ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

«Δεικτοδότηση μεγάλης κλίμακας δεδομένων για υποστήριξη χωρικών επερωτημάτων στο Hadoop DFS»

«Indexing large-scale data to support spatial queries on Hadoop DFS»

Σοφία Γεωργιακάκη

Διπλωματική Εργασία

Επιβλέπων Καθηγητής: Επ. Καθηγητής Αντώνιος Δεληγιαννάκης

Εξεταστική Επιτροπή: Επ. Καθηγητής Αντώνιος Δεληγιαννάκης
Καθηγητής Μίνως Γαροφαλάκης
Επ. Καθηγητής Βασίλης Σαμολαδάς

Χανιά, Νοέμβριος 2011

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially,
because it produces objects of beauty.

Prof. Donald E. Knuth

Ευχαριστίες

Για τους ανθρώπους που με βοήθησαν, με τον ένα ή τον άλλο τρόπο, κατά την εξέλιξη της εργασίας αυτής αλλά και καθ' όλη τη διάρκεια των σπουδών μου, αυτές οι γραμμές δεν είναι αρκετές.

Η επιτυχής ολοκλήρωση της παρούσας εργασίας δεν θα ήταν εφικτή χωρίς την στήριξη ανθρώπων γύρω μου, στους οποίους είμαι ευγνώμων για τον χρόνο που αφιέρωσαν και την ενθάρρυνση που μου παρείχαν.

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Αντώνη Δεληγιαννάκη, για την υποστήριξη και καθοδήγηση που μου προσέφερε κατά την εξέλιξη της παρούσας διπλωματικής εργασίας καθώς και για την επίτευξη μιας *άψογης* συνεργασίας.

Τις ειλικρινείς μου ευχαριστίες στην υπεύθυνη τεχνικό του εργαστηρίου Softnet κα. Πολυξένη Αράπη και στον μεταπτυχιακό φοιτητή του τμήματος ΗΜΜΥ Βαγγέλη Βαζαίο, για την πολύ σημαντική βοήθεια σε τεχνικές δυσκολίες ή «προγραμματιστικά διλήμματα» που παρουσιάστηκαν.

Επιπλέον, ευχαριστώ πολύ την μεταπτυχιακή φοιτήτρια του τμήματος ΗΜΜΥ Καλλιόπη Καλαντζάκη και τον διδακτορικό φοιτητή Στέργιο Χαραλαμπάκη για την βοήθειά τους στα πρώτα βήματα ανάπτυξης της εργασίας αυτής.

Η ολοκλήρωση ενός κύκλου σπουδών δεν θα σήμαινε τίποτα για μένα ως άνθρωπο, αν δεν συνοδευόταν από μια ζωή με πολλούς καλούς φίλους και άξιους ανθρώπους κοντά μου. Ευχαριστώ πολύ τη Μαρικίτα, τη Βάννα, τη Δώρα, τη Γεωργία και τη Μαρία που υπήρξαν φίλοι αναντικατάστατοι, και το Θανάση που ήταν πάντα δίπλα μου.

Η εργασία αυτή αφιερώνεται εξ' ολοκλήρου στην οικογένειά μου, με όλη την αγάπη μου. Όλος ο σεβασμός και η ευγνωμοσύνη μου ανήκουν σ' αυτούς.

Περίληψη

Ποικίλα συστήματα καλούνται να διαχειριστούν μεγάλο όγκο πολυδιάστατων δεδομένων. Οι δομές δεικτοδότησης R-trees μπορούν να επιταχύνουν ένα φάσμα επερωτημάτων, αποφεύγοντας τη σάρωση όλων των δεδομένων. Δεδομένου του όγκου των δεδομένων και της πολυπλοκότητας των επερωτημάτων, υλοποιήσαμε έναν αλγόριθμο για την κατασκευή μιας ιεραρχικής στατικής δομής δεικτοδότησης τύπου Packed Hilbert R-tree στο κατανεμημένο σύστημα αρχείων Hadoop DFS. Επιπλέον, σχεδιάσαμε και υλοποιήσαμε τρεις διαφορετικές στρατηγικές εκτέλεσης δέσμης τοπολογικών χωρικών επερωτημάτων (Range & Point Queries) στο περιβάλλον Hadoop MapReduce, χρησιμοποιώντας το αποθηκευμένο στο HDFS R-tree. Η πειραματική αξιολόγηση δείχνει την καλή απόδοση του αλγορίθμου κατασκευής R-tree, και την δυνατότητα των μεθόδων εκτέλεσης επερωτημάτων να ανταπεξέλθουν σε δέσμες επερωτημάτων σε διαφορετικές συνθήκες, ανάλογα με τη φύση των επερωτημάτων και το μέγεθος του cluster.

Λέξεις Κλειδιά: χωρικά δεδομένα, πολυδιάστατο ευρετήριο, Packed Hilbert R-tree, Hadoop MapReduce, HDFS, επεξεργασία χωρικών επερωτημάτων, Range Queries.

Abstract

Various systems need to handle massive amounts of multidimensional data. R-tree indexing structures can speed up a range of queries, by avoiding a full scan of the data set. Considering the large amount of data and the complexity of the queries, we implemented an algorithm for the construction of a hierarchical static Packed Hilbert R-tree indexing structure on the Hadoop distributed filesystem (HDFS). Moreover, we designed and implemented 3 different strategies for processing a batch of topological spatial queries (Range & Point Queries) using the Hadoop MapReduce framework, exploiting the R-tree that resides on the HDFS. The experimental evaluation shows the good performance of the algorithm that builds the R-tree, and the ability of the 3 query processing methods to handle batches of queries under different circumstances regarding the form of the queries and the size of the cluster.

Keywords: spatial data, multidimensional index, Packed Hilbert R-tree, Hadoop MapReduce, HDFS, spatial query processing, Range Queries.

Πίνακας Περιεχομένων

1. Εισαγωγή **8**

1.1 Αντικείμενο & συνεισφορά 8

1.2 Δομή της εργασίας 10

2. Θεωρητικό Υπόβαθρο **12**

2.1 Διαχείριση Χωρικών Δεδομένων (Spatial Data Management) 12

2.2 Δεικτοδότηση Χωρικών Δεδομένων με R-trees 13

2.2.1 Η Δομή των R-trees για Δυναμικά και Στατικά Δεδομένα 14

1.2.2 Packed Hilbert Rtree για Στατικά Πολυδιάστατα Δεδομένα 16

2.3 Επεξεργασία Χωρικών Επερωτημάτων (Spatial Query Processing) 18

2.3.1 Τοπολογικά Επερωτήματα (Range and Topological Queries) 20

2.3.2 Βελτιστοποίηση Εκτέλεσης Επερωτημάτων με Παραλληλοποίηση 21

3. Hadoop και MapReduce **25**

3.1 Εισαγωγή 25

3.2 Hadoop MapReduce – Προγραμματιστικό Μοντέλο 27

3.3 Hadoop MapReduce - Επίπεδο Ροής Δεδομένων 29

3.4 Hadoop Distributed FileSystem – HDFS 34

3.5 Hadoop Concepts 35

4. Κατασκευή Packed Hilbert R-tree σε περιβάλλον Hadoop MapReduce **37**

4.1 Διάρθρωση της υλοποίησης 37

4.2 Η Βοηθητική Βιβλιοθήκη Spatial Index 37

4.2.1 Προϋπάρχουσα Λειτουργικότητα: Πακέτο SpatialIndex 39

4.2.2 Προϋπάρχουσα Λειτουργικότητα: Πακέτο Storage Manager 41

4.2.3 Προϋπάρχουσα Λειτουργικότητα: Πακέτο rtree 45

4.2.4 Πρόσθετη Λειτουργικότητα: Υλοποίηση Packed Hilbert R-tree στο δίσκο. 48

4.2.5 Πρόσθετη Λειτουργικότητα: Υλοποίηση Storage Manager για αποθήκευση στο HDFS. 53

4.3 Υλοποίηση Κατασκευής Packed Hilbert R-tree σε περιβάλλον Hadoop 59

4.3.1 Εισαγωγή - Γενική περιγραφή του αλγορίθμου 60

4.3.2 Ανάλυση της υλοποίησης 63

5. Αλγόριθμοι εκτέλεσης Χωρικών Επερωτημάτων με χρήση του Packed Hilbert R-tree 75

5.1 Ανάλυση Σχεδιασμού: Λειτουργικότητα και απαιτήσεις 75

5.2 Στρατηγική 1η: Εκτέλεση επερωτημάτων με κατανομή τους στο δέντρο που τα απαντά 77

5.2.1 Κίνητρα και γενική ιδέα του αλγορίθμου 77

5.2.2 Περιγραφή της υλοποίησης 78

5.3 Στρατηγική 2η: Εκτέλεση επερωτημάτων με ισοκατανομή τους στις διεργασίες επεξεργασίας. 85

5.3.1 Κίνητρα και γενική ιδέα του αλγορίθμου 85

5.3.2 Περιγραφή της υλοποίησης 86

5.4 Στρατηγική 3η: Εκτέλεση επερωτημάτων με προεπεξεργασία 88

5.4.1 Κίνητρα και γενική ιδέα του αλγορίθμου 88

5.4.2 Περιγραφή πρώτου σταδίου της υλοποίησης: Προεπεξεργασία των επερωτημάτων 91

5.4.3 Περιγραφή δεύτερου σταδίου της υλοποίησης: Εκτέλεση των επερωτημάτων 95

6. Πειραματική Αξιολόγηση 99

6.1 Μεθοδολογία των πειραμάτων και δεδομένα εισόδου 99

6.2 Πειραματική αξιολόγηση του αλγορίθμου κατασκευής Packed Hilbert R-tree στο HDFS 101

6.3 Πειραματική αξιολόγηση των αλγορίθμων εκτέλεσης επερωτημάτων 106

6.3.1 Εκτέλεση μιας δέσμης «ομοιόμορφων», ως προς τα R-trees, επερωτημάτων 108

6.3.2 Εκτέλεση μιας δέσμης μη-ομοιόμορφων, ως προς τα R-trees, επερωτημάτων 115

7. Συμπεράσματα και Μελλοντικές επεκτάσεις 119

1. Εισαγωγή

1.1 Αντικείμενο & συνεισφορά

Επιστημονικές περιοχές που ασχολούνται με την μελέτη και απεικόνιση του κόσμου μας, παράγουν και επεξεργάζονται συνεχώς χωρικά δεδομένα (spatial data). Τα δεδομένα αυτής της μορφής περιλαμβάνουν μονοδιάστατα ή πολυδιάστατα αντικείμενα, όπως π.χ. γραμμές, σημεία, ή ακόμα και κινούμενες τροχιές.

Χωρικά δεδομένα χρησιμοποιούνται ευρέως από εφαρμογές όπως συστήματα γεωγραφικών πληροφοριών (geographical information systems ή GIS), προγράμματα σχεδίασης, εφαρμογές μηχανικής όρασης ή ρομποτικής. Για γρήγορη πρόσβαση σε τέτοια δεδομένα, μια καλή τεχνική είναι η ανάπτυξη ευρετηρίων και δομών δεικτοδότησης όπως τα R-trees. Οι εφαρμογές των R-trees καλύπτουν ένα ευρύ φάσμα, όπως συστήματα για αποθήκευση και επεξεργασία χωρικών δεδομένων, δεδομένων με παράμετρο το χρόνο (temporal ή spatiotemporal data), βίντεο ή εικόνων. Οι διάφορες μορφές δεικτοδότησης R-tree βελτίωσαν τις τεχνικές αποδοτικής εκτέλεσης χωρικής αναζήτησης ή άλλων επερωτημάτων (spatial queries), αποφεύγοντας να σαρώσουν ολόκληρο το σύνολο δεδομένων. Παραδείγματα τέτοιων χωρικών επερωτημάτων είναι τα τοπολογικά επερωτήματα που εξετάζουν την ικανοποίηση μιας χωρικής σχέσης (π.χ. ανεύρεση των αντικειμένων που επικαλύπτουν ή περιέχουν ένα δοσμένο αντικείμενο) ή επερωτήματα όπου λαμβάνει χώρα η πράξη της σύζευξης (Spatial Join Queries).

Καθώς αυξάνεται ο όγκος των χωρικών δεδομένων και η πολυπλοκότητα των απαιτούμενων υπολογισμών για την ανάλυσή τους, τα παραδοσιακά μοντέλα επεξεργασίας μέσω Σχεσιακών Συστημάτων Βάσεων δεδομένων (RDBMSs) ή απλών σειριακών εφαρμογών αποδεικνύονται ανεπαρκή. Η παραλληλοποίηση των διεργασιών είναι μια στρατηγική που θα μπορούσε να επιταχύνει την αποθήκευση και ευρετηριοποίηση των δεδομένων. Επιπλέον, κατανεμημένα περιβάλλοντα θα μπορούσαν να αναλάβουν λειτουργικότητα όπως την εκτέλεση χωρικών επερωτημάτων.

Υπό αυτές τις συνθήκες, ένα ενδιαφέρον ερευνητικό θέμα είναι το να εξετάσουμε τη δυνατότητα για κατανεμημένη κατασκευή ενός R-tree χρησιμοποιώντας το περιβάλλον Hadoop. Το Hadoop Map/Reduce είναι ένα framework λογισμικού γραμμένο σε Java για την ανάπτυξη εφαρμογών παράλληλης επεξεργασίας μεγάλου όγκου δεδομένων (της τάξης δεκάδων Gigabytes/Terabytes) σε υπολογιστές συνδεδεμένους μεταξύ τους μέσω δικτύου (cluster υπολογιστών), με τρόπο αξιόπιστο και ανεκτικό στις αστοχίες. Στο Hadoop η διαχείριση της αποθήκευσης παρέχεται από το σύστημα αρχείων HDFS (Hadoop

Distributed FileSystem) και οι αλγόριθμοι αναπτύσσονται χρησιμοποιώντας το προγραμματιστικό μοντέλο MapReduce.

Είναι σημαντικό να αναφέρουμε, ότι το Hadoop έχει αρχίσει να καθιερώνεται ως ένα εξαιρετικό framework για την ανάπτυξη εφαρμογών παράλληλα που εκτελούν υπολογιστικές εργασίες με τρόπο μαζικό σε όλο το σύνολο δεδομένων της εισόδου (batch-processing). Παρέχει βελτιστοποιημένες τεχνικές για σειριακή ανάγνωση/εγγραφή μεγάλων blocks δεδομένων και για το λόγο αυτό πολλές φορές αντενδείκνυται για εφαρμογές low-latency όπου η επεξεργασία είναι πιο στοχευμένη και αφορά ένα μικρό μέρος των διαθέσιμων δεδομένων [27]. Ωστόσο, η ισχύς που δίνει το Hadoop για την εύκολη ανάπτυξη παράλληλων αλγορίθμων που προσπελαύνουν τεράστια σύνολα δεδομένων καθώς και η αυτοποιημένη διαχείριση αστοχιών, οδηγούν πολλούς ερευνητές σε προσπάθειες να χρησιμοποιήσουν τα πλεονεκτήματα αυτά και σε ευρύτερες εφαρμογές.

Η ανάπτυξη δομών δεικτοδότησης R-trees και η χρήση τους για εκτέλεση επερωτημάτων, είναι ερευνητικά επιτεύγματα που εφαρμόστηκαν κατά κόρον στις χωρικές βάσεις δεδομένων ή σε συστήματα παράλληλων βάσεων δεδομένων. Και έχει ως εκ τούτου ενδιαφέρον η προοπτική να επεκταθούν και να προσαρμοστούν και στο μοντέλο Hadoop, ώστε να μπορούν να εφαρμοστούν εξίσου για ημιδομημένα ή μη-δομημένα χωρικά αντικείμενα και να εκμεταλλευτούν τα πλεονεκτήματα του Hadoop.

Καθώς το σύστημα αρχείων του Hadoop HDFS στηρίζει στατικά δεδομένα που γράφονται μια φορά και διαβάζονται πολλές, οι τεχνικές Bulk-loading για ανάπτυξη R-tree με μαζική φόρτωση δεδομένων φαίνεται να μπορούν να εφαρμοστούν. Πράγματι, νέες δημοσιεύσεις την τελευταία 2ετία επιχείρησαν διερεύνηση του θέματος, χωρίς φυσικά να το εξαντλούν. Ακόμα λιγότερο έχουν διερευνηθεί τα περιθώρια αποδοτικής εκτέλεσης χωρικών επερωτημάτων στο Hadoop MapReduce και ειδικότερα η παράλληλη εκτέλεση δέσμης επερωτημάτων χρησιμοποιώντας ένα R-tree που βρίσκεται αποθηκευμένο στο HDFS.

Αντικείμενο της παρούσας διπλωματικής είναι η διερεύνηση των 2 παραπάνω ζητημάτων.

Στα πλαίσια αυτά, σχεδιάστηκε και υλοποιήθηκε ένας αλγόριθμος για την κατασκευή ενός Packed Hilbert R-tree χρησιμοποιώντας το framework Hadoop. Το Packed Hilbert R-tree που σχεδιάσαμε δεικτοδοτεί πολυδιάστατα χωρικά δεδομένα ,τα οποία αναπαριστώνται από το ελάχιστο περιβάλλον ορθογώνιό τους (Minimum Bounding Rectangle ή MBR). Το δέντρο κατασκευάζεται με την τεχνική μαζικής φόρτωσης δεδομένων (bulk-loading) και αποθηκεύεται στο HDFS ενός cluster υπολογιστών. Για την παράλληλη δημιουργία του ευρετηρίου μας, χρησιμοποιήσαμε μια τεχνική γραμμικής ταξινόμησης των πολυδιάστατων ορθογωνίων που βασίζεται σε μορφοκλάσματα (space filling curves ή fractals) και την τιμή Hilbert, καταφέρνοντας έτσι να χωρίσουμε τα ορθογώνια προς

δεικτοδότηση σε ομάδες με κοντινά στο χώρο αντικείμενα. Οι ομάδες δεδομένων μοιράζονται στη συνέχεια ανάμεσα σε έναν αριθμό διαθέσιμων κόμβων παράλληλης επεξεργασίας του cluster και ένα R-tree χτίζεται για κάθε κόμβο. Με τον τρόπο αυτό έχουμε πλέον μια δομή Packed Hilbert R-tree που μπορεί να είναι αποτελεσματική για την εκτέλεση χωρικών επερωτημάτων.

