάζονται μετασχηματισμοί και ενέργειες, οι οποίες μετατρέπονται σε Map/Reduce από το περιβάλλον εκτέλεσης του Spark.

Παρέχει τέσσερις κύριες βιβλιοθήκες, οι οποίες είναι το Spark SQL, Spark Streaming, MLlib και GraphX. Οι βιβλιοθήκες αυτές είναι διαλειτουργικές, δηλαδή τα δεδομένα μπορούν να διαβάζονται από όλες τις βιβλιοθήκες. Για παράδειγμα, τα δεδομένα ροής μπορούν να διαβαστούν από την βιβλιοθήκη Spark SQL, και να δημιουργηθεί ένας πίνακας από αυτά.

4.2 Ιστορία

Το Apache Spark [7] ξεκίνησε ως ερευνητικό project από τον Matei Zaharia στο πανεπιστήμιο της California το 2009 και μετατράπηκε σε project ανοιχτού κώδικα στις αρχές του 2010 κάτω από την άδεια BSD. Ο κύριος στόχος του project ήταν η ανάλυση μεγάλων δεδομένων και ο λόγος που αναπτύχθηκε ήταν για να ξεπεράσει τις ανεπάρκειες του MapReduce. Ενώ το MapReduce είχε μεγάλη επιτυχία και είχε μεγάλη αποδοχή από την επιστημονική κοινότητα, δεν μπορούσε να εφαρμοστεί σε μεγάλο φάσμα προβλημάτων. Συγκεκριμένα, δεν είχε καλή απόδοση σε προβλήματα που απαιτούσαν κοινή χρήση δεδομένων χαμηλής καθυστέρησης σε πολλές παράλληλες λειτουργίες. Το Spark παρουσίασε το Resilient Distributed

Dataset (RDD), μία ανεκτική σε σφάλματα συλλογή στοιχείων που μπορούν να λειτουργήσουν παράλληλα.

4.3 Το Resilient Distributed Dataset (RDD) στο περιβάλλον Spark

Τα RDDs [8] είναι αμετάβλητα, διαμερισμένα και κατανεμημένα σύνολα δεδομένων που χρησιμοποιούνται από το Spark για την επεξεργασία των δεδομένων. Είναι επίσης ανθεκτικά σε σφάλματα και μπορούν να αναδημιουργηθούν σε οποιοδήποτε στάδιο επεξεργασία σε περίπτωση που προκύψει κάποια αποτυχία στο δίκτυο ή στους κόμβους. Μπορούν να δημιουργηθούν είτε παραλληλίζοντας μία υπάρχουσα συλλογή δεδομένων του προγράμματος, είτε διαβάζοντας ένα σύνολο δεδομένων από εξωτερικό χώρο αποθήκευσης όπως το Hadoop Distributed File System (HDFS) της Apache, την μη σχεσιακή βάση δεδομένων Cassandra κλπ., είτε μετατρέποντας RDD που υπάρχει ήδη στην μνήμη και επεξεργάζεται από το Spark. Το Spark χρησιμοποιεί τα RDDs για να πετύχει παρόμοια αποτελέσματα επεξεργασίας με το μοντέλο MapReduce αλλά με πολύ μεγαλύτερη ταχύτητα.

Η κύρια διαφορά ανάμεσα στο μοντέλο MapReduce και το Spark βρίσκεται στην προσέγγιση της επεξεργασίας των δεδομένων. Ενώ το MapReduce χρειάζεται να γράψει και να διαβάσει δεδομένα από έναν δίσκο για να τα επεξεργαστεί, το RDD επιτρέπει στο Spark να επεξεργαστεί τα δεδομένα στην μνήμη.

4.3.1 Πράξεις σε RDD

Οι δύο πράξεις που μπορούν να εφαρμοστούν σε ένα RDD είναι μετασχηματισμοί (transformations) και ενέργειες (actions). Ένας μετασχηματισμός είναι μία συνάρτηση η οποία δημιουργέι ένα RDD από ένα ήδη υπάρχων, ενώ μία ενέργεια επιστρέφει κάποια τιμή από το πρόγραμμα οδήγησης μετά την εκτέλεση του. Ένα χαρακτηριστικό των μετασχηματισμών στο Spark είναι ότι είναι lazy. Αυτό σημαίνει ότι κάθε φορά που εκτελεί ο χρήστης κάποιον μετασχηματισμό, αυτός δεν εκτελείτε κατευθείαν. Το Spark κρατάει αρχείο με τους μετασχηματισμούς και τους εκτελεί μόνο όταν εκτελεστεί κάποια ενέργεια στο καινούργιο RDD που προκύπτει. Με αυτόν τον τρόπο, χρειάζονται να εκτελεστούν κάθε φορά μόνο οι μετασχηματισμοί που χρειάζονται μειώνοντας με αυτόν τον τρόπο τις φορές που χρειάζεται να επικοινωνήσουν η συστάδα και το πρόγραμμα οδηγός, κάνοντας το Spark αποδοτικότερο.

Μερικοί από τους πιο συνηθισμένους μετασχηματισμούς σε RDD είναι οι εξής:

- **map:** Ο μετασχηματισμός map() παίρνει ως όρισμα μία συνάρτηση και την εφαρμόζει σε κάθε στοιχείο του RDD. Το αποτέλεσμα της είναι ένα νέο RDD με όλα τα στοιχεία που επέστρεψε η συνάρτηση που είχε ως όρισμα.

- **flatmap:** Η flatmap() είναι παρόμοια με τον μετασχηματισμό map() με την διαφορά ότι η συνάρτηση που περνάει ως όρισμα της δεν επιστρέφει αναγκαία μόνο ένα αποτέλεσμα. Μπορεί να επιστρέψει από 0 και πάνω. Δηλαδή μετατρέπει ένα RDD M στοιχείων, σε

ένα άλλο RDD N στοιχείων.

- **filter:** Η filter() παίρνει ως όρισμα μία συνθήκη. Τα στοιχεία του νέου RDD είναι αυτά που επέστρεψαν true στην συνθήκη.

Τέλος, μερικές από τις πιο συνηθισμένες ενέργειες πάνω σε RDD είναι οι παρακάτω:

- **collect:** Η ενέργεια collect() είναι από τις πιο σύνηθες και η πιο απλή από όλες. Όταν εκτελείται, επιστρέφει όλο το περιεχόμενο του RDD στο πρόγραμμα οδηγό. Συνήθως εκτελείται σε μικρό σετ δεδομένων, μετά από την filter, group κλπ. αλλιώς μπορεί να καταλήξει σε πρόβλημα μη διαθέσιμης μνήμης.
- **count:** Η count() επιστρέφει τον αριθμό των στοιχείων μέσα στο RDD.
- **reduce:** Η reduce() παίρνει ως όρισμα δύο στοιχεία του RDD και έναν δυαδικό τελεστή και πραγματοποιεί την πράξη ανάμεσα τους. Για παράδειγμα, η τιμή εξόδου της reduce του παρακάτω κώδικα είναι 6.

```
rdd = sc.parallelize([1, 2, 3])
rdd.reduce(lambda a, b: a * b)
```

4.4 Κατευθυνόμενο Ακυκλικό Γράφημα (Directed Acyclic Graph) στο περιβάλλον Spark

Στην προηγούμενη ενότητα είπαμε ότι οι μετασχηματισμοί στο Spark είναι lazy, δηλαδή εκτελούνται μόνο όταν στο προκύπτον RDD εκτελείται μία ενέργεια. Ο τρόπος με τον οποίο το Spark 'θυμάται' τους μετασχηματισμούς που έχουν γίνει για να τους εκτελέσει αργότερα είναι μέσο ενός κατευθυνόμενου άκυκλου γραφήματος (DAG). Το DAG από την θεωρία γράφων είναι είναι ένα κατευθυνόμενο γράφημα από του οποίου οι ακμές πάνε από κόμβο σε κόμβο χωρίς να έχει κύκλους. Δηλαδή εάν ακολουθήσεις μία διαδρομή στο γράφημα, δεν γίνεται να ξανά περάσεις από κόμβο που έχεις επισκεφτεί ήδη μία φορά. Στο Spark οι κόμβοι είναι τα RDDs και οι ακμές οι μετασχηματισμοί πάνω σε αυτούς. Κάθε φορά που καλείται κάποια ενέργεια, το δημιουργημένο DAG υποβάλλεται στο **DAG Scheduler** ο οποίος διαχωρίζει περαιτέρω το γράφημα σε στάδια του σχεδίου εκτέλεσης.