Σε δεύτερη φάση σχεδιάστηκαν και υλοποιήθηκαν 3 διαφορετικοί αλγόριθμοι που εκτελούν μια δέσμη επερωτημάτων κατανεμημένα σε περιβάλλον Hadoop MapReduce, χρησιμοποιώντας ως βοηθητική δομή τα Packed Hilbert R-trees που βρίσκονται αποθηκευμένα στο HDFS. Οι αλγόριθμοι υποστηρίζουν την εκτέλεση τοπολογικών επερωτημάτων, και συγκεκριμένα ερωτημάτων εύρους ή Range Queries που στοχεύουν σε ένα μικρό μέρος ενός συνόλου χωρικών δεδομένων. Ο λόγος για την ανάπτυξη 3 αλγορίθμων ήταν η προσπάθειά μας να προτείνουμε αλγορίθμους που να μπορούν να ανταπεξέλθουν ικανοποιητικά σε διαφορετικές συνθήκες, ανάλογα με την φύση των επερωτημάτων, το μέγεθος του cluster, και τον βαθμό ομοιομορφίας των χωρικών δεδομένων στα οποία στοχεύει το κάθε query. Σε κάθε βήμα ανάπτυξης του εκάστοτε αλγορίθμου υπήρξε κριτήριο η όσο το δυνατόν καλύτερη δρομολόγηση των επερωτημάτων στους κόμβους επεξεργασίας, και η προσαρμογή της ιεραρχικής δομής των R-tree στις ιδιαιτερότητες του Hadoop και του HDFS. Έτσι, ο πρώτος αλγόριθμος δρομολογεί τα επερωτήματα στους κόμβους επεξεργασίας με κριτήριο το επιμέρους R-tree που εν δυνάμει τα απαντά. Ο δεύτερος αλγόριθμος δρομολογεί τα επερωτήματα με κριτήριο την ισοκατανομή τους στους κόμβους επεξεργασίας ανεξάρτητα από το επιμέρους R-tree που τα απαντά. Τέλος, η 3η στρατηγική χρησιμοποιεί προεπεξεργασία και τεχνικές δειγματοληψίας ώστε να διαμοιράσει τα επερωτήματα λαμβάνοντας υπόψιν και την ισοκατανομή στους κόμβους αλλά και το επιμέρους R-tree που απαντά κάθε query.

Τέλος, πραγματοποιήθηκαν πειράματα ώστε να αξιολογηθεί η απόδοση των υλοποιήσεών μας τόσο για την παράλληλη κατασκευή του Packed Hilbert R-tree, όσο και για την κατανεμημένη εκτέλεση μιας δέσμης χωρικών επερωτημάτων. Τα πειράματά μας έγιναν με τρόπο τέτοιο, ώστε να μετρηθεί η απόδοση της κάθε υλοποίησης ως προς το χρόνο εκτέλεσης, την εγκυρότητα των αποτελεσμάτων εξόδου και την επεκτασιμότητα (παράγοντες scale-up και size-up).

1.2 Δομή της εργασίας

Τα Κεφάλαια που ακολουθούν στην παρούσα εργασία είναι οργανωμένα ως εξής: Το Κεφάλαιο 2 παρουσιάζει το θεωρητικό υπόβαθρο της εργασίας, αναλύοντας την βασική θεωρία γύρω από τα χωρικά δεδομένα (2.1), την δεικτοδότησή τους με R-trees (2.2) και την εκτέλεση χωρικών επερωτημάτων (2.3).

Το Κεφάλαιο 3 αναλύει το περιβάλλον Hadoop. Παρουσιάζονται τα βασικά του χαρακτηριστικά και πλεονεκτήματα (3.1), το προγραμματιστικό περιβάλλον MapReduce (3.2), το επίπεδο ροής δεδομένων (3.3) και βασικές έννοιες του HDFS και του συστήματος γενικότερα (3.4 και 3.5) .

Το Κεφάλαιο 4 παρουσιάζει λεπτομερώς την υλοποίηση του αλγορίθμου κατασκευής ενός Packed Hilbert R-tree. Αρχίζει με την διάρθρωση της υλοποίησης (4.1), την επισκόπηση της βιβλιοθήκης Spatial Index (4.2) και καταλήγει στην τελική υλοποίηση του αλγορίθμου (4.3).

Το Κεφάλαιο 5 περιγράφει τον σχεδιασμό και την υλοποίηση των 3 στρατηγικών παράλληλης εκτέλεσης μιας δέσμης επερωτημάτων. Παρουσιάζεται η λειτουργικότητα και οι απαιτήσεις (5.1) και μετά οι 3 διαφορετικοί αλγόριθμοι (5.2, 5.3 και 5.4).

Το Κεφάλαιο 6 παρουσιάζει την πειραματική αξιολόγηση των υλοποιήσεων, για κατασκευή του T-tree (6.1) και για εκτέλεση δέσμης «ομοιόμορφων» και μη επερωτημάτων (6.2 και 6.3 αντίστοιχα).

Τέλος, το Κεφάλαιο 7 αναφέρει τα συμπεράσματα που προέκυψαν και τις μελλοντικές επεκτάσεις.

2. Θεωρητικό Υπόβαθρο

2.1 Διαχείριση Χωρικών Δεδομένων (Spatial Data Management)

Τα χωρικά δεδομένα (spatial data) απαντώνται σε πολλές εφαρμογές. Ανάμεσά τους, τα συστήματα γεωγραφικών πληροφοριών (geographical information systems ή GIS), οι εφαρμογές σχεδίασης με τη βοήθεια υπολογιστή (π.χ. CAD), η μηχανική όραση και η ρομποτική.

Πολλές εφαρμογές που διαχειρίζονται μεγάλα σύνολα χωρικών δεδομένων, απαιτούν μοντέλα για την αποδοτική διαχείριση γεωμετρικών αντικειμένων όπως σημεία, γραμμές, περιοχές ή όγκοι σε μια ή περισσότερες διαστάσεις. Τα συστήματα βάσεων δεδομένων που διαχειρίζονται τέτοιου είδους αντικείμενα έχουν καθιερωθεί με το όνομα χωρικές βάσεις δεδομένων (spatial databases) και διαθέτουν δομές δεδομένων ικανές να εκτελέσουν ποικίλα χωρικά επερωτήματα.

Τα διάφορα συστήματα γεωγραφικών πληροφοριών (GIS) και παρόμοιες εφαρμογές διαχειρίζονται σύνθετα και μεγάλα σε όγκο δεδομένα δύο κατηγοριών:

-δεδομένα σε μορφή εικόνας (raster data), όπως δορυφορικές ή εναέριες εικόνες που παράγονται π.χ. από ειδικούς αισθητήρες ή δορυφορικές λήψεις.

-δεδομένα σε μορφή διανύσματος (vector data), όπως σημεία, γραμμές, πολύγωνα που παράγονται π.χ. από εφαρμογές GPS.

Τα πληροφοριακά συστήματα που διαχειρίζονται αυτά τα δεδομένα, χρειάζονται δομές ώστε να ανταπεξέλθουν σε διάφορες απαιτήσεις λειτουργικότητας, όπως την εκτέλεση χωρικών επερωτημάτων.

Σήμερα, οι χωρικές βάσεις δεδομένων και τα ποικίλα συστήματα γεωγραφικών πληροφοριών έχουν παγιωθεί ως μια ώριμη ερευνητική περιοχή, καθώς η διαχείριση χωρικών δεδομένων (συμπεριλαμβανομένων δεδομένων με πρόσθετη παράμετρο το χρόνο, όπως κινούμενα αντικείμενα ή τροχιές) μελετάται εκτενώς και αναπτύσσονται συνεχώς ακαδημαϊκές εργασίες και εμπορικές εφαρμογές. Καθώς αυξάνεται ο όγκος των χωρικών δεδομένων και η πολυπλοκότητα των απαιτούμενων υπολογισμών για την ανάλυσή τους, τα παραδοσιακά μοντέλα επεξεργασίας μέσω Σχεσιακών Συστημάτων Βάσεων δεδομένων (RDBMSs) ή απλών εφαρμογών σε προσωπικούς υπολογιστές (CAD) δεν επαρκούν. Υπό αυτές τις συνθήκες, νέα υπολογιστικά μοντέλα παράλληλης

επεξεργασίας, όπως το Hadoop MapReduce, παρέχουν μια προοπτική για επεκτάσιμες (scalable) εφαρμογές διαχείρισης χωρικών δεδομένων.

2.2 Δεικτοδότηση Χωρικών Δεδομένων με R-trees

Για περισσότερες από 3 δεκαετίες, εξελίσσεται διαρκώς η υποστήριξη χωρικών αντικειμένων και επερωτημάτων σε διάφορα πληροφοριακά συστήματα (όπως τα σχεσιακά συστήματα βάσεων δεδομένων). Παρατηρείται μια διαρκής εξέλιξη των μοντέλων αναπαράστασης των χωρικών δεδομένων, των μηχανισμών δεικτοδότησης (indexes), αλλά και των αλγορίθμων για αποδοτική επεξεργασία επερωτημάτων. Το 1984, ο Antonin Guttman [1] πρότεινε την δομή R-tree ως μια αποδοτική μέθοδο δεικτοδότησης ορθογώνιων αντικειμένων. Από τότε έχουν προταθεί πολλές παραλλαγές της πρωτότυπης έκδοσης του R-tree που παρέχουν αποτελεσματική πρόσβαση και διαχείριση πολυδιάστατων χωρικών αντικειμένων [2]. Οι διάφορες μορφές δεικτοδότησης R-tree βελτίωσαν τις τεχνικές αποδοτικής εκτέλεσης επερωτημάτων (spatial queries), ενώ άρχισαν να υποστηρίζουν παράλληλη επεξεργασία (για μεγάλα σύνολα δεδομένων) ή μαζική φόρτωση δεδομένων στο δέντρο (bulk-loading).

Τα R-trees είναι δομές δεδομένων για τη δεικτοδότηση χωρικών δεδομένων όπως σημεία, γραμμές, επιφάνειες, όγκους ή γεωγραφικά δεδομένα περισσότερων από 2 διαστάσεων. Οι εφαρμογές των R-trees καλύπτουν ένα ευρύ φάσμα, όπως συστήματα για αποθήκευση και επεξεργασία χωρικών δεδομένων, δεδομένων με παράμετρο το χρόνο (temporal ή spatiotemporal data), βίντεο ή εικόνων.

Η απλή δομή των διαφόρων τύπων R-tree αλλά και η ομοιότητά τους με το μηχανισμό λειτουργίας των B-trees, επέτρεψε την εύκολη ενσωμάτωση των R-trees σε εφαρμογές διαχείρισης χωρικών δεδομένων, όπως τα Συστήματα Βάσεων Δεδομένων. Σιγά σιγά τεχνικές που εφαρμόζονταν στα B-trees άρχισαν να επεκτείνονται και στα R-trees, ώστε να βελτιώσουν την δεικτοδότηση και την εκτέλεση επερωτημάτων σε πολυδιάστατα δεδομένα.

Ο αριθμός των παραλλαγών R-trees που παρουσιάστηκαν στην επιστημονική βιβλιογραφία είναι μεγάλος, όπως περιγράφονται και στις δημοσιεύσεις [2], [3], [4], [5]. Για τις ανάγκες της παρούσας διπλωματικής, θα αναφέρουμε μόνο την βασική πρωτότυπη εκδοχή του R-tree (για δυναμικά και στατικά περιβάλλοντα) και μια παραλλαγή του. Η παραλλαγή Hilbert R-tree, και συγκεκριμένα στη στατική του μορφή (Packed Hilbert R-tree), θα είναι αυτή που θα χρησιμοποιηθεί στα επόμενα κεφάλαια για την υλοποίηση του αλγορίθμου μας.

2.2.1 Η Δομή των R-trees για Δυναμικά και Στατικά Δεδομένα

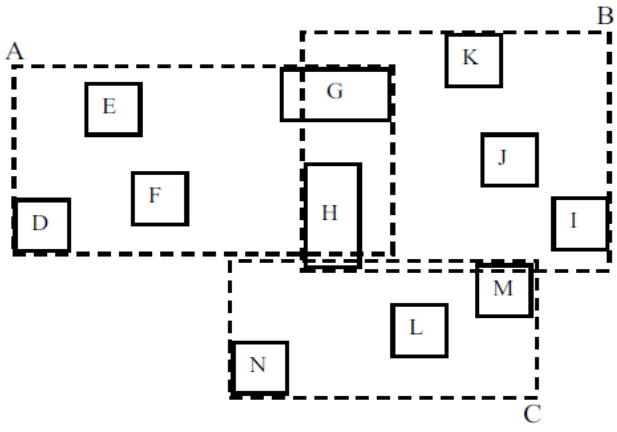
Τα R-trees είναι ιεραρχικές δομές δεδομένων που χρησιμοποιούνται για την οργάνωση/δεικτοδότηση συνόλων πολυδιάστατων γεωμετρικών δεδομένων (έστω διάστασης d), τα οποία αναπαριστώνται από το ελάχιστο περιβάλλον ορθογώνιο σε d διαστάσεις (Minimum Bounding Rectangle ή MBR) .

Κάθε εσωτερικός κόμβος του δέντρου αντιστοιχεί στο ελάχιστο MBR που περικλείει τους κόμβους παιδιά του. Περιέχει ζεύγη της μορφής (MBR, P), όπου MBR το MBR που περικλείει γεωμετρικά τον κόμβο-παιδί και P ο δείκτης προς τον αντίστοιχο κόμβο-παιδί. Τα φύλλα του δέντρου περιέχουν δείκτες προς τα δεδομένα (αντί για δείκτες προς τους κόμβους παιδιά που περιέχουν οι εσωτερικοί κόμβοι). Συγκεκριμένα, τα φύλλα περιέχουν ζεύγη της μορφής (MBR, O), όπου O ο δείκτης προς το γεωμετρικό αντικείμενο και MBR το MBR που περικλείει το αντικείμενο αυτό.

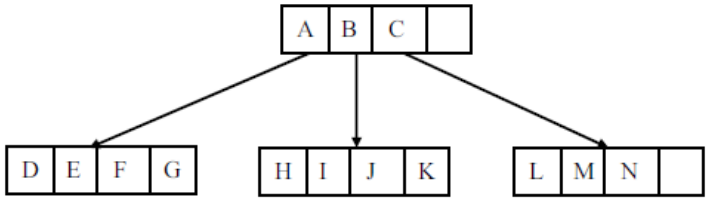
Κάθε κόμβος του δέντρου συνήθως υλοποιείται ως μια σελίδα στο δίσκο, για βελτιστοποίηση του μηχανισμού αποθήκευσης/ανάκτησής του. Τα R-trees είναι ισορροπημένα δέντρα, δηλαδή όλα τα φύλλα τους βρίσκονται στο ίδιο επίπεδο (τυπικά το επίπεδο 0). Η ρίζα βρίσκεται στο ανώτερο επίπεδο και περιέχει τουλάχιστον δύο ζεύγη (δηλαδή έχει τουλάχιστον 2 κόμβους-παιδιά), εκτός εάν είναι φύλλο.

Προφανώς, καθώς τα MBRs είναι ευρύτερα από τις περιοχές δεδομένων που περικλείουν, τα MBRs διαφορετικών κόμβων του δέντρου μπορεί να επικαλύπτονται. Ωστόσο, κάποιο χωρικό δεδομένο προς δεικτοδότηση, ακόμα και αν συμπίπτει με αρκετούς κόμβους στο δέντρο, θα αποθηκευτεί μόνο σε έναν από αυτούς.

Παρακάτω, η Εικόνα 2.1 απεικονίζει μερικά γεωγραφικά αντικείμενα και η Εικόνα 2.2 το αντίστοιχο R-tree. Τα ορθογώνια-δεδομένα D εως N αποθηκεύονται στους κόμβους φύλλα, ενώ τα MBRs A, B και C αποθηκεύονται στους εσωτερικούς κόμβους.



Εικόνα 2.1: Παράδειγμα γεωγραφικών δεδομένων και των MBRs που τα οργανώνει σε εσωτερικούς κόμβους του δέντρου.



Εικόνα 2.2: Το αντίστοιχο R-tree.

Η εκδοχή του R-tree για δυναμικά περιβάλλοντα, επιτρέπει στα χωρικά δεδομένα να εισάγονται στο R-tree ένα-ένα, ενώ παράλληλα υποστηρίζει και διαγραφές αντικειμένων από το δέντρο.

Οι εισαγωγές νέων δεδομένων προωθούνται από τη ρίζα έως τα φύλλα. Σε κάθε επίπεδο, επιλέγεται ο κόμβος που θα χρειαστεί μικρότερη μεγέθυνση του MBR του. Τελικά, το αντικείμενο εισάγεται στο κατάλληλο φύλλο εφόσον δεν είναι γεμάτο, αλλιώς το φύλλο διασπάται για να δημιουργηθούν 2 νέα στη θέση του.

Ωστόσο, πολλές εφαρμογές χρησιμοποιούν στατικά δεδομένα, όπου οι ενημερώσεις ενός δέντρου μέσω επιπλέον εισαγωγών ή διαγραφών είναι σπάνιες ή μηδενικές. Τέτοιες εφαρμογές περιλαμβάνουν για παράδειγμα βάσεις δεδομένων περιβαλλοντικές ή χαρτογραφικές. Για τέτοιες περιπτώσεις στατικών δεδομένων, αναπτύχθηκαν δομές δέντρων packed R-tree χρησιμοποιώντας μεθόδους για την δημιουργία τους γνωστές ως Packing ή Bulk-Loading [6]. Τα packed R-trees πετυχαίνουν μεγιστοποίηση του αξιοποιημένου χώρου σε κάθε κόμβο (maximum storage utilization), ελαχιστοποίηση της επικάλυψης μεταξύ των κόμβων του δέντρου και μικρότερο κόστος αποθήκευσης. Επιπλέον, εξασφαλίζουν βελτίωση της απόδοσης στην εκτέλεση χωρικών επερωτημάτων [7].

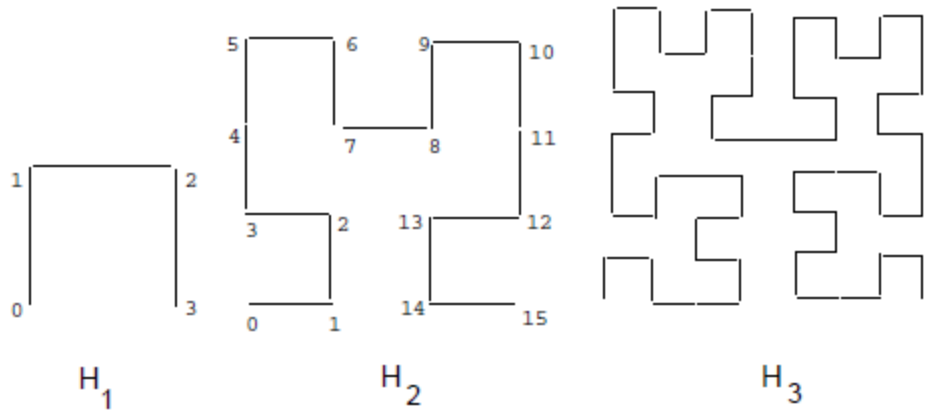
Ο πρώτος αλγόριθμος packing προτάθηκε από τους Roussopoulos και Leifker το 1985 [6], μετά την παρουσίαση του R-tree. Ο αλγόριθμος πρότεινε ταξινόμηση των αντικειμένων σύμφωνα με κάποιο χωρικό κριτήριο (π.χ. σύμφωνα με αύξουσα τιμή μιας συντεταγμένης) και μετά ομαδοποίησή τους σε κόμβους-φύλλα. Αυτή η πρώτη προσέγγιση, αποτέλεσε τη βάση για την ανάπτυξη άλλων επιστημονικών εργασιών που διαμόρφωσαν τις διάφορες παραλλαγές για packed R-trees.

2.2.2 Packed Hilbert R-tree για Στατικά Πολυδιάστατα Δεδομένα

Η απόδοση ενός R-tree στηρίζεται σε μεγάλο βαθμό στο πόσο καλός είναι ο αλγόριθμος που ομαδοποιεί τα ορθογώνια δεδομένα (rectangles) σε έναν κόμβο. Οι Ibrahim Kamel και Christos Faloutsos πρότειναν την χρήση μορφοκλασμάτων στο χώρο (space filling curves ή fractals), και συγκεκριμένα της καμπύλης Hilbert, για τον καθορισμό μιας γραμμικής ταξινόμησης των πολυδιάστατων rectangles ([8], [11], [12]). Ο τύπος R-tree που χρησιμοποιεί την μέθοδο αυτή ονομάστηκε Hilbert R-tree [9].

Με τον διεθνή όρο fractal στα Μαθηματικά, τη Φυσική αλλά και σε πολλές επιστήμες ονομάζεται ένα γεωμετρικό σχήμα που επαναλαμβάνεται αυτούσιο σε άπειρο βαθμό μεγέθυνσης, κι έτσι συχνά αναφέρεται σαν "απείρως περίπλοκο". Το fractal παρουσιάζεται ως "μαγική εικόνα" που όσες φορές και να μεγεθυνθεί οποιοδήποτε τμήμα του θα συνεχίζει να παρουσιάζει ένα εξίσου περίπλοκο σχέδιο με μερική ή ολική επανάληψη του αρχικού. Ένα fractal επισκέπτεται όλα τα σημεία ενός k-διάστατου πλέγματος (grid) ακριβώς μια φορά χωρίς ποτέ να επικαλύπτει τον εαυτό του. Μερικά από τα γνωστά fractals είναι η Z order, η καμπύλη Hilbert και η καμπύλη Gray code. Έχει αποδειχθεί πειραματικά ότι η καμπύλη Hilbert επιτυγχάνει την καλύτερη ομαδοποίηση (clustering) από τις 3 μεθόδους που αναφέρθηκαν [10].

Στην Εικόνα 2.3 απεικονίζεται η βασική καμπύλη Hilbert (τάξης 1) σε ένα πλέγμα 2x2, σημειωμένη ως H1. Η καμπύλη Hilbert τάξης i προκύπτει από την αντικατάσταση κάθε ακμής της βασικής καμπύλης Hilbert (H1) από την καμπύλη Hilbert τάξης i-1, η οποία μπορεί να έχει υποστεί ανάκλαση ή περιστροφή. Οι καμπύλες Hilbert τάξης 2 και 3 φαίνονται επίσης στην Εικόνα 2.3 ως H2 και H3 αντίστοιχα. Καθώς η τάξη της καμπύλης Hilbert τείνει στο άπειρο, η καμπύλη που προκύπτει γίνεται fractal διάστασης 2. Η καμπύλη Hilbert μπορεί να επεκταθεί και σε περισσότερες διαστάσεις. Το μονοπάτι μιας καμπύλης Hilbert ορίζει μια γραμμική ταξινόμηση των σημείων του πλέγματος. Στην Εικόνα 2.3 η καμπύλη H2 εκφράζει μια ταξινόμηση των σημείων για ένα 4x4 πλέγμα. Για παράδειγμα, το σημείο (0,0) στην H2 έχει τιμή Hilbert (Hilbert value) 0, ενώ το σημείο (1,1) έχει τιμή Hilbert 2.



Εικόνα 2.3: Καμπύλες Hilbert τάξης 1, 2 και 3 για πλέγματα 2x2, 4x4 και 8x8 αντίστοιχα.

Το Hilbert R-tree είναι μια δομή δεδομένων βασισμένη στα R-trees. Πιο συγκεκριμένα, είναι μια παραλλαγή R-tree όπου τα γεωμετρικά αντικείμενα χαρακτηρίζονται από την τιμή Hilbert (Hilbert value) του ελάχιστου ορθογωνίου MBR που τα περιβάλλει. Η τιμή Hilbert ενός ορθογωνίου, όπως έχει αποδειχθεί από επιστημονικές δημοσιεύσεις [7], ισούται με την τιμή Hilbert του κέντρου του.

Η ιδέα της χρήσης της τιμής Hilbert οδηγεί σε καλύτερη ταξινόμηση των γεωμετρικών αντικειμένων που επιθυμούμε να εισαχθούν στο Hilbert R-tree. Η ταξινόμηση αυτή συνεπάγεται καλύτερη ομαδοποίηση των χωρικών δεδομένων σε κόμβους του δέντρου, μεγιστοποιώντας την αξιοποίηση του χώρου σε κάθε κόμβο (maximum storage utilization). Με τον τρόπο αυτό το δέντρο έχει μικρότερο ύψος και μεγαλύτερο fan out. Επιπλέον, η χρήση της τιμής Hilbert ομαδοποιεί τα αντικείμενα που είναι «κοντά» στον χώρο μαζί σε έναν κόμβο, καταλήγοντας σε κόμβους με όσο γίνεται μικρό MBR και καλύτερη επίδοση στην εκτέλεση επερωτημάτων.

Στην επιστημονική δημοσίευση «On packing R-trees» [7] οι Kamel και Faloutsos παρουσιάζουν το Packed Hilbert R-tree. Προτείνουν την ταξινόμηση των γεωγραφικών δεδομένων-ορθογωνίων σύμφωνα με την τιμή Hilbert του κέντρου τους. Στην συνέχεια, διατρέχοντας την ταξινομημένη λίστα, αναθέτουμε κάθε σύνολο από C ορθογώνια σε έναν κόμβο-φύλλο του δέντρου. Η κατασκευή του δέντρου συνεχίζεται από κάτω προς τα πάνω (bottom-up) στο ίδιο μοτίβο. Παρακάτω παρουσιάζεται ο αλγόριθμος κατασκευής Packed Hilbert R-tree σε ψευδοκώδικα.

Αλγόριθμος Packed Hilbert R-tree: packing γεωγραφικών δεδομένων σε ένα Rtree.

Βήμα 1. Υπολόγισε την τιμή Hilbert για κάθε ορθογώνιο (data rectangle)

Βήμα 2. Ταξινόμησε τα ορθογώνια κατά αύξουσα τιμή Hilbert

Βήμα 3. /* Σχηματισμός των κόμβων-φύλλων (επίπεδο δέντρου l=0) */
While(υπάρχουν ακόμα ορθογώνια)
Κατασκεύασε έναν νέο κόμβο-φύλλο
Ανάθεσε τα επόμενα C ορθογώνια σε αυτόν τον κόμβο

Βήμα 4. /* Σχηματισμός των εσωτερικών κόμβων (ανώτερα επίπεδα δέντρου l+1) */
While(υπάρχουν πάνω από 1 κόμβος στο επίπεδο l)
Ταξινόμησε τους κόμβους του επιπέδου l κατά αύξων χρόνο δημιουργίας
Επανέλαβε το Βήμα3

Πίνακας 2.1: Αλγόριθμος Packed Hilbert R-tree.

Ο αλγόριθμος Packed Hilbert R-tree ομαδοποιεί τα γεωγραφικά αντικείμενα (τα οποία είναι σε μορφή ορθογωνίου ή αντιπροσωπεύονται από το MBR τους) με κριτήριο την τοπικότητα, μέσω της χρήσης της τιμής Hilbert. Το αποτέλεσμα θα είναι τα δεδομένα κάθε φύλλου να είναι «κοντά» μεταξύ τους στην γραμμική ταξινόμηση και πιθανότατα να είναι κοντά και στο χώρο. Έτσι, το MBR κάθε κόμβου-φύλλου θα είναι το ελάχιστο δυνατό, τείνοντας μάλιστα να έχει μορφή τετραγωνική. Αυτό συνεπάγεται ότι τα MBRs των φύλλων θα έχουν όσο το δυνατόν μικρότερο εμβαδόν και περίμετρο, γεγονός που οδηγεί σε καλύτερη απόδοση στην εκτέλεση επερωτημάτων με τη βοήθεια του δέντρου [7].

2.3 Επεξεργασία Χωρικών Επερωτημάτων (Spatial Query Processing)

Η επεξεργασία χωρικών επερωτημάτων αποτελεί ένα πεδίο υψηλών απαιτήσεων, τόσο λόγω της πολυπλοκότητα των χωρικών δεδομένων και επερωτημάτων, αλλά και λόγω των πολύ μεγάλων συνόλων χωρικών δεδομένων που είναι διαθέσιμα προς επεξεργασία.

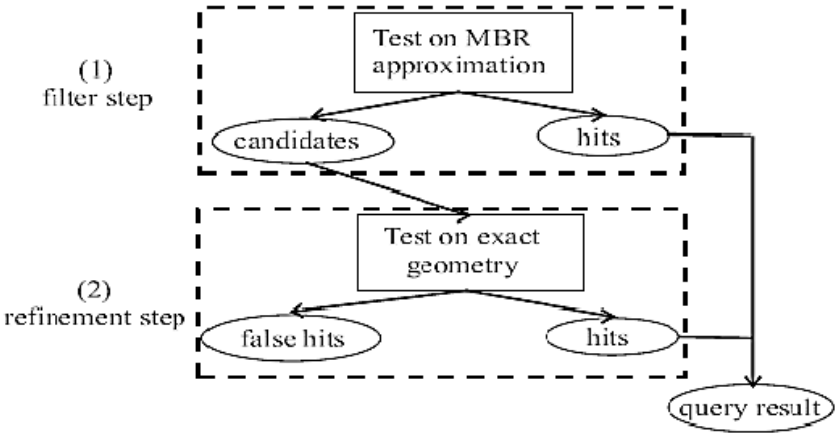
Τα R-trees αποτελούν μια αποδοτική λύση στην διαχείριση τέτοιων δεδομένων και στην εκτέλεση επερωτημάτων σ' αυτά. Οι διάφορες παραλλαγές R-trees που έχουν παρουσιαστεί, προσφέρουν ευελιξία στην υποστήριξη πολλών διαφορετικών χωρικών τελεστών (spatial operators).

Οι πιο κοινοί χωρικοί τελεστές είναι οι παρακάτω:

- ✓ Τοπολογικοί (Topological), όπως παραδείγματος χάριν ανεύρεση των αντικειμένων που επικαλύπτουν ή περιέχουν ένα δοσμένο αντικείμενο.
- ✓ Κατεύθυνσης (Directional), όπως παραδείγματος χάριν ανεύρεση των αντικειμένων που βρίσκονται νότια ενός αντικειμένου.
- ✓ Απόστασης (Distance), όπως παραδείγματος χάριν ανεύρεση των αντικειμένων που βρίσκονται σε λιγότερο από μια δεδομένη απόσταση από ένα συγκεκριμένο αντικείμενο.

Οι τελεστές αυτοί αποτελούν βασικά στοιχεία για ανάπτυξη πιο πολύπλοκων τελεστών σε εφαρμογές διαχείρισης χωρικών δεδομένων.

Όπως αναφέρθηκε στο προηγούμενο υποκεφάλαιο, τα R-tree χρησιμοποιούν MBRs για να εκφράσουν προσεγγιστικά σχήματα γεωγραφικών δεδομένων πιο περίπλοκα, όπως πολύγωνα. Η εκτέλεση χωρικών επερωτημάτων ως εκ τούτου, πραγματοποιείται σε δυο φάσεις [13]. Αρχικά φιλτράρονται τα πιθανά MBRs που απαντούν στο επερώτημα και σε δεύτερη φάση ελέγχεται το υποψήφιο αντικείμενο αυτό καθ' αυτό ώστε να διαπιστωθεί εάν όντως ικανοποιεί τις συνθήκες του επερωτήματος. Η δεύτερη φάση τις περισσότερες φορές είναι απαραίτητη, καθώς μπορεί το MBR του γεωγραφικού αντικειμένου να πληρεί τα κριτήρια του επερωτήματος ενώ το πραγματικό αντικείμενο όχι.



Εικόνα 2.4: Φάσεις εκτέλεσης ενός χωρικού επερωτήματος.

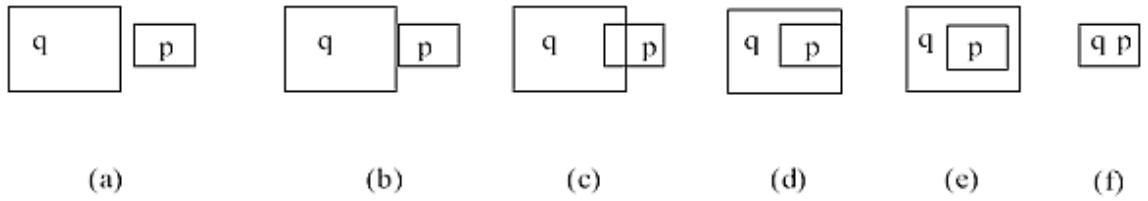
Σημαντικά χωρικά επερωτήματα είναι και queries γνωστά ως επερωτήματα κοντινότερου γείτονα (Nearest Neighbor Queries) ή και χωρικά επερωτήματα όπου λαμβάνει χώρα η πράξη της σύζευξης (Spatial Join Queries). Για τις ανάγκες της παρούσας διπλωματικής, θα παρουσιάσουμε με περισσότερη έμφαση τα τοπολογικά επερωτήματα στην ενότητα που ακολουθεί.

2.3.1 Τοπολογικά Επερωτήματα (Range and Topological Queries)

Η πιο κοινή μορφή επερωτήματος που συναντάμε όταν χρησιμοποιούμε R-trees είναι το επερώτημα εύρους (range query), το οποίο αναζητά όλα τα αντικείμενα τα οποία επικαλύπτει μια δοσμένη περιοχή (query region). Σε πολλές περιπτώσεις, η query region έχει τη μορφή ορθογωνίου, οπότε το query ονομάζεται και επερώτημα παραθύρου (window query). Ως τέτοιας μορφής επερωτήματα μπορούμε να θεωρήσουμε και επερωτήματα όπου η περιοχή του query είναι σημείο, αφού και ένα σημείο μπορεί να θεωρηθεί ως ένα ορθωγώνιο που οι γωνίες του συμπίπτουν. Το query στην περίπτωση αυτή ονομάζεται και επερώτημα σημείου (point query).

Τα range queries απαντώνται με μια διαδικασία που εξετάζει πρώτα τη ρίζα του δέντρου. Για κάθε καταχώριση της ρίζας που το MBR της επικαλύπτεται από την περιοχή του query, η διαδικασία προχωράει στην εξέταση του αντίστοιχου υποδέντρου [14]. Όταν φτάσουμε στο επίπεδο των φύλλων, για κάθε αντικείμενο του οποίου το MBR επικαλύπτεται από την περιοχή του query, εξετάζεται το αντίστοιχο αντικείμενο αυτό καθ' αυτό.

Τα επερωτήματα εύρους χρησιμοποιούν συνήθως τον χωρικό τελεστή επικάλυψης (intersection operator), που μπορεί να θεωρηθεί ως η γενική μορφή ενός συνόλου πιο εξειδικευμένων χωρικών τοπολογικών τελεστών. Ο τελεστής επικάλυψης ουσιαστικά εκφράζει εάν δυο χωρικά αντικείμενα επικαλύπτονται είτε μερικώς είτε ολικώς. Εξειδικεύσεις αυτού του τελεστή είναι οι ακόλουθες χωρικές σχέσεις: συναντά (meet), ισούται με (equal), επικαλύπτει μερικώς (overlap), περιέχει (contains) και καλύπτει (covers). Σχηματικά, οι παραπάνω χωρικές σχέσεις απεικονίζονται στην Εικόνα 2.5 .