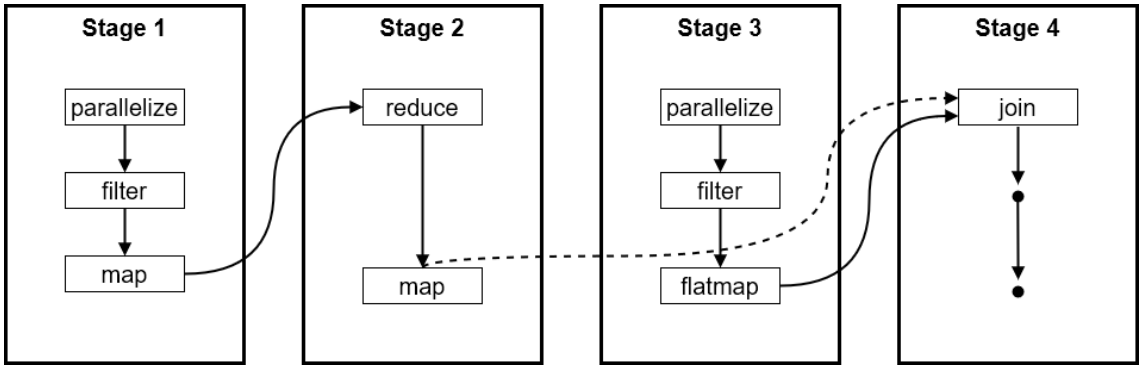
Οι περιορισμοί του MapReduce έγιναν βασικό σημείο για την εισαγωγή του DAG στο Spark. Οι υπολογισμοί στο MapReduce γίνονται σε τρία βήματα:

1. Τα δεδομένα διαβάζονται από το Hadoop Distributed File System (HDFS).
2. Εκτελούνται οι Map και Reduce διαδικασίες.
3. Το υπολογισμένο αποτέλεσμα γράφεται πίσω στο HDFS.

Έτσι κάθε διαδικασία του MapReduce είναι ανεξάρτητη από κάθε άλλη και δεν ξέρει ποια διαδικασία έρχεται μετά της. Για κάποιες επαναλήψεις του προγράμματος δεν υπάρχει λόγος

να διαβαστούν και να γραφτούν οι ενδιάμεσες τιμές της εκτέλεσης στην μνήμη. Με αυτόν τον τρόπο σπαταλάτε μνήμη στο HDFS.

Το Spark σε αντίθεση με το μοντέλο MapReduce δημιουργέι ένα DAG από τα διαδοχικά στάδια υπολογισμού. Με αυτόν τον τρόπο, βελτιστοποιείτε το σχέδιο εκτέλεσης. Γίνονται λιγότερες εγγραφές και ανακατανομές δεδομένων που δεν χρειάζονται. Αντίθετα, στο MapReduce κάθε βήμα πρέπει να βελτιστοποιηθεί κάθε βήμα χειροκίνητα. Το σχήμα 4.1 δείχνει ένα παράδειγμα ενός DAG.



Σχήμα 4.1: *Παράδειγμα DAG*

4.5 Αρχιτεκτονική του Apache Spark

Όπως αναφέρθηκε σε προηγούμενη ενότητα, το Spark λειτουργεί συνήθως σε μία συστάδα υπολογιστών ή αλλιώς σε ένα κατανεμημένο σύστημα. Για να λειτουργήσει ένα τέτοιο σύστημα χρειάζεται έναν υπολογιστή που να διαχειρίζεται όλη την συστάδα. Στο Spark, ο υπολογιστής αυτός καλείται κύριος κόμβος ή αλλιώς **master node**.

4.5.1 Master Node

Το master node χρησιμοποιείται για την ενορχήστρωση της συστάδας του Spark. Δηλαδή οργανώνει τα tasks που διαχωρίζονται στην συστάδα, καθώς και ποιοι υπολογιστές είναι διαθέσιμοι καθ'όλη τη διάρκεια ζωής της.

1. Το master node είναι ένας υπολογιστής όπους όλους τους υπόλοιπους. Χρησιμοποιείτε για να 'φιλοξενήσει' το πρόγραμμα οδηγό (driver program) του Spark και να διαχειριστεί όλη την συστάδα. Ο κόμβος αυτός είναι ο σύνδεσμος ανάμεσα στον χρήστη και την υπόλοιπη συστάδα.

2. Μιας και το Spark είναι γραμμένο στην γλώσσα προγραμματισμού Scala, είναι αναγκαίο κάθε υπολογιστής στο κατανεμημένο σύστημα να τρέχει ένα JVM (Java Virtual Machine), για να μπορεί το Spark να δουλεύει με το υλικό των υπολογιστών.

3. Το Spark Program που τρέχει μέσα στο JVM χρησιμοποιείται για να δημιουργήσει το SparkContext, το οποίο είναι το σημείο πρόσβασης του χρήστη στο κατανεμημένο σύστημα.

Το driver program επίσης περιέχει τον DAG scheduler, task scheduler, backend scheduler και το block manager. Όλα αυτά τα τμήματα είναι υπεύθυνα για την μετάφραση του κώδικα του χρήστη σε Spark εργασίες που θα εκτελεστούν στην συστάδα. Το SparkContext επικοινωνεί με τον συντονιστή συστάδας (**cluster manager**), ο οποίος είναι υπεύθυνος για την απόκτηση πόρων από την συστάδα Spark και για την κατανομή τους σε μία διεργασία Spark. Αυτοί οι πόροι έρχονται με την μορφή των κόμβων εργαζομένων (worker nodes).

4.5.2 Worker Node(s)

Τα worker nodes απαρτίζουν το ΄κατανεμημένο΄ κομμάτι της συστάδας. Οι κόμβοι αυτοί δεν είναι αναγκαίο να είναι πανομοιότυποι, δηλαδή μπορούν να έχουν διαφορετικά μεγέθη και αποδόσεις.

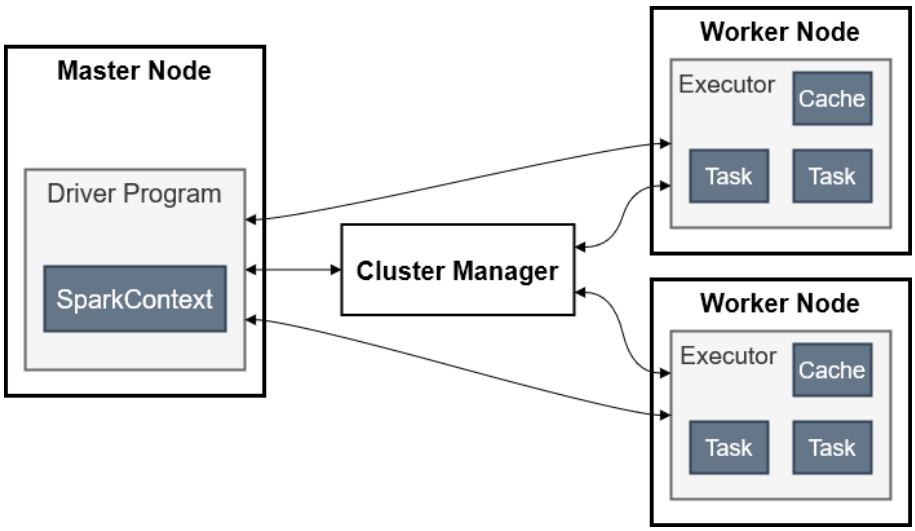
1. Τα worker nodes είναι συνήθως ξεχωριστοί υπολογιστές.

2. Όπως και με το master node, για να τρέξει το Spark σε ένα worker node, πρέπει να έχει εγκαταστημένη την κατάλληλη έκδοση της Java.

3. Κάθε worker node τρέχει έναν executor. Όταν τρέχει το πρόγραμμα οδηγός στο master node, το SparkContext επικοινωνεί με τον cluster manager, όπου αυτός στην συνέχεια βρίσκει τους πόρους στα worker nodes, και τους αναθέτει τα tasks που πρέπει να εκτελέσει ο executor. Ο cluster manager βλέπει πόσες θέσεις (slots) έχει κάθε worker node, οι οποίες είναι ο αριθμός των πυρήνων του επεξεργαστή του υπολογιστή. Σε κάθε πυρήνα μπορεί να του ανατεθεί ένα task.

4. Η μνήμη του worker node είναι χωρισμένη σε δύο τμήματα. Στην μνήμη αποθήκευσης και στην λειτουργική μνήμη. Ο διαχωρισμός της μνήμης είναι συνήθως 50:50, κάτι που μπορεί να αλλαχτεί από τις ρυθμίσεις του Spark.

Στο σχήμα 4.2 φαίνεται η αρχιτεκτονική του Spark.



Σχήμα 4.2: *Παράδειγμα αρχιτεκτονικής Spark*

Κεφάλαιο 5

Υλοποίηση R-Tree στο περιβάλλον Spark

Σε αυτό το κεφάλαιο θα παρουσιαστεί αναλυτική η υλοποίηση του R-Tree στο κατανεμημένο περιβάλλον Spark. Συγκεκριμένα θα αναλυθούν οι τρόποι δημιουργίας του RDD που περιέχει τα στοιχεία προς εισαγωγή, την κατανεμημένη αναζήτηση εύρους, την κατανεμημένη εύρεση κορυφογραμμής, καθώς και την κατανεμημένη εύρεση κορυφογραμμής μέσα σε ένα συγκεκριμένο εύρος.

5.1 Κατασκευή κατανεμημένου συνόλου δεδομένων

Το αρχικό σύνολο δεδομένων των σημείων διαβάζεται από το αρχείο data.csv. Στην πρώτη γραμμή το αρχείο έχει ως επικεφαλίδα το όνομα της κάθε διάστασης του αρχείου, και στις υπόλοιπες γραμμές περιέχει τις συντεταγμένες, διαχωρισμένες μεταξύ τους με κόμμα (Comma Seperated Values). Επομένως, για να κατασκευαστεί το κατάλληλο RDD μετατρέποντας τα σημεία σε Entry objects έπρεπε να διαβαστεί το παραπάνω αρχείο και να υποστεί κάποιες μεταμορφώσεις. Συγκεκριμένα, παιρνόντας τα στοιχεία του αρχείου σε ένα RDD, πρώτα αφαιρέθηκε η επικεφαλίδα που δεν περιείχε σημεία, και στην συνέχεια διαχωρίστηκαν τα σημεία σε λίστα με σημείο διαχωρισμού το κόμμα. Τέλος, μεταμορφώθηκαν τα σημεία σε αντικείμενα Entry, δίνοντας σε καθένα και ένα όνομα. Ο παρακάτω κώδικας δείχνει την διαδικασία που περιγράφηκε παραπάνω.

```
# Creating the points RDD
pointsRDD = sc.textFile('data-sets\data.csv', minPartitions=4)
header = points.first()
pointsRDD = pointsRDD.filter(lambda row: row != header)
pointsRDD =pointsRDD.map(lambda x: x.split(",")[0:2])
pointsRDD = pointsRDD.map(lambda x: Rectangle(
    Point(int(x[0]), int(x[1])), Point(int(x[0]), int(x[1]))))

# Creating the entries RDD
pointsRDD = pointsRDD.zipWithIndex()
entriesRDD = pointsRDD.map(lambda x: Entry("E"+str(x[1]), x[0]))
```

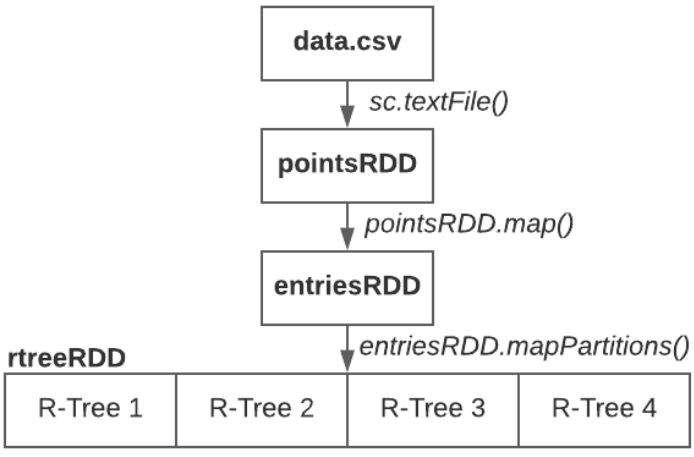
Ο παραπάνω κώδικας μπορεί να γραφεί και σε λιγότερες γραμμές, αλλά έχει διασπαστεί για την καλύτερη κατανόηση των μεταμορφώσεων.

5.2 Αρχική κατασκευή του δέντρου R-Tree στο περιβάλλον Spark

Ο παραλληλισμός έχει αξιοποιηθεί για την κλιμάκωση των R-Tree σε μεγάλα σύνολα δεδομένων. Μια πρώιμη μελέτη [9] εξετάζει την αποθήκευση ενός R-Tree σε ένα σύστημα πολλαπλών δίσκων, αποθηκεύοντας κάθε κόμβο που εισάγετε στο δίσκο που περιέχει τους περισσότερους ανόμοιους κόμβος για την βελτιστοποίηση ολόκληρου του συστήματος. Αυτός ο διαχωρισμός αν και χρήσιμος στις αναζητήσεις εύρους, δεν είναι πάντα ο βέλτιστος για τις αναζητήσεις κορυφογραμμής. Στην υλοποίηση που παρουσιάζεται, χωρίζεται το RDD που περιέχει τα Entry objects σε partitions, όπου στο κάθε ένα δημιουργούνται εκ νέου R-Trees. Το RDD που προκύπτει λειτουργεί ως ένα top level index. Παρακάτω δίνονται οι μετασχηματισμοί που έγιναν πάνω στο RDD για την δημιουργία των κατανεμημένων R-Trees.

```
# Function used inside mapPartitions to create the R-Trees
def create_rtree(entries):
    rtree = RTree()
    for entry in entries:
        rtree.insert(entry)
    yield rtree
# The function create_rtree is used on the data of each partition
rtreeRDD = entries.mapPartitions(create_rtree)
```

Το σχήμα 5.1 παρουσιάζει τις αλλαγές που έγιναν πάνω στο αρχικό σύνολο δεδομένων.