Εικόνα 2.5: (a) Μη επικάλυψη, (b) συναντά(q,p), (c) επικαλύπτει μερικώς(q,p), (d) καλύπτει(q,p), (e) περιέχει(q,p), (f) Ισούται με(q,p).

Ο αλγόριθμος για εκτέλεση ενός επερωτήματος εύρους με τη βοήθεια ενός R-tree παρουσιάζεται παρακάτω. Για κάθε καταχώρηση E ενός κόμβου N, E.MBR είναι το αντίστοιχο MBR της καταχώρησης και E.pointer είναι ο δείκτης που αντιπροσωπεύει την συγκεκριμένη καταχώρηση στο επόμενο (κάτω) επίπεδο του δέντρου ως κόμβο ή το αναγνωριστικό (identifier ή id) του αντικειμένου εάν βρισκόμαστε σε επίπεδο φύλλων.

Αλγόριθμος RangeQuery (Κόμβος N, Περιοχή Επερωτήματος Q): Βρίσκει τα ορθογώνια του R-tree με ρίζα N, που επικαλύπτονται από μια ορθογώνια περιοχή επερωτήματος Q. Το σύνολο απαντήσεων είναι το σύνολο A.

Βήμα 1. Εάν ο N δεν είναι κόμβος-φύλλο
Εξέτασε κάθε καταχώρηση E του N για να βρεις αυτές που E.MBR που επικαλύπτουν το Q.

Βήμα 2. Για κάθε τέτοια καταχώρηση E εκτέλεσε την RangeQuery(E.pointer, Q).

Βήμα3. Αλλιώς εάν ο N είναι κόμβος-φύλλο
Εξέτασε κάθε καταχώρηση E του N για να βρεις αυτές που E.MBR που επικαλύπτουν το Q.

Βήμα 4. Πρόσθεσε αυτές τις καταχωρήσεις στο σύνολο A.

Πίνακας 2.2: Αλγόριθμος για εκτέλεση ενός Range Query.

Τα αποτελέσματα του παραπάνω αλγόριθμου δίνουν τα υποψήφια αποτελέσματα με βάση τα MBRs των χωρικών αντικειμένων του R-tree. Το τελικό σύνολο των αποτελεσμάτων προκύπτει με ένα τελευταίο βήμα όπου τα αντικείμενα αυτά καθ' αυτά ελέγχονται εάν ικανοποιούν την συνθήκη του επερωτήματος.

2.3.2 Βελτιστοποίηση Εκτέλεσης Επερωτημάτων με Παραλληλοποίηση

Η υιοθέτηση του αποδοτικότερου σχεδίου εκτέλεσης ενός χωρικού επερωτήματος, απαιτεί συνήθως εργαλεία για την πρόβλεψη του αριθμού των χωρικών αντικειμένων που θα ανακτηθούν από το query (selectivity factor), καθώς και το κόστος από άποψη διεργασιών εισόδου/εξόδου (I/O cost) ή υπολογιστικής προσπάθειας (CPU effort). Στις παραδοσιακές τεχνικές βελτιστοποίησης χρησιμοποιούνται μοντέλα υπολογισμού της πολυπλοκότητας, υπολογισμοί της επιλεκτικότητας (selectivity) ή και στατιστικές τεχνικές με χρήση ιστογραμμάτων ή δειγματοληψίας.

Στην περίπτωση χρήσης R-trees, η πρόβλεψη της απόδοσης μιας τεχνικής εκτέλεσης ενός χωρικού επερωτήματος σχετίζεται άμεσα με τον αριθμό των κόμβων και των καταχωρήσεων του R-tree που θα χρειαστεί να προσπελαστούν. Σαν γενικό κανόνα, επιδιώκουμε το σχέδιο εκτέλεσης που θα απαιτεί πρόσβαση στους λιγότερους κόμβους του δέντρου, από τη στιγμή που ένας κόμβος αντιστοιχεί σε μια σελίδα του δίσκου. Βέβαια, το πραγματικό κόστος μπορεί να είναι ακόμα μικρότερο, εάν χρησιμοποιούνται τεχνικές ενδιάμεσης αποθήκευσης στη μνήμη (buffering).

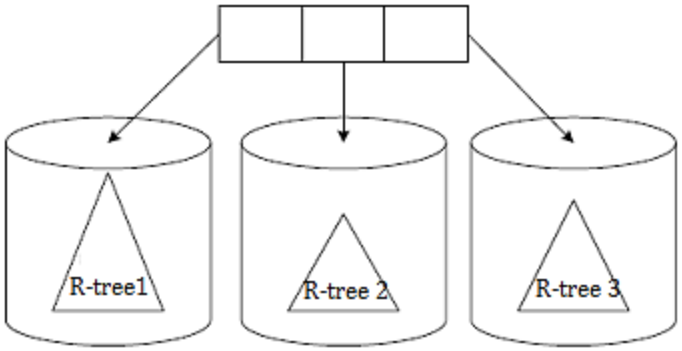
Η πολύπλοκη φύση των χωρικών δεδομένων ή και επερωτημάτων, σε συνδυασμό με την τεράστια ποσότητα δεδομένων που καλείται συχνά να διαχειριστεί μια εφαρμογή, έχουν οδηγήσει τους προγραμματιστές στην επιστράτευση πολλαπλών μονάδων επεξεργασίας (πολλαπλοί επεξεργαστές, μηχανήματα ή δίσκοι) και τεχνικών παραλληλοποίησης για πιο αποδοτική εκτέλεση χωρικών επερωτημάτων [15].

Κατά τον σχεδιασμό αλγορίθμων για παράλληλη επεξεργασία χωρικών δεδομένων, χρειάζεται προσεκτική μελέτη 3 σημαντικών απαιτήσεων:

- ✓ Παράγοντας speed-up: Εκφράζει την απόδοση του αλγορίθμου όταν αυξάνεται ο βαθμός παραλληλοποίησης (π.χ. με χρήση περισσότερων επεξεργαστών) με σταθερό μέγεθος δεδομένων εισόδου. Ιδανικά, το speed-up είναι γραμμικό, που σημαίνει ότι εάν π.χ. διπλασιάσω τον βαθμό παραλληλοποίησης ο χρόνος εκτέλεσης μειώνεται στο μισό.
- ✓ Παράγοντας size-up: Εκφράζει την απόδοση του αλγορίθμου όταν αυξάνεται το μέγεθος δεδομένων εισόδου, με σταθερό βαθμό παραλληλοποίησης (χρήση του ίδιου αριθμού επεξεργαστών ή δίσκων π.χ.).
- ✓ Παράγοντας scale-up: Εκφράζει την απόδοση του αλγορίθμου όταν αυξάνεται και ο βαθμός παραλληλοποίησης (π.χ. με χρήση περισσότερων επεξεργαστών) αλλά και το μέγεθος δεδομένων εισόδου.

Μια πολύ ενδιαφέρουσα στρατηγική παραλληλοποίησης κάνει χρήση ενός συνόλου διαφορετικών υπολογιστών συνδεδεμένων μεταξύ τους μέσω δικτύου (cluster υπολογιστών). Τέτοιο μοντέλο ανάπτυξης κατανεμημένων εφαρμογών είναι και το Hadoop MapReduce [16], που θα παρουσιαστεί αναλυτικότερα στο επόμενο κεφάλαιο. Η χρήση κατανεμημένων συστημάτων για επεξεργασία χωρικών επερωτήσεων αυξάνει την απόδοση, καθώς κατανέμει το φόρτο εργασίας σε πολλούς υπολογιστές που μπορούν να εκτελούν υπολογισμούς παράλληλα.

Υπάρχουν αρκετές εναλλακτικές τεχνικές σχεδίασης ενός συστήματος διαχείρισης χωρικών δεδομένων ώστε να μπορεί να επωφεληθεί από κατανεμημένα συστήματα ([18], [19]). Για κατασκευή R-trees που να μπορούν να στηρίξουν επεξεργασία επερωτημάτων κατανεμημένα μια καλή τεχνική είναι η κατασκευή ενός αριθμού από ανεξάρτητα R-trees [17]. Τα χωρικά δεδομένα διαμερίζονται (μοιράζονται) ανάμεσα σε έναν αριθμό διαθέσιμων κόμβων επεξεργασίας και ένα R-tree χτίζεται για κάθε κόμβο.



Εικόνα 2.6: Ανεξάρτητα δέντρα R-trees σε κάθε κόμβο επεξεργασίας.

Η απόδοση του συστήματος εξαρτάται από 2 σημαντικούς παράγοντες: τον τρόπο που τα χωρικά δεδομένα θα διαμοιραστούν ώστε να κατασκευαστούν καλής ποιότητας R-trees, αλλά και τον τρόπο που θα διαμοιράζονται τα επερωτήματα στους κόμβους επεξεργασίας για εκτέλεση.

Όσο αφορά την κατανομή των χωρικών δεδομένων για την κατασκευή των R-trees:

- ✓ Κατανομή με βάση τα δεδομένα (Data Distribution):

  Τα χωρικά δεδομένα διαμοιράζονται στους κόμβους επεξεργασίας με κριτήριο την ανάθεση ίσου περίπου αριθμού δεδομένων σε κάθε κόμβο, συνήθως χρησιμοποιώντας συναρτήσεις κατακερματισμού ή τεχνικές Round Robin. Η μέθοδος αυτή εξασφαλίζει ισορροπημένη κατανομή των δεδομένων, αλλά καθώς δεν υπάρχει κάποιο χωρικό κριτήριο, η εκτέλεση επερωτημάτων θα εξετάζει συνήθως όλους τους κόμβους.

- ✓ Κατανομή με βάση το χώρο (Space Distribution):

  Ο k-διάστατος χώρος χωρίζεται σε τόσες διαμερίσεις όσοι και οι διαθέσιμοι κόμβοι επεξεργασίας. Τα χωρικά δεδομένα διαμοιράζονται στους κόμβους επεξεργασίας με κριτήριο την τιμή τους στον k-διάστατο χώρο. Στην περίπτωση μη ομοιόμορφων δεδομένων ωστόσο, αυτή η μέθοδος θα δημιουργήσει άνισες διαμερίσεις, με συνέπεια ο φόρτος της εκτέλεσης επερωτημάτων να μην μπορεί να μοιραστεί ισορροπημένα ανάμεσα στους κόμβους επεξεργασίας.

Όσο αφορά τη στρατηγική εκτέλεσης επερωτημάτων κατανεμημένα, απαιτείται προσοχή στην ισορροπημένη κατανομή του φόρτου εργασίας στους κόμβους επεξεργασίας. Στην αντίθετη περίπτωση, ο κόμβος που επεξεργάζεται την πιο χρονοβόρα διεργασία θα προκαλέσει καθυστέρηση σε όλο το σύστημα (bottleneck). Επιπλέον, οι παρακάτω παρατηρήσεις πρέπει να ληφθούν υπ' όψιν:

- ✓ Στην περίπτωση που το επερώτημα αφορά ένα συγκεκριμένο υποσύνολο των χωρικών δεδομένων του R-tree που βρίσκονται σε λίγους μόνο κόμβους, οι κόμβοι αυτοί θα προκαλέσουν καθυστέρηση στην συνολική εκτέλεση και θα χαθεί η έννοια της παραλληλοποίησης αφού οι υπόλοιποι κόμβοι θα παραμείνουν αδρανείς .
- ✓ Στην περίπτωση που τα χωρικά δεδομένα κατά την κατασκευή του δέντρου είχαν διαμοιραστεί στους κόμβους με μη χωρικό κριτήριο (με την τεχνική της κατανομής με βάση τα δεδομένα που περιγράφηκε παραπάνω), το επερώτημα θα πρέπει να εκτελεστεί σε όλους τους κόμβους επεξεργασίας, ακόμα και αν στην πραγματικότητα τα δεδομένα στα δέντρα τους δεν απαντάνε το επερώτημα.

3. Hadoop και MapReduce

Το Hadoop Map/Reduce είναι ένα framework λογισμικού γραμμένο σε Java για την ανάπτυξη εφαρμογών παράλληλης επεξεργασίας μεγάλου όγκου δεδομένων (της τάξης δεκάδων Gigabytes/Terabytes) σε υπολογιστές συνδεδεμένους μεταξύ τους μέσω δικτύου (cluster υπολογιστών), με τρόπο αξιόπιστο και ανεκτικό στις αστοχίες. Το Hadoop δημιουργήθηκε από τον Doug Cutting, αλλά έγινε ευρύτερα γνωστό το 2004, όταν η Google δημοσίευσε την σχετική επιστημονική εργασία [16]. Σήμερα αποτελεί ένα συνεχώς αναπτυσσόμενο project του Apache Software Foundation, που χρησιμοποιείται παγκοσμίως για ποικίλες ακαδημαϊκές εργασίες αλλά και επιχειρησιακές εφαρμογές σε μεγάλες εταιρείες όπως η Yahoo!, το Facebook, και οι New York Times [22].

3.1 Εισαγωγή

Στις εφαρμογές που διαχειρίζονται μεγάλο όγκο δεδομένων (εκατοντάδες Megabytes, Gigabytes ή Terabytes), η εκτέλεση των υπολογισμών κατανεμημένα αποτελεί συχνά μονόδρομο. Το Hadoop παρέχει ένα framework για την ανάπτυξη κατανεμημένων εφαρμογών όπου η επεξεργασία των δεδομένων παραλληλοποιείται σε ένα σύνολο κόμβων ενός cluster, καταλήγοντας σε προγράμματα με υψηλή απόδοση και επεκτασιμότητα. Η διαχείριση της αποθήκευσης παρέχεται από το HDFS (Hadoop Distributed FileSystem) [21] και το προγραμματιστικό μοντέλο από το MapReduce.

Τα κύρια μέρη του Hadoop είναι τα εξής [20]:

- ✓ Core : Ένα σύνολο από πρόσθετα και διεπαφές για κατανεμημένα συστήματα αρχείων και γενικές λειτουργίες εισόδου/εξόδου (serialization, Java RPC, persistent data structures).
- ✓ MapReduce : Ένα κατανεμημένο μοντέλο επεξεργασίας δεδομένων και περιβάλλον εκτέλεσης που τρέχει σε μεγάλους cluster κοινών υπολογιστών (commodity machines).
- ✓ HDFS : Ένα κατανεμημένο σύστημα αρχείων, βελτιστοποιημένο για γρήγορη πρόσβαση σε μεγάλο όγκο δεδομένων.

Το μοντέλο MapReduce παίρνει σαν είσοδο ένα σύνολο από ζευγάρια της μορφής <κλειδί, τιμή> και παράγει στην έξοδο ένα σύνολο από ζευγάρια της ίδιας μορφής. Ο προγραμματιστής εκφράζει τον αλγόριθμο ορίζοντας δύο συναρτήσεις που διαχειρίζονται τέτοια ζεύγη, την map και την reduce, , και η ροή δεδομένων ελέγχεται από το σύστημα.

Η κεντρική ιδέα του MapReduce στο Hadoop είναι ότι τα δεδομένα εισόδου μπορούν να χωριστούν σε λογικά κομμάτια (input splits) και κάθε κομμάτι μπορεί να υποστεί επεξεργασία ανεξάρτητα από μια διεργασία Map (Map task), δίνοντας έξοδο της μορφής <κλειδί, τιμή>. Η έξοδος της φάσης map ομαδοποιείται σε διαμερίσεις (partitions), με κριτήριο την τιμή του κλειδιού του ζεύγους εξόδου. Κάθε διαμέριση θα αποτελέσει είσοδο σε μια διεργασία Reduce (Reduce task). Τα ζευγάρια κάθε διαμέρισης ομαδοποιούνται και ταξινομούνται με κριτήριο το κλειδί και δίνονται ως είσοδος στην συνάρτηση reduce, έτσι ώστε κάθε κλήση της reduce να έχει είσοδο της μορφής <κλειδί, λίστα τιμών>. Τα αποτελέσματα όλων των διεργασιών Reduce είναι η τελική έξοδος του προγράμματος.

Το Hadoop προσφέρει ένα επίπεδο αφαίρεσης στη διαδικασία ανάπτυξης παράλληλων προγραμμάτων, αναλαμβάνοντας να διαχειριστεί τις λεπτομέρειες του διαμοιρασμού των δεδομένων, την αντιμετώπιση των αστοχιών υλικού και την ισορροπημένη κατανομή του φόρτου εργασίας στον cluster, διαχωρίζοντας έτσι την λογική του αλγορίθμου (business logic) από τον κώδικα ελέγχου της παραλληλοποίησης. Έτσι, ο προγραμματιστής μπορεί να συγκεντρωθεί στην ανάπτυξη της λειτουργικότητας της εφαρμογής.

Στην πραγματικότητα, και ιδιαίτερα όταν χρησιμοποιούμε μη εξειδικευμένα μηχανήματα (commodity PCs), η πιθανότητα αστοχίας υλικού είναι υψηλή. Ένα από τα μεγαλύτερα πλεονεκτήματα του Hadoop είναι η δυνατότητά του να χειρίζεται τις αστοχίες, καθώς το σύστημα ανιχνεύει διεργασίες που έχουν αποτύχει και τις επαναδρομολογεί σε άλλους κόμβους του cluster. Έτσι, η αξιοπιστία διασφαλίζεται σε επίπεδο λογισμικού και όχι εξαρτώμενη από την ποιότητα του hardware, επιτρέποντάς μας να εκτελούμε απαιτητικές εφαρμογές σε cluster με μικρό κόστος.