Σχήμα 5.1: *Δημιουργία του rtreeRDD*

5.3 Λειτουργίες

Στην παρούσα διπλωματική εργασία υλοποιήθηκαν οι παρακάτω λειτουργίες στο κατανεμημένο R-Tree:

- Κατασκευή R-Tree
- Αναζήτηση εύρους (Range Query)
- Αναζήτηση κορυφογραμμής (Skyline Search)
- Αναζήτηση κορυφογραμμής μέσα σε συγκεκριμένο εύρος

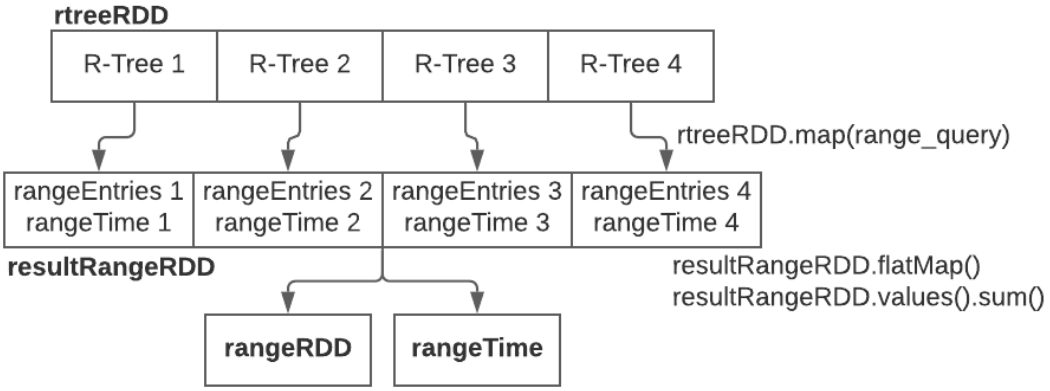
Στις ενότητες 5.1 και 5.2 έχουν παρουσιαστεί η δημιουργία του συνόλου δεδομένων και η δημιουργία του R-Tree από αυτό. Στην συνέχεια θα αναλυθούν και οι τρόπου υλοποίησης της αναζήτησης εύρους, της αναζήτησης κορυφογραμμής και της αναζήτησης κορυφογραμμής μέσα σε συγκεκριμένο εύρος.

5.3.1 Κατανεμημένη αναζήτηση εύρους

Για την πραγματοποίηση αναζήτησης εύρους, γίνεται map η συνάρτηση αναζήτησης εύρους που έχει υλοποιηθεί ήδη σε κάθε root κόμβο του rtreeRDD του οποίου το MBR περιέχει το ορθογώνιο της αναζήτησης. Τα αποτελέσματα επιστρέφονται σε ένα άλλο RDD, μαζί με τον χρόνο εκτέλεσης σε κάθε worker node. Τέλος, στο προκύπτον RDD εφαρμόζουμε έναν μετασχηματισμό flatMap ο οποίος παίρνει τις λίστες που επιστράφηκαν από τα R-Tree, και τις ενώνει σε μία. Παρακάτω δίνεται παράδειγμα εκτέλεσης.

```
# Example of a range query
range_rec = Rectangle(Point(40, 40), Point(60, 60))
resultRangeRDD = rtreeRDD.map(lambda tree: tree.range_query(
    range_rec))

# Flatten the result lists with the entries to a single list
rangeRDD = resultRangeRDD.flatMap(lambda result: flatten_entries(
    result))

# Add the times each worker node needed to execute the query into
    a single variable
rangeTime = resultRangeRDD.values().sum()
```

Το σχήμα δείχνει 5.2 σχηματικά τους μετασχηματισμούς που εφαρμόστηκαν.



Σχήμα 5.2: *Δημιουργία του rangeRDD και rangeTime*

5.3.2 Αναζήτηση κορυφογραμμής και αναζήτηση κορυφογραμμής σε συγκεκριμένο εύρος

Όπως και στο range query που παρουσιάστηκε στην υποενότητα **;;**, έτσι και στην αναζήτηση κορυφογραμμής η συνάρτηση εύρεσης κορυφογραμμής εφαρμόζεται σε κάθε partition του R-Tree. Το αποτέλεσμα αποθηκεύεται σε ένα νέο RDD το οποίο περιέχει λίστες με τα σημεία κορυφογραμμής κάθε R-Tree του κατανεμημένου περιβάλλοντος. Η διαφορά με το range query είναι ότι τα στοιχεία του κάθε R-Tree που επιστράφηκαν δεν είναι αναγκαία και τα στοιχεία κορυφογραμμής ολόκληρου του συνόλου δεδομένων. Για την λύση του προβλήματος εφαρμόστηκε ο αλγόριθμός **All Local Skyline (ALS)** [10], ο οποίος είναι η βασική προσέγγιση εύρεσης κορυφογραμμής σε κατανεμημένο περιβάλλον και περιγράφετε στον αλγόριθμο 5.1. Τα entries των κορυφογραμμών των R-Tree αποθηκεύονται στην κύρια μνήμη του προγράμματος και στην συνέχεια δημιουργείται νέο τοπικό R-Tree με αυτά. Στο τοπικό R-Tree εφαρμόζεται η μέθοδος εύρεσης κορυφογραμμής και τα στοιχεία που επιστρέφει είναι και το τελικό αποτέλεσμα. Η διαδικασία είναι η ίδια και για την εύρεση κορυφογραμμής σε συγκεκριμένο εύρος.

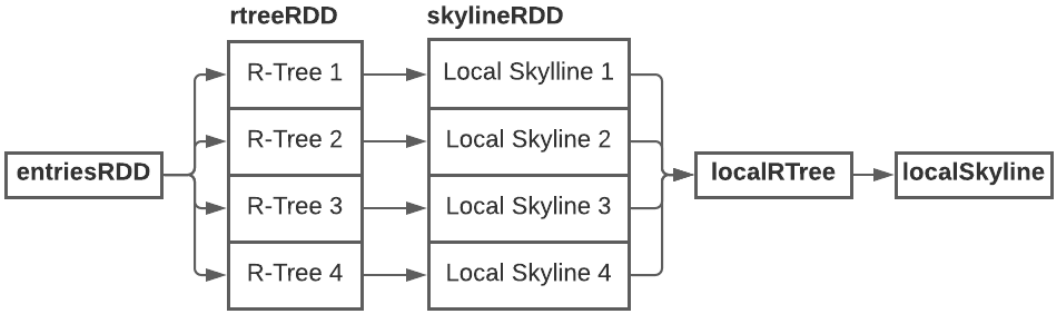
Αλγοριθμος 5.1: *Κατανεμημένη εύρεση κορυφογραμμής*

```
1: let D be the input dataset
2: datasetRDD ← parallelize D into partitions
3: for all partitions in datasetRDD do
4:     skylinesRDD ← calculate local skyline points in parallel
5: end for
6: Collect the local skylines to the main memory and create a local R-Tree
7: Apply the skyline method on the local R-Tree to calculate the final skyline
```

Παρακάτω δίνεται κώδικας εκτέλεσης του αλγορίθμου που περιγράφηκε.

```
# Skyline search on each partition
resultSkylineRDD = rtreeRDD.map(lambda tree: tree.bbs_skyline())
skylineRDD = resultSkylineRDD.flatMap(
    lambda result: flatten_entries(result))

# Creating a local R-Tree to merge the skyline points
localRTree = RTree()
for entry in skylineRDD.toLocalIterator():
    localRTree.insert(entry)
localSkylinePoints, time_localSkyline = localRTree.bbs_skyline()
```

Στο σχήμα 5.3 παρουσιάζεται σχηματικά η διαδικασία για την εύρεση κορυφογραμμής σε κατανεμημένο περιβάλλον.



Σχήμα 5.3: *Distributed Skyline Search*

Κεφάλαιο 6

Πειραματική αξιολόγηση

Σε αυτό το κεφάλαιο πραγματοποιείται η πειραματική αξιολόγηση των υλοποιήσεων που παρουσιάστηκαν μέχρι τώρα. Συγκεκριμένα θα συγκριθούν η κεντρικοποιημένη υλοποίηση του R-Tree, καθώς και η κατανεμημένη υλοποίηση του στο περιβάλλον Spark.

6.1 Πειραματικό Περιβάλλον

Για την υλοποίηση της κατανεμημένης δομής χρησιμοποιήθηκε η έκδοση 3.1.2 του Apache Spark σε εικονική συστάδα ενός υπολογιστή. Το λειτουργικό σύστημά του ήταν Windows 10, με 16GB Ram και επεξεργαστή Intel(R) Core(TM) i7-4790K χρονισμένο στα 4.00GHz. Ο επεξεργαστής διαθέτει 4 πυρήνες και 8 νήματα. Σε κάθε εκτέλεση του προγράμματος το αρχείο εισόδου διαιρείται σε 8 τμήματα και το Spark είναι ρυθμισμένο να αξιοποιεί όλα τα νήματα.

6.2 Πειραματικά Δεδομένα

Για την αξιολόγηση των πειραμάτων δημιουργήθηκαν 4 διαφορετικά σύνολα δεδομένων με την βοηθητική βιβλιοθήκη random της Python. Συγκεκριμένα κατασκευάστηκαν 4 αρχεία, όπου το κάθε ένα περιέχει 100.000, 500.000, 1.000.000 και 5.000.000 σημεία αντίστοιχα. Όλα τα σημεία τους είναι μοναδικά. Για κάθε συντεταγμένη x, y, z δόθηκε ένα εύρος τιμών από το οποίο κάθε τιμή μπορούσε να επιλεχθεί μόνο μία φορά χρησιμοποιώντας την μέθοδο της βιβλιοθήκης random, sample(). Τέλος, τα αποτελέσματα γράφτηκαν σε ένα αρχείο csv με τις βιβλιοθήκη csv.

6.3 Τρόπος χρονομέτρησης

Για την χρονομέτρηση των διαφόρων λειτουργιών χρησιμοποιήθηκε η βιβλιοθήκη time της Python, και πιο συγκεκριμένα η μέθοδος perf_counter(). Η μέθοδος αυτή επιστρέφει μία τιμή ενός μετρητή απόδοσης, δηλαδή ενός ρολογιού με την υψηλότερη διαθέσιμη ανάλυση για την μέτρηση μιας σύντομης διάρκειας. Το αρχικό σημείο αναφοράς της πρώτης κλήσης της

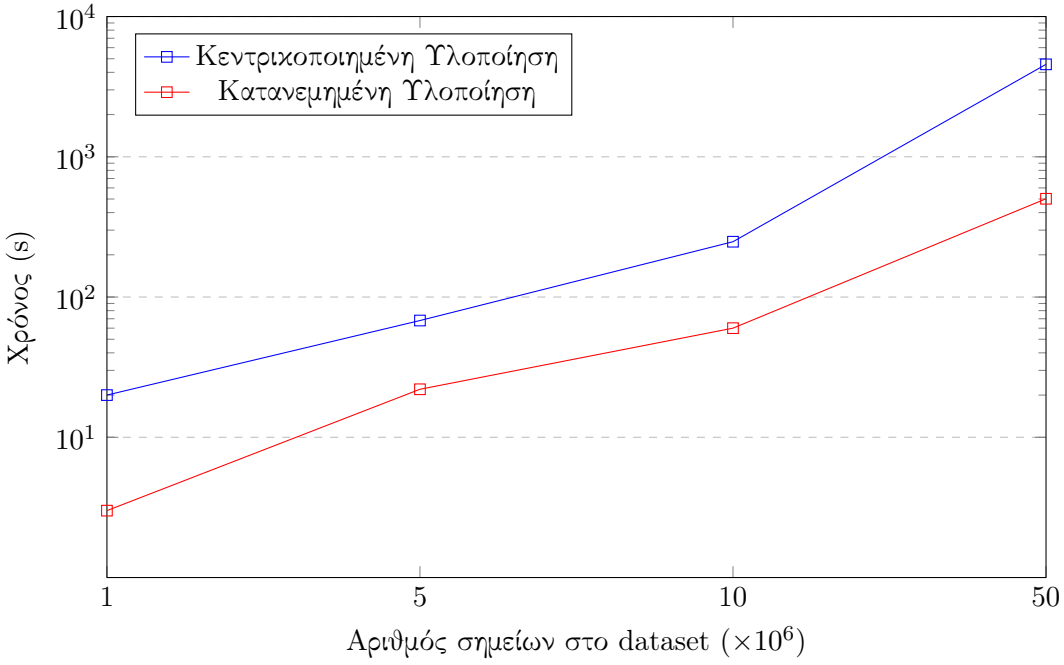
είναι απροσδιόριστο, επομένως η τιμή της έχει μόνο νόημα ως διαφορά μεταξύ δύο διαδοχικών κλήσεών της. Παρακάτω δίνεται κώδικας με παράδειγμα χρήσης της βιβλιοθήκης.

```
from time import perf_counter

start = perf_counter()
rtree.import_csv(dataset)
end = perf_counter()

# Time elapsed creating an R-Tree
timeElapsed = end - start
```

6.4 Πειραματικά αποτελέσματα

6.4.1 Κατασκευή R-Tree

Για την κατασκευή του R-Tree χρησιμοποιήθηκαν τέσσερα διαφορετικά dataset, τα οποία δόθηκαν ως είσοδο πέντε φορές το καθένα. Κάθε φορά τα σημεία ακολουθούσαν διαφορετική τυχαία κατανομή. Για τα αποτελέσματα πάρθηκε ο μέση τιμή του χρόνου κατασκευής. Η κατανεμημένη υλοποίηση ήταν γρηγορότερη σε κάθε περίπτωση.