Η φιλοσοφία του Hadoop ως προς την διαχείριση των δεδομένων, συμπυκνώνεται στην φράση “είναι φτηνότερο να αναθέσεις την επεξεργασία εκεί που βρίσκονται τα δεδομένα, από το να τα μεταφέρεις” (αρχή της τοπικότητας ή data locality). Ένας υπολογισμός σε μια εφαρμογή είναι αποδοτικότερο να εκτελεστεί «κοντά» στα δεδομένα τα οποία διαχειρίζεται, ειδικά όταν έχουμε μεγάλο όγκο δεδομένων. Το σύστημα αρχείων HDFS παρέχει διεπαφές ώστε να εκτελεί τους υπολογισμούς εκεί που η πρόσβαση στα δεδομένα είναι γρηγορότερη (ιδανικά τοπική), μειώνοντας έτσι την κίνηση στο δίκτυο του cluster και αυξάνοντας στην απόδοση του συστήματος (throughput).

Το Hadoop είναι ιδανική επιλογή για προβλήματα που αναλύουν μεγάλα σύνολα δεδομένων μαζικά (batch processing), και που τα δεδομένα γράφονται μια φορά και διαβάζονται πολλές (write once-read many). Επιπλέον, το Hadoop μπορεί να διαχειριστεί μη δομημένα ή ημι-δομημένα δεδομένα, σε αντίθεση με τα Σχεσιακά Συστήματα Βάσεων Δεδομένων (RDBMSs) που στηρίζονται σε Σχήματα (database Schemas).

Με την αυξανόμενη διάδοσή του, το Hadoop MapReduce έχει επεκταθεί πέρα από το αρχικό του πεδίο εφαρμογών (log-processing), σε νέες εφαρμογές ακαδημαϊκού περιεχομένου ή συστήματα στήριξης επιχειρηματικών αποφάσεων.

3.2 Hadoop MapReduce – Προγραμματιστικό Μοντέλο

Το MapReduce είναι ένα προγραμματιστικό μοντέλο για παράλληλη επεξεργασία δεδομένων.

Λειτουργεί χωρίζοντας την επεξεργασία σε 2 φάσεις: τη φάση Map και τη φάση Reduce. Κάθε φάση έχει ζευγάρια της μορφής <κλειδί, τιμή> σαν είσοδο και έξοδο, των οποίων οι τύποι ορίζονται απ' τον προγραμματιστή. Ο προγραμματιστής ορίζει επιπλέον 2 συναρτήσεις, τη συνάρτηση map και τη συνάρτηση reduce.

Οι συναρτήσεις αυτές έχουν την παρακάτω γενική μορφή:

map: (Key1, Value1) → list(Key2, Value2)

reduce: (Key2, λίστα(Value2)) → list(Key3, Value3)

όπου: Key1, Value1 οι τύποι των δεδομένων εισόδου της συνάρτησης map,

Key2, Value2 οι τύποι των δεδομένων εξόδου της συνάρτησης map και εισόδου της συνάρτησης reduce,

Και Key3, Value3 οι τύποι των δεδομένων εξόδου της συνάρτησης reduce.

Η συνάρτηση map παίρνει ως είσοδο ζευγάρια δεδομένων της μορφής <κλειδι, τιμή>, τα επεξεργάζεται και επιστρέφει έναν αριθμό από ζευγάρια δεδομένων της ίδιας μορφής.

Map : (key1, value1) -> list (key2, value2)

Εφαρμόζεται παράλληλα σε κάθε κομμάτι που έχει διαχωριστεί η είσοδος, παράγοντας έτσι ένα σύνολο από ζευγάρια σε κάθε κλήση της.

Στη συνέχεια, το MapReduce διαμερίζει την έξοδο της φάσης Map σε R διαμερίσεις, μια για κάθε διεργασία Reduce (Reducer ή Reduce task). Η συνάρτηση διαμέρισης (partition function) καθορίζει την διαμέριση του κάθε ζεύγους <κλειδί, τιμή> ανάλογα με την τιμή του κλειδιού.

partition: (K2, V2) → integer

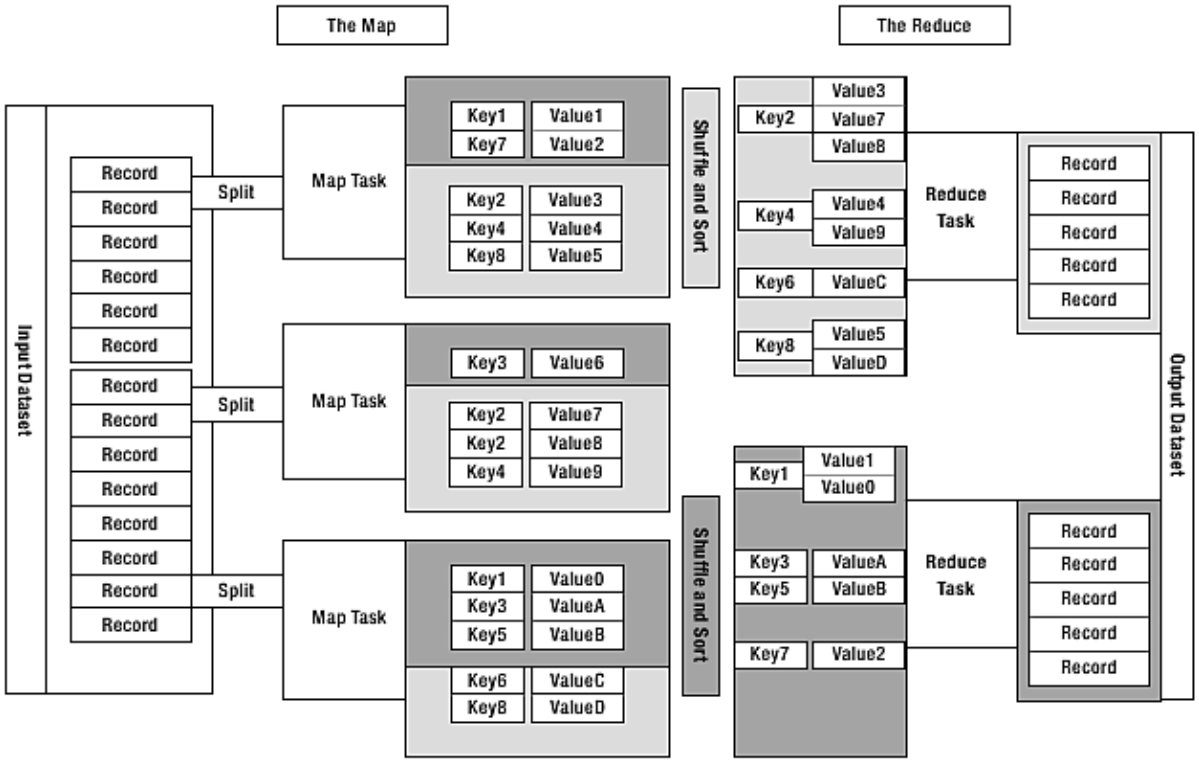
Σε κάθε διεργασία Reduce, το MapReduce συγκεντρώνει όλα τα ζευγάρια <κλειδί, τιμή>, τα ταξινομεί και τα ομαδοποιεί με βάση την τιμή του κλειδιού, έτσι ώστε μια ομάδα να περιλαμβάνει όλες τις τιμές εξόδου για ένα συγκεκριμένο κλειδί. Δημιουργούνται έτσι ζευγάρια της μορφής <κλειδί, λίστα τιμών>, όπου κάθε ζεύγος θα αποτελέσει είσοδο για μια κλήση της συνάρτησης reduce.

Reduce : (key2, list (value2)) -> list (key3, value3)

Τα αποτελέσματα όλων των κλήσεων της συνάρτησης reduce, από όλες τις διεργασίες

Reduce, αποτελούν το τελικό αποτέλεσμα του MapReduce προγράμματος.

Προαιρετικά ο προγραμματιστής χρησιμοποιεί και μια τρίτη συνάρτηση, τη συνάρτηση combine. Η συνάρτηση combine είναι μια συνάρτηση που τρέχει τοπικά στον υπολογιστή που έτρεξε τη συνάρτηση map. Η κλήση της γίνεται πριν τα αποτελέσματα της map οδηγηθούν στη φάση Reduce μέσω δικτύου, εκτελώντας μια ενδιάμεση επεξεργασία που διαφορετικά θα έπρεπε να γίνει στην φάση Reduce. Έτσι ένα ποσοστό της επεξεργασίας έχει γίνει χωρίς μεταφορά δεδομένων στο δίκτυο, έχοντας ως αποτέλεσμα γρηγορότερη επεξεργασία και λιγότερη επιβάρυνση του δικτύου του cluster.



Εικόνα 3.1: Το προγραμματιστικό μοντέλο MapReduce.

Παρακάτω παρουσιάζεται ένα τυπικό παράδειγμα αλγορίθμου MapReduce σε ψευδοκώδικα, που μετράει την συχνότητα εμφάνισης λέξεων (WordCount) σε διάφορα έγγραφα.

Αλγόριθμος WordCount: Μετράει την συχνότητα εμφάνισης λέξεων σε διάφορα έγγραφα.

```
void map(String name, String document){
        // name: document name
        // document: document contents
        for each word w in document:
                EmitIntermediate(w, ”1”);    }

void reduce(String word, Iterator partialCounts){
        // word: a word
        // partialCounts: a list of aggregated partial counts
        int result = 0;
        for each pc in partialCounts:
                result += ParseInt(pc);
        Emit(AsString(result));                }
```

Πίνακας 3.1: Αλγόριθμος WordCount

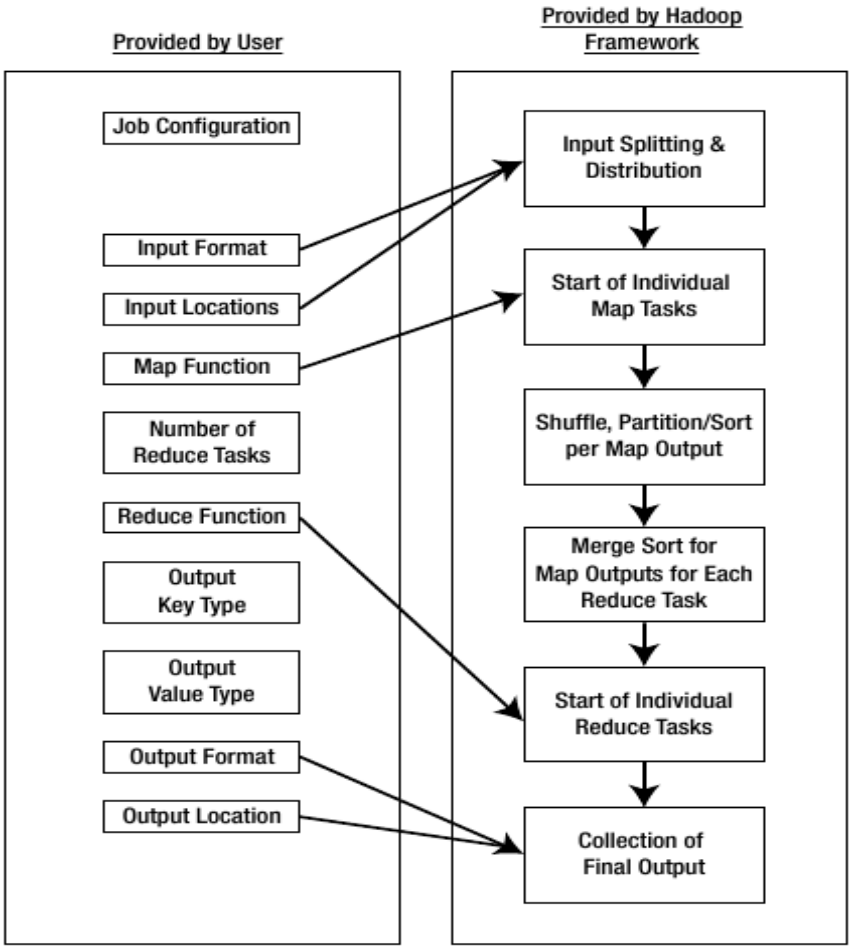
Το κάθε έγγραφο χωρίζεται σε λέξεις και η κάθε λέξη έχει αρχικά μια συχνότητα ίση με 1. Η ίδια η λέξη είναι το κλειδί με βάση το οποίο θα ομαδοποιηθούν οι συχνότητες. Άρα, η έξοδος του Map είναι της μορφής ((word1, 1), (word2, 1), (word1, 1)), ενώ η είσοδος του Reduce για το κλειδί word1 θα είναι (word1, (1,1)). Η reduce αρκεί να αθροίσει τη λίστα από τους άσσους για να βρει τη συνολική συχνότητα της λέξης. Η χρήση της συνάρτησης reduce ως συνάρτηση combine στον παραπάνω κώδικα είναι πολύ συνηθισμένη. Τα αποτελέσματα της map θα συγκεντρωθούν τότε σε λίστες στην κύρια μνήμη αντί για τον δίσκο. Η έξοδος του κόμβου προς τη reduce θα είναι τότε (word, συχνότητα στο συγκεκριμένο κομμάτι του είσοδο).

3.3 Hadoop MapReduce - Επίπεδο Ροής Δεδομένων

Μια εργασία MapReduce (MapReduce job) είναι μια εφαρμογή που ο χρήστης επιθυμεί να εκτελεστεί. Αποτελείται από τα δεδομένα εισόδου, το πρόγραμμα γραμμένο σύμφωνα με

το μοντέλο MapReduce, και πληροφορίες παραμετροποίησης (configuration information). Το Hadoop τρέχει την εργασία διασπώντας την σε διεργασίες (tasks) 2 τύπων: διεργασίες Map και διεργασίες Reduce.

Προκειμένου να εκτελεστούν, οι εφαρμογές θα πρέπει να ορίσουν τις τοποθεσίες εισόδου/εξόδου, τις συναρτήσεις map και reduce, μέσω υλοποιήσεων των κατάλληλων διεπαφών ή abstract κλάσεων. Αυτές, και άλλες παράμετροι, συνθέτουν την παραμετροποίηση της εργασίας (*job configuration*). Ο Hadoop *job client* στη συνέχεια αναθέτει την εργασία και τις παραμέτρους για εκτέλεση στον JobTracker, ο οποίος αναλαμβάνει την ευθύνη κατανομής της εκτέλεσης στον cluster και την επίβλεψη της εξέλιξης της εργασίας.



Εικόνα 3.2: Στάδια εκτέλεσης μιας εργασίας Hadoop.

Τα στάδια της ροής δεδομένων κατά την εκτέλεση ενός Hadoop MapReduce προγράμματος είναι:

1. Input Reader: Ο Input Reader αναλαμβάνει να χωρίσει τα δεδομένα εισόδου σε κατάλληλου μεγέθους κομμάτια (input splits), τυπικά μεταξύ 16 και 128 Megabytes. Ο Input Reader έχει ως είσοδο δεδομένα από το σύστημα αρχείων HDFS και ως έξοδο παράγει ζευγάρια της μορφής <κλειδί, τιμή>. Εσωτερικά αυτό το στάδιο είναι υλοποιημένο με δύο βήματα: η κλάση InputFormat χωρίζει την είσοδο σε splits και η συσχετιζόμενη κλάση RecordReader, χωρίζει το κάθε split σε ζευγάρια της μορφής <κλειδί, τιμή> τα οποία θα αποτελέσουν την είσοδο της φάσης Map. Το Hadoop παρέχει έναν αριθμό από έτοιμες κλάσεις InputFormat για διάβασμα των δεδομένων εισόδου με διάφορους τρόπους (και από δεδομένα σε μορφή κειμένου ή δυαδική), αλλά ο προγραμματιστής μπορεί να υλοποιήσει μια δική του κλάση InputFormat για τις ανάγκες της εφαρμογής του.

2. Κλάση Map: Ανάλογα με τον αριθμό των input splits, δημιουργείται από το σύστημα ίσος αριθμός διεργασιών Map (Mappers ή Map tasks). Οι διεργασίες κατανέμονται στον cluster και ο κάθε κόμβος αναλαμβάνει έναν αριθμό Mappers, οι οποίοι θα εκτελεστούν παράλληλα. Ο κάθε Mapper αναλαμβάνει ένα μοναδικό κομμάτι εισόδου (split), το οποίο έχει ήδη χωριστεί σε ζευγάρια της μορφής <κλειδί, τιμή>. Κάθε ζευγάρι με τη σειρά του πυροδοτεί μία νέα κλήση της συνάρτησης map που έχει καθοριστεί απ' τον χρήστη. Τα αποτελέσματα όλων των κλήσεων της συνάρτησης map, για όλες τις διεργασίες Map (map tasks) γράφονται στον δίσκο ταξινομημένα σύμφωνα με το κλειδί.

3. συνάρτηση Partition: Ο χρήστης ορίζει τον αριθμό των διεργασιών Reduce (reducers ή Reduce tasks) στις παραμέτρους configuration της εργασίας. Όταν ο αριθμός των Reducers είναι πάνω από 1 (έστω R), το MapReduce διαμερίζει την έξοδο της φάσης Map σε R διαμερίσεις, μια για κάθε διεργασία Reduce (Reducer ή Reduce task). Η συνάρτηση διαμέρισης (partition function) καθορίζει την διαμέριση του κάθε ζεύγους <κλειδί, τιμή> ανάλογα με την τιμή του κλειδιού. Αυτό έχει σαν αποτέλεσμα όλα τα ζευγάρια με ίδια τιμή κλειδιού να ανήκουν στην ίδια διαμέριση, η οποία μπορεί φυσικά να περιέχει ζευγάρια που αντιστοιχούν σε πολλές τιμές κλειδιού. Η τυπική συνάρτηση διαμέρισης εφαρμόζει κατακερματισμό στην τιμή του κλειδιού (hashing), αλλά ο χρήστης μπορεί να χρησιμοποιήσει άλλες συναρτήσεις partitioning που παρέχονται απ' το Hadoop ή να ορίσει μια δική του.