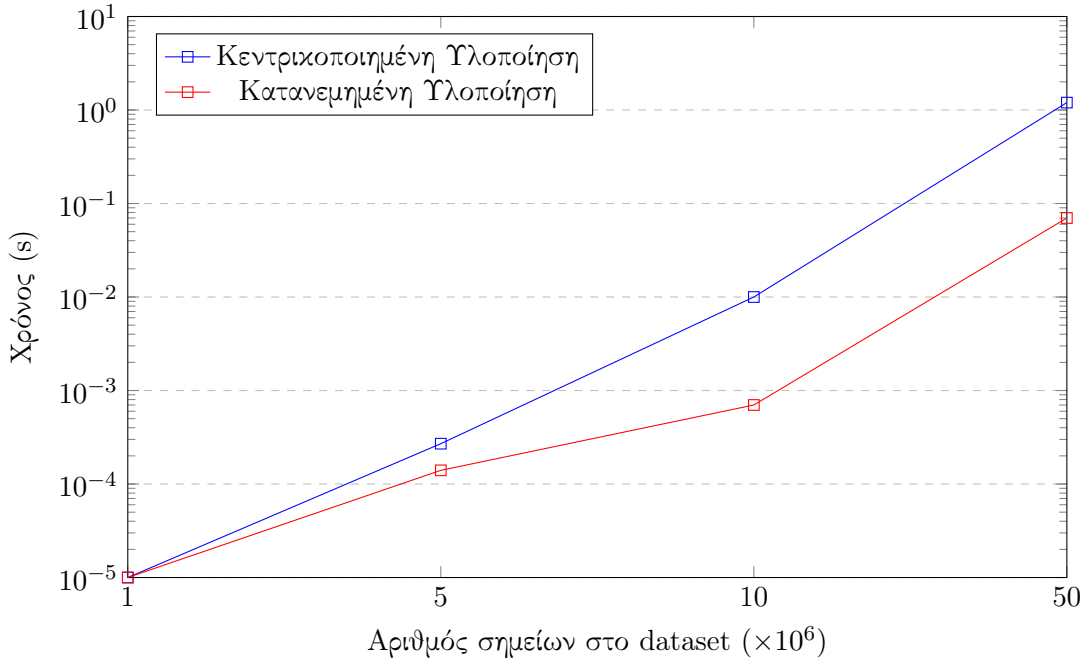
Σχήμα 6.1: *Χρόνος κατασκευής R-Tree*

Αριθμός σημείων στο dataset ($\times 10^6$)	**Κεντρικοποιημένη υλοποίηση (sec)**	**Κατανεμημένη υλοποίηση (sec)**
1	20	3
5	68	22
10	248	60
50	4562	502

Πίνακας 6.1: *Χρόνος κατασκευής R-Tree*

6.4.2 Αναζητήσεις εύρους

Για τα πειράματα αναζήτησεις εύρους έγιναν σε κάθε dataset 500 τυχαίες αναζητήσεις εύρους. Τα τελικά αποτελέσματα είναι η μίση τιμή των 500 αναζητήσεων. Είναι εμφανές ότι η διαφορά των δύο υλοποιήσεων γίνεται όλο και πιο ξεκάθαρη καθώς αυξάνεται το μέγεθος των δεδομένων.



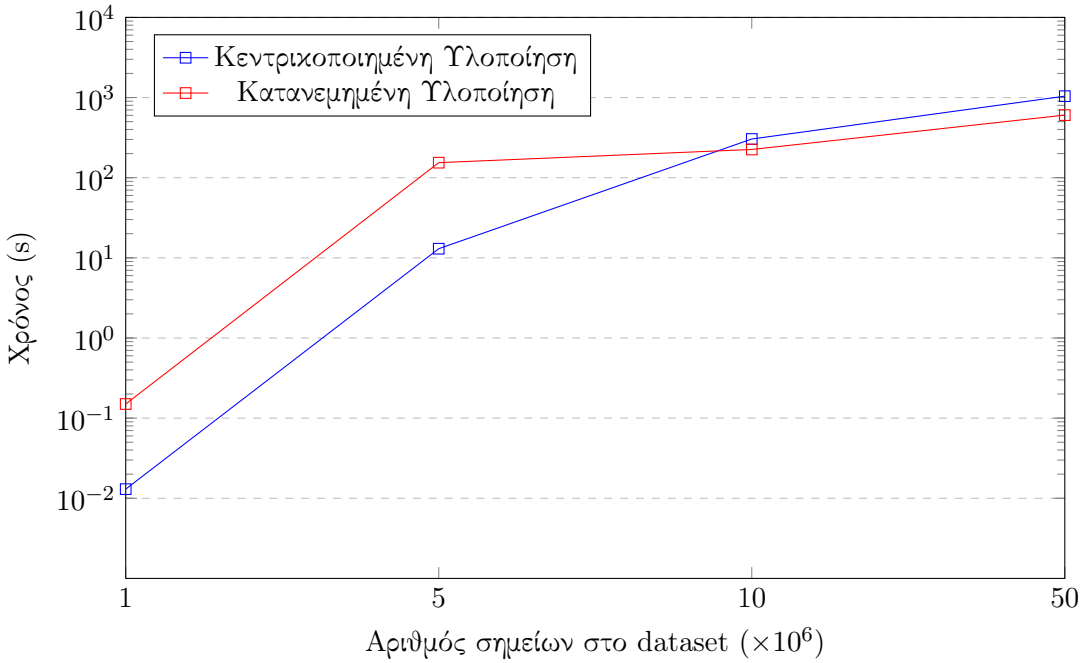
Σχήμα 6.2: *Χρόνος αναζήτησης εύρους*

Αριθμός σημείων στο dataset ($\times 10^6$)	**Κεντρικοποιημένη υλοποίηση (sec)**	**Κατανεμημένη υλοποίηση (sec)**
1	0.00001	0.00001
5	0.00027	0.00014
10	0.01	0.0007
50	1.2	0.07

Πίνακας 6.2: *Χρόνος αναζήτησης εύρους*

6.4.3 Εύρεση κορυφογραμμής

Για το πείραμα της εύρεσης της κορυφογραμμής, χρησιμοποιήθηκαν dataset τεσσάρων διαφορετικών μεγεθών, όπου το κάθε ένα είχε διαφορετική κατανομή κάθε φορά. Η διαδικασία αυτή έγινε δέκα φορές. Είναι εμφανές ότι στα μικρότερα σύνολα δεδομένων, η κεντρικοποιημένη υλοποίηση είναι αποδοτικότερη. Αυτό οφείλεται στο γεγονός ότι σε κάθε partition στο Spark, υπολογίζονται και αντικείμενα, τα οποία δεν ανήκουν στην τελική κορυφογραμμή. Επιπλέον, χρειάζεται τοπικά να ξανά δημιουργηθεί ένα καινούργιο R-Tree για την τελική κορυφογραμμή, κάτι που από μόνο του έχει μεγάλο κόστος. Η κατανεμημένη υλοποίηση έχει καλύτερη απόδοση όταν η κεντρικοποιημένη υλοποίηση ξεκινάει να αντιμετωπίζει προβλήματα μνήμης.



Σχήμα 6.3: *Χρόνος εύρεσης κορυφογραμμής*

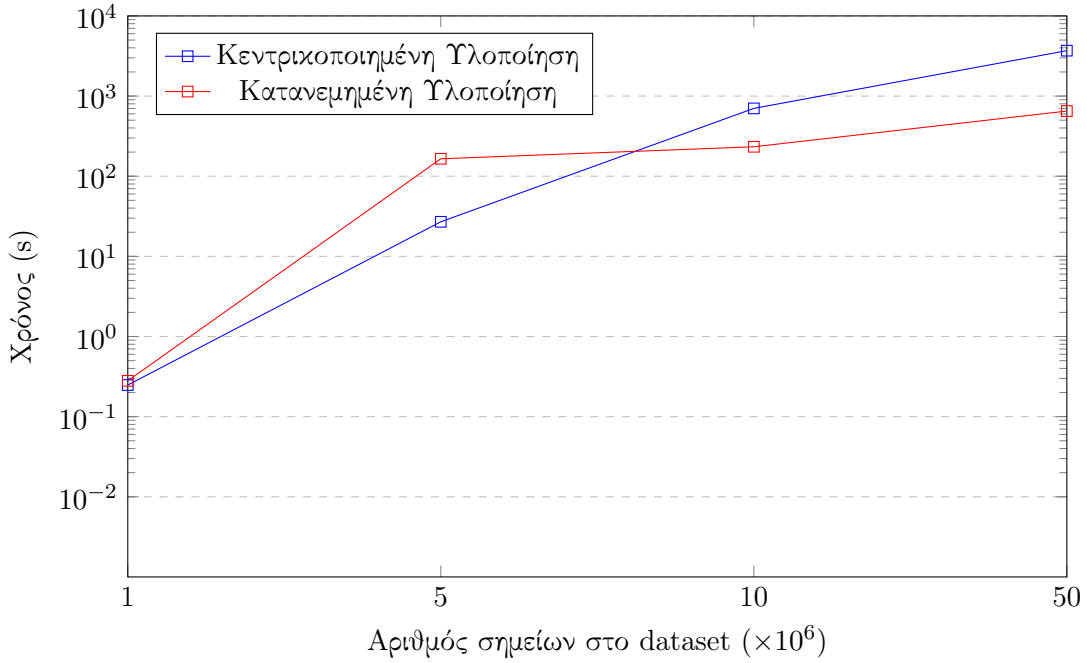
Αριθμός σημείων στο dataset ($\times 10^6$)	**Κεντρικοποιημένη υλοποίηση (sec)**	**Κατανεμημένη υλοποίηση (sec)**
1	0.013	0.15
5	13	154
10	304	225
50	1041	604

Πίνακας 6.3: *Χρόνος εύρεσης κορυφογραμμής*

6.4.4 Εύρεση κορυφογραμμής σε συγκεκριμένο εύρος

Η μεθοδολογία που ακολουθήθηκε είναι η ίδια με την υποενότητα 6.4.3. Όπως και σε εκείνη την περίπτωση, έτσι και εδώ η κατανεμημένη υλοποίηση είναι αποδοτικότερη μόνο όταν

η κεντρικοποιημένη υλοποίηση αντιμετωπίζει προβλήματα μνήμης. Ωστόσο, μία ενδιαφέρουσα παρατήρηση είναι το γεγονός ότι η εύρεση κορυφογραμμής με αυτήν σε συγκεκριμένο εύρος στην κατανεμημένη υλοποίηση έχουν πολλή κοντινούς χρόνους εκτέλεσης. Κάτι το οποίο δεν ισχύει και στην κεντρικοποιημένη υλοποίηση.



Σχήμα 6.4: *Χρόνος εύρεσης κορυφογραμμής σε συγκεκριμένο εύρος*

Αριθμός σημείων στο dataset ($\times 10^6$)	**Κεντρικοποιημένη υλοποίηση (sec)**	**Κατανεμημένη υλοποίηση (sec)**
1	0.248	0.28
5	27	165
10	702	233
50	3687	650

Πίνακας 6.4: *Χρόνος εύρεσης κορυφογραμμής σε συγκεκριμένο εύρος*

Κεφάλαιο 7

Συμπεράσματα και προοπτικές

Στα πλαίσια αυτής της διπλωματικής υλοποιήθηκαν στην γλώσσα προγραμματισμού Python το χωρικό ευρετήριο R-Tree, και στο κατανεμημένο περιβάλλον Spark υλοποιήθηκε η κατανεμημένη εκδοχή του. Μαζί με τις βασικές λειτουργίες του R-Tree υλοποιήθηκαν μέθοδοι οπτικοποίησης του δέντρου όπως και η αναζήτηση κορυφογραμμής για την οποία παρουσιάστηκαν οι προκλήσεις για την κατανεμημένη υλοποίηση της.

7.1 Συμπεράσματα

Στο κεφάλαιο 6 είδαμε ότι η κατανεμημένη υλοποίηση έχει σαφώς καλύτερη απόδοση όσο μεγαλώνει το αρχικό σύνολο δεδομένων σε ερωτήματα εύρους. Υπάρχουν ελάχιστες υλοποιήσεις του R-Tree σε κατανεμημένο περιβάλλον, και σε συνδυασμό με την πολύ καλή του απόδοση σε μεγάλο όγκο δεδομένων μπορούμε να συμπεράνουμε ότι είναι εμφανές ότι αξίζει να μελετηθεί η δομή σε μεγαλύτερο βάθος.

Αξίζει επίσης να σημειωθεί και η αναζήτηση κορυφογραμμής στο κατανεμημένο R-Tree. Αν και πιο αποδοτική από την κεντρικοποιημένη υλοποίηση στα μεγαλύτερα σύνολα δεδομένων, η ανάγκη για επαρκή μνήμη στο Master Node του cluster δεν πρέπει να αγνοηθεί. Σε μεγαλύτερα σύνολα δεδομένων θα πρέπει να χρησιμοποιηθούν κάποιες τεχνικές μείωσης των πιθανών σημείων που επιστρέφει το cluster.

7.2 Μελλοντικές Προεκτάσεις

Σύμφωνα με όλα τα παραπάνω που είδαμε, έγιναν εμφανές κάποια προβλήματα με την παρούσα υλοποίηση και τα περιθώρια βελτίωσης που υπάρχουν ακόμη για ένα αποδοτικότερο κατανεμημένο R-Tree. Παρακάτω παραθέτονται μερικές ιδέες και πιθανές λύσεις σε μερικά από αυτά τα προβλήματα.

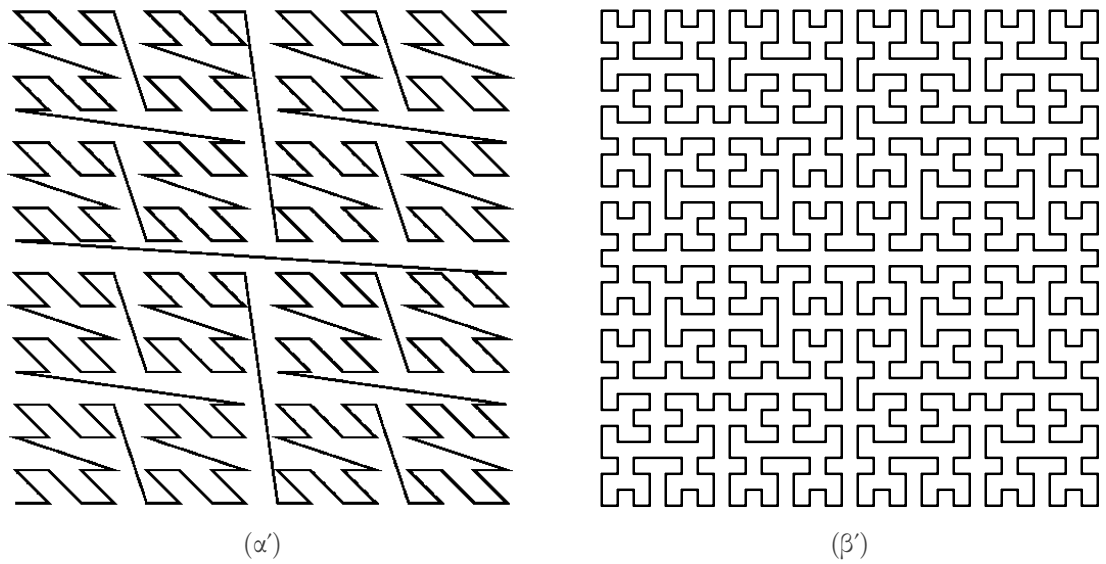
7.2.1 Μαζική φόρτωση δεδομένων

Σε ένα μεγάλο σύνολο δεδομένων η πρώτη βελτίωση στην απόδοση του R-Tree θα μπορούσε να έρθει υπό την μορφή της μαζικής φόρτωσης δεδομένων (**bulk loading**). Ερευνητές έχουν προτείνει αλγορίθμους οι οποίοι κατασκευάζουν ένα R-Tree τοποθετώντας άμεσα τα αντικείμενα δεδομένων στους κόμβους φύλλων αντί για την μεμονωμένη εισαγωγή τους [11] [12] [13]. Οι περισσότεροι από αυτούς βασίζονται σε κάποια διάταξη των δεδομένων. Για παράδειγμα στην έρευνα του Ρουσσόπουλου και Leifker [14], τα δεδομένα ταξινομούνται σύμφωνα με την συντεταγμένη του x άξονα.