4. φάση Shuffle και κλάση Sort Comparator: Η διαμέριση-είσοδος για την κάθε διεργασία Reduce μεταφέρεται από τους κόμβους που έτρεξε η φάση Map μέσω δικτύου και συγκεντρώνεται στον κόμβο που θα εκτελέσει την κάθε διεργασία Reduce. Η διαδικασία αυτή της συγκέντρωσης των ζευγαριών είναι γνωστή σαν διαδικασία Shuffle. Στη συνέχεια τα ζευγάρια κάθε διαμέρισης ταξινομούνται με βάση την τιμή του κλειδιού κάθε ζεύγους, σύμφωνα με την κλάση Sort Comparator.

5. κλάση Grouping Comparator: Με βάση αυτή την κλάση, το Hadoop ομαδοποιεί τα

ζευγάρια κάθε διαμέρισης με την ίδια τιμή κλειδιού. Έτσι δημιουργούνται ζευγάρια της μορφής <κλειδί, λίστα τιμών>.

6. συνάρτηση Reduce: Μπορεί να υπάρχουν πολλές διεργασίες Reduce που τρέχουν παράλληλα στον cluster. Ο κάθε Reducer έχει είσοδο τα ζευγάρια της μορφής <κλειδί, λίστα τιμών> που δημιουργήθηκαν στο προηγούμενο βήμα, για την διαμέριση που έχει αναλάβει. Στη συνέχεια πυροδοτείται μια κλήση της συνάρτησης reduce για κάθε τέτοιο ζεύγος, με είσοδο το ζευγάρι αυτό.

7. κλάση RecordWriter: Ο RecordWriter αναλαμβάνει να γράψει τα αποτελέσματα της φάσης Reduce στο HDFS. Η μορφή της εξόδου καθορίζεται από την κλάση OutputFormat . Το Hadoop παρέχει έναν αριθμό από έτοιμες κλάσεις OutputFormat για εγγραφή των δεδομένων εξόδου με διάφορους τρόπους (σε μορφή κειμένου ή δυαδική), αλλά ο προγραμματιστής μπορεί να υλοποιήσει μια δική του υλοποίηση για τις ανάγκες της εφαρμογής του.

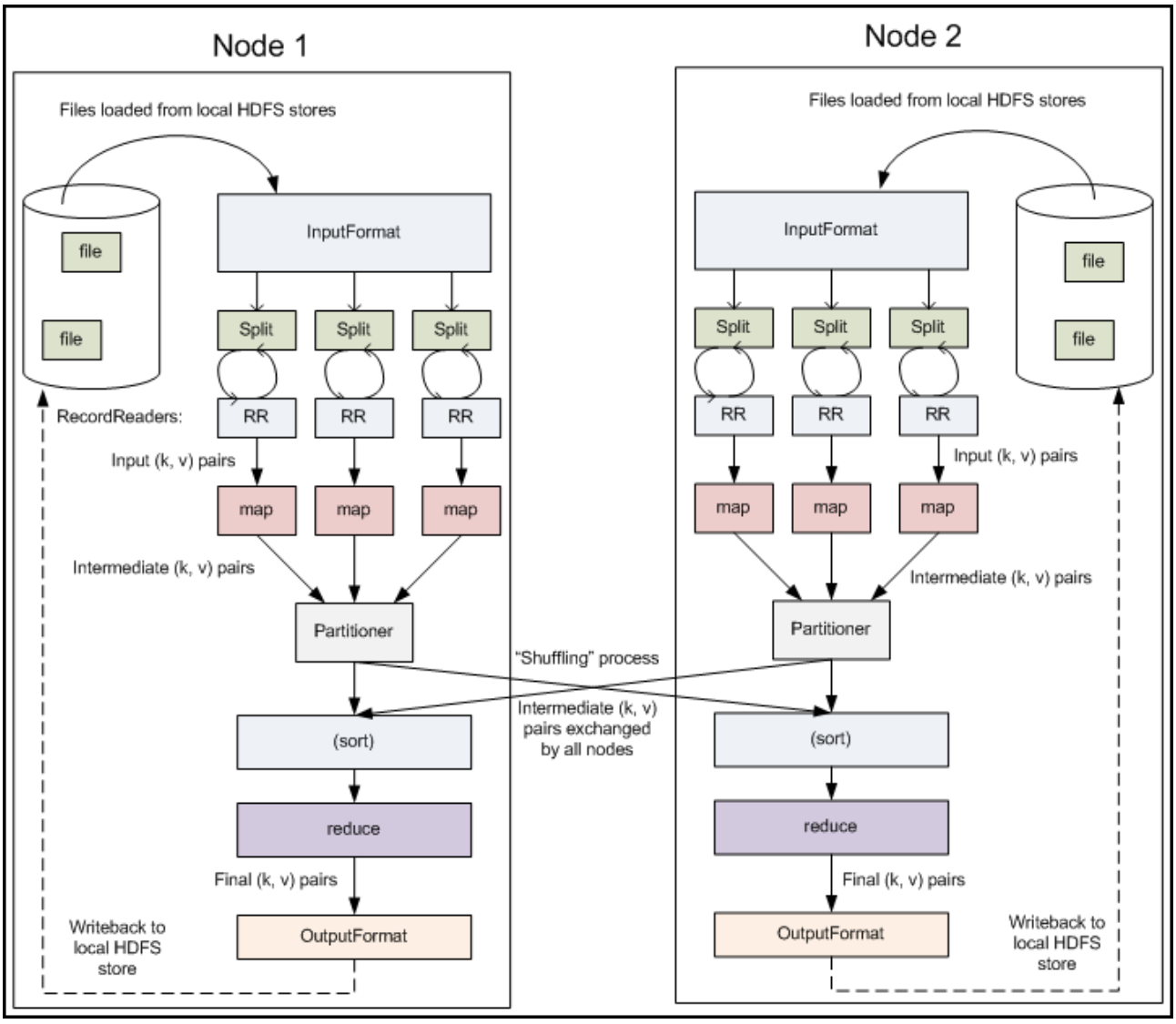
Πριν κλείσει η ενότητα αυτή, θα παραθέσουμε τη γενική μορφή την οποία έχει ένα πρόγραμμα που χρησιμοποιεί το Hadoop framework. Στο πλαίσιο που ακολουθεί, φαίνεται η γενική μορφή της συνάρτησης main που αρχικοποιεί την εργασία Hadoop, και η μορφή της κλάσης Map. Η κλάση Reduce έχει αντίστοιχη μορφή και γι' αυτό δεν παρατίθεται.

```
public static void main(String args[]) throws Exception {
        Configuration conf = new Configuration();   //για παραμετροποίηση της εργασίας
        conf.setProperty(property, default value);   // ορίζουμε τιμές παραμέτρων
        Job job = new job(conf);                     // αρχικοποιούμε νέα εργασία
        job.setJobName("Name of Job");

        job.setMapperClass(MyMapper.class);          // η κλάση Map
        job.setReducerClass(MyReducer.class);        // η κλάση Reduce
        job.setMapOutputKeyClass(DatatypeA.class);      //Μορφή εξόδου των Map, Reduce
        job.setMapOutputValueClass(DatatypeB.class);
        job.setOutputKeyClass(DatatypeC.class);
        job.setOutputValueClass(DatatypeD.class);
        job.setInputFormatClass(MyInputFormat.class);        // κλάση InputFormat
        job.setOutputFormatClass(MyOutputFormat.class);    // κλάση OutputFormat
        MyInputFormat.addInputPath(job, path_in);            //directory εισόδου
        MyOutputFormat.setOutputPath(job, path_out);        //directory εξόδου
        job.setPartitionerClass(MyPartitioner.class);
        job.setGroupingComparatorClass(MyGroupingComparator.class);
        job.waitForCompletion(true);
}
```

Πίνακας 3.2: Γενική μορφή της main() στο Hadoop framework.

```
public class MyMapper extends Mapper<DataTypeA, DataTypeB, DataTypeC, DataTypeD> {

        public void setup (Context context) throws IOException, InterruptedException {
        }                   // εκτελείται μια φορά κατά την αρχικοποίηση της διεργασίας


        public void map(DatatypeA key, DatatypeB value, Context context)
        throws IOException, InterruptedException {
                ….. εκτέλεση επεξεργασία με key,  value …
                …..
                …..
                DatatypeC output_key = ...;
                DatatypeD output_value = ...;
                context.write(output_key, output_value);
        }
}
```

Πίνακας 3.3: Γενική μορφή της Map στο Hadoop framework



Εικόνα 3.3: Η ροή δεδομένων στο Hadoop MapReduce.

3.4 Hadoop Distributed FileSystem – HDFS

Το HDFS (Hadoop Distributed FileSystem) είναι ένα σύστημα αρχείων σχεδιασμένο για αποθήκευση πολύ μεγάλων αρχείων δεδομένων (της τάξης εκατοντάδων megabytes, gigabytes, ή terabytes) σε cluster κοινών υπολογιστών (commodity hardware). Προσφέρει βελτιστοποιημένη πρόσβαση στα δεδομένα μέσω streaming τεχνικών, φορητότητα (portability) ανάμεσα σε διάφορες πλατφόρμες, και ανεκτικότητα στις αστοχίες. Τρέχει πάνω από το filesystem του λειτουργικού συστήματος και είναι σχεδιασμένο σύμφωνα με

τη master/slave αρχιτεκτονική ([21,], [24]).

Όπως και κάθε σύστημα αρχείων, τα αρχεία στο HDFS αποτελούνται από κομμάτια που είναι αποθηκευμένα σαν ανεξάρτητες μονάδες (κατά προτίμηση σε διαφορετικούς κόμβους του cluster για ισοκατανομή των δεδομένων). Ένα τυπικό μέγεθος block στο HDFS είναι 64 Megabytes, και αντιστοιχεί στην μικρότερη ποσότητα δεδομένων που μπορεί να γράψει ή να διαβάσει το HDFS. Το μεγάλο μέγεθος του block στο HDFS, ευνοεί την αποδοτική προσπέλαση σε μεγάλου όγκου δεδομένα. Για αξιοπιστία έναντι αστοχιών υλικού και εξυπηρέτηση της αρχής τοπικότητας (data locality), κάθε block αντιγράφεται πάνω από μια φορά (τυπικά 3) σε διαφορετικούς κόμβους του cluster [25].

3.5 Hadoop Concepts

Ορισμένες βασικές έννοιες του Hadoop παρουσιάζονται παρακάτω. [26]

- ✓ Namenode and Datanodes:

  Ένας HDFS cluster, λειτουργεί με βάση το μοντέλο Master/Slaves. Προκειμένου να παρέχει την λειτουργικότητα του HDFS, αναθέτει σε έναν κόμβο τον ρόλο του Namenode, και στους άλλους κόμβους του το ρόλο των Datanodes. Ο Namenode (Master) έχει το ρόλο διαχειριστή του filesystem. Κρατάει αρχείο με τις τοποθεσίες που βρίσκεται κάθε block στους κόμβους, διάφορα μεταδεδομένα για τα αρχεία και έχει τον έλεγχο των λειτουργιών εισόδου/εξόδου. Οι Datanodes (Slaves) είναι οι κόμβοι που έχουν αποθηκευμένα τα δεδομένα και παρέχουν λειτουργίες αποθήκευσης κι ανάκτησής τους.

- ✓ Jobtracker και Tasktrackers**:**

  Το framework παρέχει 2 ειδών διεργασίες στους κόμβους του cluster που αναλαμβάνουν τον έλεγχο της εκτέλεσης μιας εργασίας στο Hadoop: ένας Jobtracker και ένας αριθμός από Tasktrackers. Ο jobtracker (με ρόλο master) συντονίζει τις εργασίες που τρέχουν στον cluster, δρομολογώντας την εκτέλεση map και reduce tasks σε tasktrackers (με ρόλο slaves). Οι tasktrackers τρέχουν τα tasks, στέλνοντας αναφορά προόδου στον jobtracker. Σε περίπτωση που η διεργασία αποτύχει στην εκτέλεσή της, ο jobtracker επαναδρομολογεί το task σε έναν διαφορετικό tasktracker.

- ✓ Τύποι Δεδομένων και αρχείων στο HDFS:

Το Hadoop παρέχει ένα σύνολο από βασικούς τύπους δεδομένων, οι οποίοι αντιστοιχούν στους βασικούς τύπους δεδομένων που παρέχει η Java, αλλά είναι βελτιστοποιημένοι για σειριοποίηση και μεταφορά διαμέσου του δικτύου του cluster. Σύμφωνα με την γενική μορφή δεδομένων <κλειδί, τιμή> που διαχειρίζεται το Hadoop, οι τύποι δεδομένων που αντιστοιχούν σε κλειδιά υλοποιούν την διεπαφή WritableComparable, ενώ οι τύποι δεδομένων που αντιστοιχούν σε τιμές υλοποιούν την διεπαφή Writable. Το Hadoop παρέχει ένα σύνολο από βασικούς τύπους δεδομένων, κατ' αντιστοιχία με τους βασικούς τύπους δεδομένων που παρέχει η Java, αλλά ο προγραμματιστής μπορεί να ορίσει νέους τύπους για τις ανάγκες της εφαρμογής του.

Επιπλέον, το Hadoop παρέχει κλάσεις με υλοποιήσεις που αντιστοιχούν σε διάφορες μορφές αρχείων, για αποθήκευση δεδομένων σε δυαδική μορφή ή υπό μορφή κειμένου. Τα αρχεία στο Hadoop γράφονται μια φορά αλλά μπορούν να προσπελαστούν πολλές φορές.

4. Κατασκευή Packed Hilbert R-tree σε περιβάλλον Hadoop MapReduce

4.1 Διάρθρωση της υλοποίησης

Ο κώδικας της υλοποίησής μας είναι χωρισμένος σε πακέτα που λειτουργούν σαν encapsulation layers, αναλαμβάνοντας από ένα κομμάτι λειτουργικότητας.

Για την εκτέλεση των επιμέρους λειτουργιών του R-tree (business logic) και την διαχείριση της αποθήκευσης του δέντρου στο μέσο που θα επιλέγαμε αναπτύξαμε τον κώδικα που συμπεριλαμβάνεται στο πακέτο spatialindex της εφαρμογής. Ως βάση χρησιμοποιήσαμε την εξαιρετική βιβλιοθήκη Spatial Index Library version 0.44.2b που ανέπτυξε ο κ. Marios Hadjieleftheriou σε Java και υποστηρίζει αξιόπιστες μεθόδους δεικτοδότησης (Indexes) χωρικών αντικειμένων μέσω ενός επεκτάσιμου framework. Όλες οι διεπαφές και οι κλάσεις που εκφράζουν χωρικές οντότητες του Index, κόμβους του δέντρου, αλλά και οι κλάσεις που ορίζουν την λειτουργικότητα των R-trees βρίσκονται εδώ. Επίσης, εδώ γίνεται η διαχείριση της αποθήκευσης των χωρικών δεδομένων και των πληροφοριών του R-tree στο μέσο που θα επιλέξουμε. Στην υπάρχουσα λειτουργικότητα της βιβλιοθήκης προσθέσαμε κλάσεις για την υποστήριξη της λογικής Packed Hilbert R-tree και για υποστήριξη αποθήκευσης στο HDFS. Το κεφάλαιο 4.2 παρουσιάζει αναλυτικά την βοηθητική αυτή βιβλιοθήκη, όπως την προσαρμόσαμε για τις ανάγκες της διπλωματικής εργασίας αυτής.

Το κεφάλαιο 4.3 παρουσιάζει την υλοποίηση της κατασκευής ενός Packed Hilbert Rtree. Η λειτουργικότητα προσφέρεται από την κλάση Rtree_bulkLoad του default πακέτου της εφαρμογής. Η κλάση αυτή χρησιμοποιεί στιγμιότυπα κλάσεων του πακέτου spatialindex για την αρχικοποίηση ενός νέου δέντρου και την μαζική φόρτωση σ' αυτό δεδομένων. Τέλος, το R-tree αποθηκεύεται στο HDFS χρησιμοποιώντας στιγμιότυπα των κλάσεων του προσθέσαμε στο πακέτο spatialindex για υποστήριξη αποθήκευσης στο HDFS.

4.2 Η Βοηθητική Βιβλιοθήκη Spatial Index

Για τις ανάγκες της παρούσας διπλωματικής, χρειαζόμασταν μια επιμέρους βιβλιοθήκη που θα προσαρμόζαμε στην υλοποίησή μας και θα αναλάμβανε την διαχείριση των

λειτουργιών του R-tree μας. Η επιμέρους αυτή οντότητα θα είχε αφενός την ευθύνη της εκτέλεσης των επιμέρους λειτουργιών του R-tree (business logic) και αφετέρου την διαχείριση της αποθήκευσης του δέντρου στο μέσο που θα επιλέγαμε.

Για την υλοποίηση του πακέτου αυτού, χρησιμοποιήσαμε ως βάση την εξαιρετική βιβλιοθήκη Spatial Index Library version 0.44.2b που ανέπτυξε ο κ. Marios Hadjieleftheriou σε Java και διανέμεται ως ελεύθερο λογισμικό με άδεια Lesser General Public License [27]. Σ' αυτήν προσθέσαμε πρόσθετη λειτουργικότητα, η οποία θα παρουσιαστεί εκτενώς στις υποενότητες 4.2.4 και 4.2.5, με στόχο να εξυπηρετήσει τους στόχους αυτής της εργασίας.

Οι στόχοι που εξυπηρετεί η βιβλιοθήκη Spatial Index είναι οι εξής:

- ✓ Υποστήριξη αξιόπιστων μεθόδων δεικτοδότησης (Indexes) χωρικών αντικειμένων, μέσω ενός επεκτάσιμου framework.
- ✓ Υποστήριξη της εκτέλεσης απαιτητικών χωρικών επερωτημάτων με χρήση του Index, όπως επερωτήματα εύρους (Range query), σημείου (Point query), k-κοντινότερου γείτονα (k-nearest neighbor query), και άλλων που μπορεί να ορίσει ο χρήστης.
- ✓ Εύκολες λειτουργίες εισαγωγής/διαγραφής/ενημέρωσης του Index μέσω κατάλληλων διεπαφών.
- ✓ Δυνατότητα του χρήστη να διαμορφώνει αυτός ένα μεγάλο εύρος ποιοτικών παραμέτρων. Όλες τα βασικά ποιοτικά χαρακτηριστικά του Index και της μεθόδου αποθήκευσης (όπως το μέγεθος της σελίδας αποθήκευσης, η χωρητικότητα των κόμβων του δέντρου και άλλες) είναι παραμετροποιήσιμα.
- ✓ Υποστήριξη διαφόρων μεθόδων αποθήκευσης του δέντρου. Ο χρήστης μπορεί να επιλέξει δομές αποθήκευσης του Index στη μνήμη, στο δίσκο ή να ορίσει δικές του.

Η προϋπάρχουσα δομή της βιβλιοθήκης περιλαμβάνει τα ακόλουθα 3 πακέτα:

- ✓ Πακέτο spatialindex: Περιέχει τις διεπαφές τις οποίες πρέπει να υλοποιεί κάθε Index.
- ✓ Πακέτο rtree: Περιέχει υλοποιήσεις των διεπαφών του spatialindex παρέχοντας έτσι πλήρη λειτουργικότητα για κατασκευή R-trees.
- ✓ Πακέτο storagemanager: Περιέχει γενικές διεπαφές τις οποίες πρέπει να υλοποιεί κάθε οντότητα υπεύθυνη για την αποθήκευση ενός Index σε κάποιο μέσο (μνήμη, δίσκος, ή οτιδήποτε άλλο ορίσει ο χρήστης π.χ. HDFS). Η βιβλιοθήκη παρέχει έτοιμες υλοποιήσεις επίσης οντοτήτων διαχείρισης αποθήκευσης (storage manager) στη μνήμη και το δίσκο.

Επίσης, στα παραπάνω πακέτα περιέχονται κάποιες βασικές βοηθητικές κλάσεις (spatial Index Utilities) που υπηρετούν γενικές ανάγκες.

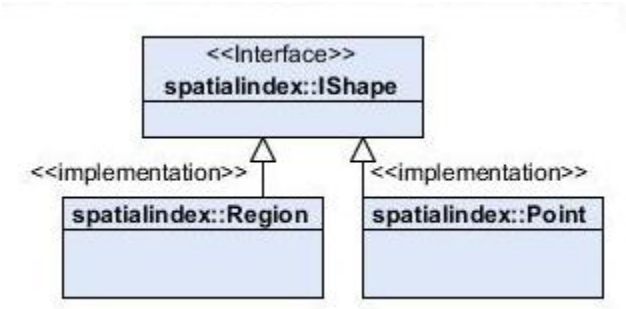
Στις υποενότητες του κεφαλαίου αυτού που ακολουθούν, θα παρουσιάσουμε τα βασικά χαρακτηριστικά της βιβλιοθήκης και την προϋπάρχουσα λειτουργικότητα που

υποστηρίζεται από κάθε πακέτο. Τέλος, θα αναπτύξουμε την πρόσθετη λειτουργικότητα που ενσωματώσαμε εμείς στην βιβλιοθήκη ώστε να υποστηρίξει την ανάπτυξη Packed Hilbert R-trees (στον δίσκο, αλλά και στο HDFS μέσω του framework Hadoop).

4.2.1 Προϋπάρχουσα Λειτουργικότητα: Πακέτο SpatialIndex

Το πακέτο Spatialindex παρέχει γενικές διεπαφές (interfaces) για την ανάπτυξη χωρικών δομών δεικτοδότησης (spatial indexes) οποιασδήποτε μορφής, όπως παραδείγματος χάριν ενός R-tree. Περιλαμβάνει διεπαφές που μπορούν να θεωρηθούν ως οδηγός για την υλοποίηση κλάσεων που εκφράζουν τους κόμβους ενός δέντρου, δεδομένα, κτλ ανάλογα με την δομή του Index.

Το πακέτο παρέχει κλάσεις-διεπαφές που εκφράζουν κόμβους δέντρου (INode), δεδομένα (IData) ή γεωμετρικά σχήματα (IShape) και κληρονομούν την γενική κλάση-διεπαφή IEntry. Φυσικά, ο προγραμματιστής μπορεί να προσθέσει και άλλες που τυχόν χρειάζεται η δομή δεικτοδότησης που θέλει να υλοποιήσει.



Εικόνα 4.1: Διάγραμμα Κλάσεων που απεικονίζει τις διεπαφές που αναπαριστάνουν τα χωρικά δεδομένα.

Παρέχεται επίσης μια διεπαφή ISpatialIndex, την οποία πρέπει να υλοποιεί κάθε χωρικό Index που θα δημιουργήσουμε. Η διεπαφή αυτή ορίζει τις επικεφαλίδες συναρτήσεων που αντιστοιχούν στις βασικές λειτουργίες που πρέπει να στηρίζει κάθε Index: εισαγωγή δεδομένων, διαγραφή δεδομένων, εντολή σύγχρονης αποθήκευσης του δέντρου (flush) καθώς και λειτουργικότητα που δρομολογεί την εκτέλεση διαφόρων ειδών queries (containment/intersection/point/nearestNeighbor Query). Φυσικά, ο χρήστης μπορεί να εκτελέσει τους δικούς του τύπους επερωτημάτων μέσω μιας συνάρτησης που παρέχεται στην διεπαφή. Τέλος, η διεπαφή ISpatialIndex ορίζει επικεφαλίδες συναρτήσεων για έλεγχο της σωστής δομής του Index (συνάρτηση isIndexValid()) και για εμφάνιση στατιστικών στοιχείων (συνάρτηση getStatistics()), γεγονός που κάνει πιο εύκολη την αποσφαλμάτωση (debugging) της υλοποίησης και τον έλεγχο της σωστής λειτουργίας.

Ο προγραμματιστής μπορεί, επιπρόσθετα από τις διεπαφές του πακέτου που ορίζουν κάποιους τύπους επερωτημάτων, να παραμετροποιήσει τα επερωτήματα χρησιμοποιώντας την διεπαφή IVisitor (βασίζεται στο μοντέλο Visitor που παρουσιάζεται στην επιστημονική δημοσίευση [28]). Η IVisitor παρέχει call-back συναρτήσεις για προσπέλαση και έλεγχο κόμβων ή δεδομένων του Index. Παραδείγματα χρήσης της διεπαφής αυτής περιλαμβάνουν την απεικόνιση γραφικά ενός επερωτήματος ή την καταμέτρηση των κόμβων που προσπελάστηκαν για εξαγωγή στατιστικών στοιχείων.

Επιπλέον, ο προγραμματιστής μπορεί να υλοποιήσει πιο περίπλοκα επερωτήματα χρησιμοποιώντας την διεπαφή IQueryStrategy που στηρίζεται στο μοντέλο στρατηγικής επερωτημάτων της επιστημονικής δημοσίευσης [28]).

Παρακάτω παραθέτουμε κάποιες επεξηγήσεις σχετικά με την λογική πάνω στην οποία βασίζει η διεπαφή ISpatialIndex την ανάπτυξη της λειτουργικότητάς της. Με την ίδια λογική λειτουργούν όλα τα Indexes που μπορεί να υλοποιήσει ο προγραμματιστής, καθώς θα υλοποιούν την διεπαφή αυτή.

- ✓ Χωρικά αντικείμενα που δέχεται το Index: Το Index δεικτοδοτεί αντικείμενα που θα πρέπει να υλοποιούν την διεπαφή IShape που δίνεται στο πακέτο spatialindex. Μέσα στο πακέτο, δίνονται ήδη δύο κλάσεις που υλοποιούν την διεπαφή αυτή και εκφράζουν 2 βασικά γεωγραφικά αντικείμενα: Η κλάση Region που εκφράζει μια περιοχή σχήματος ορθογωνίου, και η κλάση Point που εκφράζει γεωγραφικά σημεία. Να σημειωθεί, ότι και οι δύο υλοποιήσεις αφορούν τον k-διάστατο χώρο και η διάσταση k ορίζεται από τον χρήστη.

- ✓ Εισαγωγή Δεδομένων στο Index:
  Μια καταχώρηση χωρικών δεδομένων μπορεί να εισαχθεί χρησιμοποιώντας την μέθοδο *insertData* της διεπαφής ISpatialIndex. Η μέθοδος δέχεται ως ορίσματα ένα γεωμετρικό σχήμα τύπου IShape, ένα διάνυσμα από bytes όπου ο χρήστης μπορεί να συσχετίσει αν θέλει κάποια δεδομένα με το συγκεκριμένο χωρικό αντικείμενο (η παράμετρος αυτή μπορεί να είναι και null) και τέλος το αναγνωριστικό (identifier ή id) που αντιστοιχεί στο χωρικό αντικείμενο.
  Ανάλογα με την φύση του Index, η συνάρτηση αυτή θα εισάγει το γεωμετρικό σχήμα (το αντικείμενο προς δεικτοδότηση) στην εσωτερική δομή του Index.
  Είναι σημαντικό το γεγονός ότι κάθε αντικείμενο πρέπει να χαρακτηρίζεται από ένα μοναδικό αναγνωριστικό id, που θα ταυτοποιεί το αντικείμενο και θα κάνει δυνατή την εύρεσή του, την διαγραφή ή την ενημέρωσή του. Είναι ευθύνη του χρήστη να παρέχει προς το Index μοναδικά ids συσχετισμένα με τα δεδομένα.

- ✓ Διαγραφή Δεδομένων στο Index:
  Μια καταχώρηση χωρικών δεδομένων μπορεί να διαγραφεί χρησιμοποιώντας την μέθοδο *deleteData* της διεπαφής ISpatialIndex. Η μέθοδος δέχεται ως ορίσματα το γεωμετρικό σχήμα τύπου IShape που αντιστοιχεί στην καταχώρηση και το αναγνωριστικό id που αντιστοιχεί στο χωρικό αντικείμενο. Η παροχή πληροφορίας για το σχήμα (το ίδιο το γεωμετρικό αντικείμενο δηλαδή) της καταχώρησης είναι απαραίτητη για την εύρεση του αντικειμένου, καθώς τα χωρικά Indexes ομαδοποιούν τα δεδομένα με βάση κάποια γεωμετρικά χαρακτηριστικά.

- ✓ Χρήση Index ids:
  Όταν δημιουργείται ένα νέο Index, θα πρέπει ένα νέο αναγνωριστικό id να του ανατίθεται που θα χρησιμοποιείται κάθε φορά που θέλουμε να ανακτήσουμε το Index από την μονάδα μόνιμης αποθήκευσης (π.χ. από ένα αρχείο στο δίσκο). Το Index id δεν επιλέγεται απ' τον χρήστη, αλλά δημιουργείται με συγκεκριμένο τρόπο από την βιβλιοθήκη και επιστρέφεται στον χρήστη μέσω ενός στιγμιότυπου της βοηθητικής κλάσης PropertySet.
  Η βοηθητική κλάση PropertySet χρησιμοποιείται για παραμετροποίηση στο index, καθώς ο χρήστης μπορεί να ορίσει παραμέτρους και τις τιμές τους και να τις περάσει μέσω ενός στιγμιότυπου της PropertySet σε διάφορες συναρτήσεις (κυρίως constructors αντικειμένων) της βιβλιοθήκης. Η κλάση αυτή εξυπηρετεί ουσιαστικά τις ανάγκες αρχικοποίησης όλων των στιγμιότυπων κλάσεων της βιβλιοθήκης Spatial Index, καθώς ο χρήστης μπορεί να συσχετίσει συμβολοσειρές (το όνομα μιας παραμέτρου π.χ.) με κλάσεις αντικείμενων (όπου εκφράζουν την τιμή της παραμέτρου).
  Έτσι, η τιμή του id για το νέο Index επιστρέφεται στον χρήστη μέσω της τιμής του πεδίου-παραμέτρου με όνομα IndexIdentifier ενός στιγμιότυπου της κλάσης PropertySet.
  Η χρήση Index ids κάνει δυνατή την αποθήκευση πολλών Indexes στο ίδιο μέσο, όπου μπορούν να ανακτηθούν αργότερα μέσω του μοναδικού id τους.

4.2.2 Προϋπάρχουσα Λειτουργικότητα: Πακέτο Storage Manager

Το πακέτο StorageManager παρέχει γενικές διεπαφές (interfaces) για την διαχείριση της αποθήκευσης των spatial indexes (spatial indexes) σε αποθηκευτικά μέσα οποιασδήποτε μορφής, όπως παραδείγματος χάριν στη μνήμη, το δίσκο, μια βάση δεδομένων ή ένα σύστημα αρχείων όπως το HDFS.

Οι μονάδες διαχείρισης της αποθήκευσης (storage managers) λειτουργούν σαν αυτόνομο κομμάτι προγραμματιστικά, στην λογική διαχωρισμού της λογικής της εκάστοτε εφαρμογής (business logic) από το επίπεδο αποθήκευσης των δεδομένων. Σε όρους

software engineering, ο storage manager λειτουργεί ως ένα επίπεδο αφαίρεσης (Encapsulation Layer) που αναλαμβάνει τις διεργασίες αποθήκευσης των δεδομένων (Persistent Framework). Έτσι, οι storage managers δεν έχουν καμία πληροφορία για τον τύπο των οντοτήτων που αποθηκεύουν/ανακτούν.

Το πακέτο StorageManager παρέχει μια κοινή διεπαφή για την αποθήκευση όλων των χωρικών indexes, την IStorageManager. Η διεπαφή αυτή παρέχει συναρτήσεις για αποθήκευση και ανάκτηση οντοτήτων. Οι οντότητες που διαχειρίζεται ο storage manager αντιμετωπίζονται ως απλά διανύσματα από bytes. Κατ΄ επέκταση, οι οντότητες αντιμετωπίζονται με τον ίδιο τρόπο, είτε είναι μια καταχώρηση ενός κόμβου R-tree, είτε είναι πρόσθετα δεδομένα που σχετίζονται με ένα χωρικό αντικείμενο του Index, ή οτιδήποτε άλλο θέλει να αποθηκεύσει ο χρήστης. Η ευθύνη για την μετατροπή ενός αντικειμένου σε bytestream (και αντίστροφα) δεν ανήκει στον storagemanager, αλλά στην κλάση που υλοποιεί την διεπαφή ISpatialIndex.

Οι κλάσεις που υλοποιούν την διεπαφή IStorageManager καθορίζουν τον τρόπο με τον οποίο θα αποθηκεύονται οι οντότητες. Για παράδειγμα, ένας storage manager μνήμης θα αποθήκευε τις οντότητες σε μια δομή vector στη μνήμη, συνδέοντας κάθε αντότητα με ένα μοναδικό αναγνωριστικό id. Αντίστοιχα, ένας storage manager δίσκου θα αποθήκευε οντότητες σε αρχεία τύπου random access file, ενώ ένας storage manager βάσης δεδομένων θα αποθήκευε οντότητες σε έναν σχεσιακό πίνακα. Σε κάθε περίπτωση, η κάθε υλοποίηση storage manager θα πρέπει να συσχετίζει κάθε οντότητα (οποιουδήποτε τύπου) σε ένα μοναδικό id. Τέλος, θα πρέπει να υλοποιεί την δική της στρατηγική οργάνωσης των δεδομένων προς αποθήκευση σε σελίδες (paging) και ορθής εκμετάλλευσης του αποθηκευτικού μέσου, χωρίς να εμπλέκεται σε αυτή την διαδικασία αυτός που παρέχει τις οντότητες για αποθήκευση.

Προσοχή στην εξής λεπτομέρεια: το id που αναθέτει ο storage manager είναι το id της της οντότητας του index, π.χ. το id ενός κόμβου ενός δέντρου. Αντίθετα, το id ενός χωρικού δεδομένου ανατίθεται από τον χρήστη και αποτελεί την ταυτότητα π.χ. του συγκεκριμένου ορθογωνίου. Για παράδειγμα, το ορθογώνιο με id=1987 βρίσκεται αποθηκευμένο στον κόμβο-φύλλο του δέντρου με id=3.

Παρακάτω παραθέτουμε κάποιες επεξηγήσεις σχετικά με την λογική πάνω στην οποία βασίζει η διεπαφή IStorageManager την ανάπτυξη της λειτουργικότητάς της. Με την ίδια λογική λειτουργούν όλοι οι storage managers που μπορεί να υλοποιήσει ο προγραμματιστής, καθώς θα υλοποιούν την διεπαφή αυτή.

- ✓ Αποθήκευση δεδομένων: Η συνάρτηση storeByteArray δέχεται ως ορίσματα ένα διάνυσμα από bytes (τα οποία αποτελούν τα δεδομένα που θέλουμε να αποθηκεύσουμε) και ένα αναγνωριστικό id της οντότητας που αποθηκεύεται. Εάν ο χρήστης ορίσει ως τιμή id την τιμή NewPage (στον κώδικα αντιστοιχεί σε id=-1), ο storage manager φτιάχνει ένα νέο μοναδικό id, αποθηκεύει την οντότητα/δεδομένα

και στη συνέχεια συσχετίζει τα δεδομένα αυτά με το νέο id. Αντίθετα, εάν ο χρήστης ορίσει μια ήδη υπάρχουσα τιμή id, ο storage manager γράφει τα νέα δεδομένα στην θέση των παλιών που αντιστοιχούν στο id αυτό. Φυσικά, εάν ο χρήστης ζητήσει εγγραφή χωρίς να δώσει έγγυρη τιμή id (NewPage ή id που ήδη υπάρχει αποθηκευμένο) δημιουργείται εξαίρεση (Java Exception).