7.2.2 Διάταξη Δεδομένων

Σε ένα κατανεμημένο περιβάλλον η διάταξη των δεδομένων δεν παίζει ρόλο μόνο στην δημιουργία του δέντρου, αλλά και στον διαμοιρασμό τους στους διάφορους κόμβους της συστάδας. Οι τρεις κύριες προσεγγίσεις αναδιάταξης δεδομένων για την παρούσα υλοποίηση είναι οι εξής:

- **Καμπύλες Πλήρωσης Χώρου (Space Filling Curves)**
  Η ιδέα των space filling curves είναι τα δεδομένα που έχουν μεγάλη σχέση μεταξύ τους να είναι κοντά μεταξύ τους στα δέντρα που θα προκύψουν μετά τον διαχωρισμό τους. Ως αποτέλεσμα σε μικρά queries ο φόρτος στην συστάδα είναι μικρός μιας και λίγοι υπολογιστές χρειάζεται να κάνουν κάποια αναζήτηση. Οι δύο πιο σύνηθες καμπύλες είναι η Z-order curve [15] και η Hilbert curve [16]. Παρακάτω δίνονται δύο σχήματα που αντιστοιχούν σε αυτές τις καμπύλες.



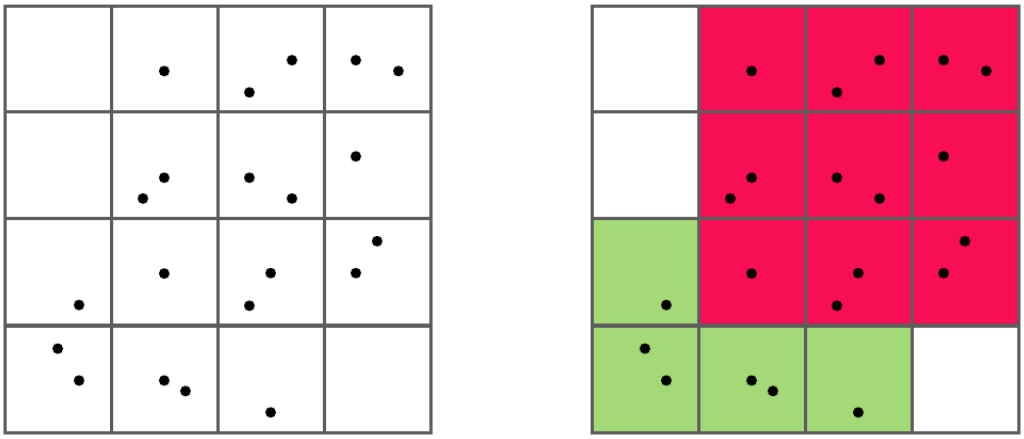
(α′) (β′)

Σχήμα 7.1: *(α′) Z-order curve, (β′) Hilbert curve*

- **Μη Γειτονική Κατανομή Δεδομένων (Non-adjacent Data Distribution)**
  Σε διατάξεις δεδομένων που διατηρείτε η αρχή της γειτονικότητα το κατανεμημένο

σύστημα έχει μειωμένη απόδοση σε ερωτήματα μεγάλου εύρους μιας και χάνετε ποσοστό του παραλληλισμού. Για αυτόν τον λόγο, ανάλογα με την χρήση του ευρετηρίου θα μπορούσε να χρησιμοποιηθεί συνάρτηση διάταξης δεδομένων η οποία θα έβαζε σε partitions τα δεδομένα με την μικρότερη σχέση μεταξύ τους.

- **Διαχωρισμός Πλέγματος (Grid Partitioning)**
  Τέλος, το μεγάλο κόστος μνήμης στο master node του αλγορίθμου εύρεσης κορυφογραμμής, έκανε εμφανές την ανάγκη περιορισμού των δεδομένων που επιστρέφουν τα R-Trees στην κύρια μνήμη για τον τελικό υπολογισμό του Skyline. Ένας τρόπος αντιμετώπισης αυτού του προβλήματος είναι η διάσπασει του αρχικού συνόλου δεδομένων σε ίδιου μεγέθους πλέγματα (grids). Στην συνέχεια τα πλέγματα που έχουν την μεγαλύτερη πιθανότητα να περιέχουν σημεία κορυφογραμμής να εισάγονται στο ίδιο partition του RDD. Ως αποτέλεσμα, τα σημεία που θα επιστρεφόταν θα είχαν αρκετά μεγαλύτερη πιθανότητα να ανήκουν και στο τελικό skyline. Δίνεται σχηματικά η ιδέα του αλγορίθμου παρακάτω.



(α′) (β′)

Σχήμα 7.2: *(α′) Δεδομένα χωρισμένα σε πλέγματα, (β′) Partitioning των πλεγμάτων σύμφωνα με τις πιθανότητες να περιέχουν Skyline points*

Βιβλιογραφία

[1] Z. Xu. *Cloud-Sea Computing Systems: Towards Thousand-Fold Improvement in Performance per Watt for the Coming Zettabyte Era*. *Computer Architecture and Systems*, 29(2):117–181, 2014.

[2] A. Guttman. *R-trees: a dynamic index structure for spatial searching*. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[3] D. Papadias, Y. Tao, G. Fu και B. Seeger. *Progressive skyline computation in database systems*. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[4] J. Dean και S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. *Communications of the ACM*, 51(1):107–113, 2008.

[5] *Matplotlib*. `https://matplotlib.org/`. Ημερομηνία πρόσβασης: 29-6-2021.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker και I. Stoica. *Spark: Cluster Computing with Working Sets*. *2nd USENIX Workshop on Hot Topiics in Cloud Computing*, Boston, MA, United States, 2010.

[7] *Apache Spark History*. `https://spark.apache.org/history.html`. Ημερομηνία πρόσβασης: 29-6-2021.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker και I. Stoica. *Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing*. *NSDI'12: Proceedings of the 9th USENIX conference on Networked SYstems Design and Implementation*, San Jose, CA, United States, 2012.

[9] I. Kamel και C. Faloutsos. *Parallel R-trees*. *ACM SIGMOD Record*, 21(2):195–204, 1992.

[10] L. Zhu, Y. Tao και S. Zhou. *Distributed Skyline Retrieval with Low Bandwidth Consumption*. *IEEE Transactions on Knowledge and Data Engineering*, 21(3):384–400, 2009.

[11] H. Haverkort και F. V. Walderveen. *Four-dimensional hilbert curves for R-trees*. *ACM Journal of Experimental Algorithmics*, 16(1):63–73, 2008.

[12] I. Kamel και C. Faloutsos. *Hilbert R-tree: An Improved R-tree using Fractals*. *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, σελίδες 500–509, 1994.

[13] S.T. Leutenegger, M. A. Lopez και J. Edgington. *STR: a simple and efficient algorithm for R-tree packing*. *Proceedings 13th International Conference on Data Engineering*, Birmingham, UK, 1997.

[14] N. Roussopoulos και D. Leifker. *Direct spatial search on pictorial databases using packed R-trees*. *ACM SIGMOD Record*, 14(4):17–31, 1985.

[15] J. A. Orenstein. *Spatial query processing in an object-oriented database system*. *ACM SIGMOD Record*, 15(2):326–336, 1986.

[16] D. Hilbert. *Ueber die stetige Abbildung einer Line auf ein Flächenstück*. *Mathematische Annalen*, 38:459–460, 1891.