- ✓ Ανάκτηση δεδομένων: Η συνάρτηση loadByteArray δέχεται ως ορίσματα το αναγνωριστικό id της οντότητας που θέλουμε να ανακτήσουμε. Εάν ο χρήστης ορίσει μια ήδη υπάρχουσα τιμή id, ο storage manager ανακτά τα δεδομένα που αντιστοιχούν στο id αυτό. Φυσικά, εάν ο χρήστης ζητήσει ανάκτηση χωρίς να δώσει έγγυρη τιμή id (id που ήδη υπάρχει αποθηκευμένο) δημιουργείται εξαίρεση (Java Exception).
- ✓ Διαγραφή δεδομένων: Η συνάρτηση deleteByteArray δέχεται ως ορίσματα το αναγνωριστικό id της οντότητας που θέλουμε να διαγράψουμε. Εάν ο χρήστης ορίσει μια ήδη υπάρχουσα τιμή id, ο storage manager διαγράφει τα δεδομένα που αντιστοιχούν στο id αυτό. Φυσικά, εάν ο χρήστης δεν δώσει έγγυρη τιμή id (id που ήδη υπάρχει αποθηκευμένο) δημιουργείται εξαίρεση (Java Exception).

Το πακέτο storagemanager παρέχει δύο έτοιμες υλοποιήσεις της διεπαφής IStorageManager:

- ✓ MemoryStorageManager:
  Αναλαμβάνει την αποθήκευση του Index στην κύρια μνήμη, χρησιμοποιώντας την δομή δεδομένων Vector που παρέχει η Java. Η υλοποίηση είναι αρκετά απλή, χωρίς να απαιτείται παραμετροποίηση από τον χρήστη. Προφανώς, η «διάρκεια ζωής» του Index είναι ίση με την διάρκεια ζωής του στιγμιότυπου MemoryStorageManager – μετά, όλα τα δεδομένα χάνονται.

- ✓ DiskStorageManager
  Αναλαμβάνει την αποθήκευση του Index στον δίσκο, χρησιμοποιώντας δύο αρχεία τύπου random access files για την αποθήκευση δεδομένων και πληροφοριών για την δομή του Index.
  Μια χαρακτηριστική παράμετρος ενός DiskStorageManager είναι το μέγεθος της σελίδας αποθήκευσης (αντιστοιχεί σαν ιδέα στην έννοια της σελίδα ενός δίσκου-page size). Εκφράζει το μικρότερο μέγεθος δεδομένων που μπορεί να γραφεί/ανακτηθεί από το μέσο αποθήκευσης και καθορίζεται απ' τον χρήστη κατά τη δημιουργία του. Κάθε οντότητα που αποθηκεύεται στον storage manager (ως διάνυσμα bytes) μπορεί, ανάλογα με το μέγεθός της, να καταλαμβάνει περισσότερες από μια σελίδες. Έτσι, το id της κάθε οντότητας συσχετίζεται με μια σειρά από σελίδες του δίσκου.

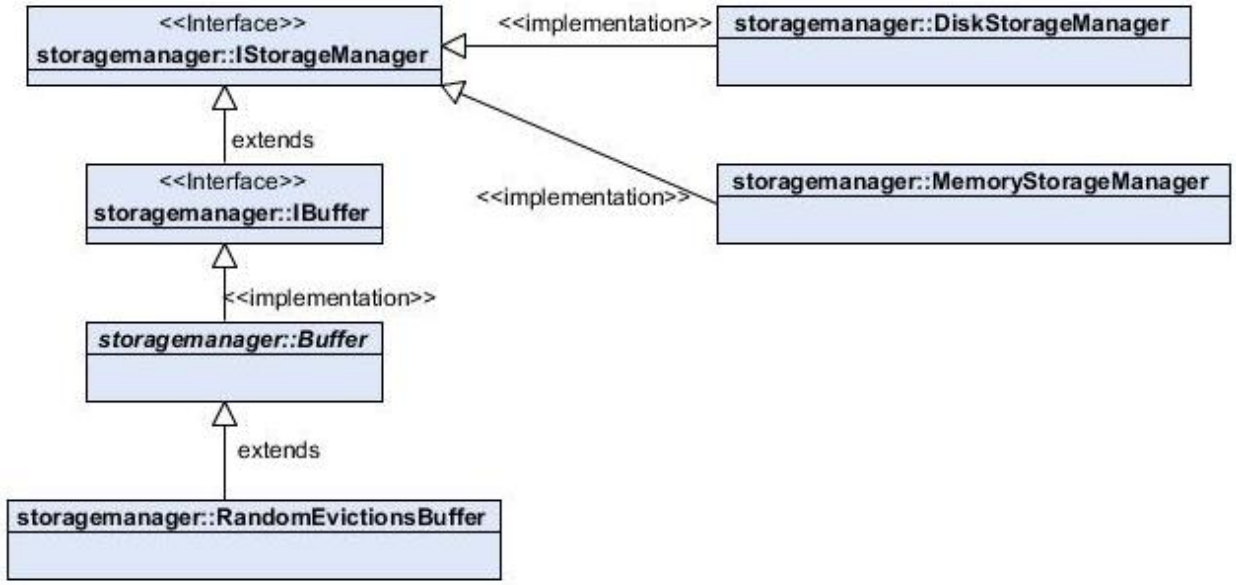
Το ένα αρχείο έχει κατάληξη .idx και αποθηκεύει πληροφορίες για την δομή του index, όπως το μέγεθος της σελίδας αποθήκευσης (page size), μια λίστα με τις άδειες σελίδες, την θέση της επόμενης άδειας σελίδας, και μια λίστα με τις σελίδες στο δίσκο που αντιστοιχούν σε κάθε id δεδομένων. Το αρχείο αυτό φορτώνεται στην μνήμη κατά την αρχικοποίηση του index, και γράφεται πάλι στο δίσκο (flush) στο τέλος της επεξεργασίας ώστε να σώσει τις πληροφορίες για τη δομή του index.
Το άλλο αρχείο έχει κατάληξη .dat και αποθηκεύει τα δεδομένα (π.χ. εάν το index είναι R-tree, τους κόμβους και τα χωρικά δεδομένα που δεικτοδοτούνται).
Παρακάτω παρατίθεται ο Πίνακας 4.1 με τις παραμέτρους που μπορεί να ορίσει ο χρήστης κατά την αρχικοποίηση ενός DiskStorageManager.

Όνομα Παραμέτρου	**Τύπος**	**Περιγραφή**
FileName	String	Το όνομα του αρχείου που θα αποθηκευτεί το Index (χωρίς καταλήξεις .index ή .dat).
Overwrite	Boolean	Εάν έχει τιμή true και το όνομα του αρχείου προϋπάρχει στο δίσκο, τα index θα πανωγραφεί και τα προηγούμενα δεδομένα θα διαγραφούν.
PageSize	Integer	Το μέγεθος σελίδας αποθήκευσης. Εάν το όνομα του αρχείου προϋπάρχει στο δίσκο και η παράμετρος Overwrite ισούται με false, σημαίνει ότι ανακτάμε ένα παλιό Index από το δίσκο, οπότε η παράμετρος αυτή αγνοείται και υιοθετούμε την τιμή PageSize του αποθηκευμένου δέντρου.

Πίνακας 4.1: Οι παράμετροι που μπορεί να ορίσει ο χρήστης κατά την αρχικοποίηση ενός DiskStorageManager.

Τέλος, το πακέτο storagemanager παρέχει μια διεπαφή για υλοποίηση στρατηγικών buffering, την διεπαφή IBuffer. Συγχρόνως , υπάρχει μια υλοποίηση της διεπαφής αυτής, η κλάση Buffer, που μέσω μιας Java δομής HashMap προσφέρει την δυνατότητα ενδιάμεσης αποθήκευσης, ώστε να μειωθεί ο αριθμός των λειτουργιών εισόδου/εξόδου που γίνονται λόγω πρόσβασης στο σκληρό δίσκο. Κάθε φορά που υπάρχει διαθέσιμη μια σελίδα (η οποία μόλις αποθηκεύτηκε ή ανακτήθηκε από τον δίσκο), εάν η σελίδα αυτή δεν υπάρχει στο buffer, προστίθεται. Εάν ο buffer είναι γεμάτος ακολουθείται συγκεκριμένη πολιτική αντικατάστασης. Η κλάση RandomEvictionsBuffer που κληρονομεί την κλάση Buffer, είναι μια υλοποίηση buffer που ακολουθεί πολιτική αντικατάστασης χρησιμοποιώντας

γεννήτρια τυχαίων αριθμών. Στην περίπτωση γεμάτου buffer δηλαδή, η θέση της σελίδας που θα αντικατασταθεί επιλέγεται με τυχαίο τρόπο μέσω της γνωστής Java κλάσης Random. Φυσικά, ο προγραμματιστής μπορεί να υλοποιήσει μια κλάση που θα κληρονομεί την κλάση Buffer και θα έχει την δική της πολιτική αντικατάστασης (replacement policy).



Εικόνα 4.2: Διάγραμμα Κλάσεων του πακέτου storagemanager που απεικονίζει τις διεπαφές που διαχειρίζονται την αποθήκευση του Index.

4.2.3 Προϋπάρχουσα Λειτουργικότητα: Πακέτο rtree

Το πακέτο rtree παρέχει διεπαφές και κλάσεις που αποτελούν ένα πλαίσιο για την κατασκευή δομών δεικτοδότησης τύπου R-tree. Το R-tree που υποστηρίζεται αποτελείται από εσωτερικούς κόμβους, κόμβους φύλλα και χωρικά δεδομένα. Κάθε κόμβος (είτε εσωτερικός είτε φύλλο) έχει συγκεκριμένη χωρητικότητα για καταχωρήσεις (node capacity) που ορίζεται ως παράμετρος από το χρήστη κατά τη δημιουργία του νέου δέντρου.

Το R-tree χειρίζεται τα χωρικά δεδομένα χρησιμοποιώντας το ελάχιστο ορθογώνιο που τα περικλείει (Minimum Bounding Region ή MBR) και ομαδοποιεί αυτά τα MBRs στους κόμβους φύλλα του. Η εκτέλεση επερωτημάτων με χρήση του R-tree, γίνεται ιεραρχικά, ξεκινώντας από τη ρίζα του δέντρου και καταλήγοντας στα φύλλα του (επίπεδο 0).

Το R-tree είναι ισορροπημένο, πράγμα που σημαίνει ότι οι κόμβοι του μπορεί να μην είναι όλοι γεμάτοι (δεν μπορούν να είναι άδειοι ωστόσο). Ο παράγοντας πληρότητας (fill factor)

στις περισσότερες παραλλαγές κυμαίνεται στο 70% του συνολικού χώρου κάθε κόμβου, και εκφράζει το ποσοστό επί της συνολικής χωρητικότητας του κόμβου που μπορεί να είναι γεμάτο. Αυτή την τιμή (η οποία λειτουργεί ως πάνω όριο) μπορεί να την ορίσει ο χρήστης κατά την δημιουργία του δέντρου, χρησιμοποιώντας την παράμετρο fill factor. Τέλος, η συνολική χωρητικότητα των κόμβων του δέντρου (fan-out) μπορεί να οριστεί από το χρήστη μέσω των παραμέτρων node capacity και leaf capacity.

Συνολικά, ο χρήστης κατά την δημιουργία του R-tree καθορίζει:

- ✓ Το μέσο που θα αποθηκευτεί το δέντρο, ορίζοντας και αρχικοποιώντας τον κατάλληλο storage manager.
- ✓ Την τιμή των παραμέτρων node capacity και leaf capacity.
- ✓ Την τιμή της παραμέτρου fill factor (μπορεί να κυμανθεί από 1% εως 100% της node capacity και leaf capacity).
- ✓ Την διάσταση των χωρικών δεδομένων στον k-διάστατο χώρο.
- ✓ Την παραλλαγή του R-tree που θα χρησιμοποιηθεί. Ανάλογα με την παράμετρο αυτή (με όνομα TreeVariant) ο χρήστης μπορεί να επιλέξει ανάμεσα από τις έτοιμες παραλλαγές που δίνονται υλοποιημένες (Linear, Quadratic ή R*) ή να επιλέξει μια δικιά του υλοποίηση παραλλαγής R-tree.

Οι παραπάνω επιλογές δίνονται μέσω ενός στιγμιότυπου της κλάσης PropertySet που περιέχει τις παρακάτω παραμέτρους:

Όνομα Παραμέτρου	**Τύπος**	**Περιγραφή & προεπιλεγμένες τιμές**
IndexIndentifier	Integer	Εάν οριστεί τιμή, ο storage manager θα προσπαθήσει να ανακτήσει το υπάρχον R-tree που αντιστοιχεί στο δοσμένο id.
Dimension	Integer	Η διάσταση των χωρικών δεδομένων στον k-διάστατο χώρο. Default τιμή:2.
IndexCapacity	Integer	Η συνολική χωρητικότητα των εσωτερικών κόμβων του δέντρου. Default τιμή:100.
LeafCapacity	Integer	Η συνολική χωρητικότητα των κόμβων φύλλων του δέντρου.

		Default τιμή:100.
FillFactor	Double	Το ποσοστό-παράγοντας πληρότητας εκφρασμένος ως δεκαδικός αριθμός. Default τιμή:0.7 που αντιστοιχεί σε 70%.
TreeVariant	Integer	Η παραλλαγή του R-tree που θα χρησιμοποιηθεί (Linear, Quadratic ή R*). Default τιμή: R*.
NearMinimumOverlapFactor	Integer	Παράμετρος που σχετίζεται με τον αλγόριθμο εισαγωγής δεδομένων. Default τιμή: 32.
SplitDistributionFactor	Double	Παράμετρος που σχετίζεται με τον αλγόριθμο εισαγωγής/διαγραφής δεδομένων. Default τιμή: 0.4.
ReinsertFactor	Double	Παράμετρος που σχετίζεται με τον αλγόριθμο εισαγωγής/διαγραφής δεδομένων. Default τιμή: 0.3.

Πίνακας 4.2: Οι παράμετροι που ορίζονται κατά την δημιουργία (ή ανάκτηση) ενός R-tree.

Το R-tree, όπως κάθε Index της βιβλιοθήκης, συσχετίζεται με ένα μοναδικό id. Στην περίπτωση του R-tree, το id του ισούται με το id του κόμβου-ρίζας του. Έτσι, έχοντας το id του δέντρου, μπορούμε να ανακτήσουμε την ρίζα από το μέσο αποθήκευσης και διαδοχικά όλο το δέντρο.

Όταν ένα υπάρχον R-tree θέλουμε λοιπόν να ανακτηθεί από το αποθηκευτικό μέσο, χρειάζεται να δώσουμε μόνο το id του και ο storage manager θα το ανακτήσει. Στην περίπτωση αυτή, κάποιες από τις παραμέτρους του αποθηκευμένου δέντρου δεν μπορούν να αλλάξουν, αλλά ανακτώνται όπως ήταν κατά την δημιουργία του. Οι παράμετροι αυτές είναι οι LeafCapacity, IndexCapacity, FillFactor και Dimension. Ωστόσο, το δέντρο μπορεί να αλλάξει π.χ. τύπο, καθώς ο τύπος του R-tree (Linear, Quadratic ή R*) παίζει ρόλο μόνο όταν (και με ποιο τρόπο) γίνονται εισαγωγές δεδομένων που προκαλούν διασπάσεις και ανακατατάξεις των κόμβων.

4.2.4 Πρόσθετη Λειτουργικότητα: Υλοποίηση Packed Hilbert R-tree στο δίσκο.

Ο αντικειμενικός σκοπός της παρούσας διπλωματικής ήταν η ανάπτυξη ενός Packed Hilbert R-tree με χρήση του framework Hadoop, που θα αποθηκεύεται στο HDFS. Και φυσικά σε δεύτερο στάδιο η ανάπτυξη στρατηγικών για επεξεργασία επερωτημάτων με χρήση του R-tree που κατασκευάστηκε. Ωστόσο, πριν την υλοποίηση του αλγορίθμου όπου το R-tree αποθηκεύεται στο HDFS, ενσωματώσαμε στη βιβλιοθήκη spatial Index μια υλοποίηση για κατασκευή Packed Hilbert R-tree [7] που αποθηκεύεται στον τοπικό δίσκο.

Ο λόγος για την υλοποίηση αυτή ήταν η πεποίθηση ότι θα βοηθούσε στον έλεγχο της σωστής λειτουργίας του αλγορίθμου Packed Hilbert R-tree που θα αναπτύσσαμε. Χρησιμοποιώντας την οντότητα που δίνεται έτοιμη για αποθήκευση ενός index στο δίσκο (κλάση DiskStorageManager του πακέτου storagemanager της βιβλιοθήκης) και αναπτύσσοντας την δική μας υλοποίηση για χτίσιμο του Packed Hilbert R-tree, θα ήταν εύκολο να ελέγξουμε την ορθή λειτουργία του. Για τους ελέγχους, αρκούσε να χρησιμοποιήσουμε τα επερωτήματα που υποστηρίζει η βιβλιοθήκη spatial index και να συγκρίνουμε τα αποτελέσματατά τους με τα αναμενόμενα αλλά και με τα αποτελέσματα που δίνουν οι άλλες εκδόσεις R-tree της βιβλιοθήκης όταν εκτελούν τα ίδια επερωτήματα στα ίδια χωρικά δεδομένα. Αργότερα, σίγουροι ότι η βασική λογική για το χτίσιμο του Packed Hilbert R-tree ήταν σωστή, θα μπορούσαμε να την προσαρμόσουμε στις ιδιαιτερότητες του Hadoop.

Ο αλγόριθμος για την κατασκευή Packed Hilbert R-tree στο δίσκο υλοποιήθηκε σε πλήρη εναρμόνιση με τη υπόλοιπη βιβλιοθήκη Spatial Index. Ενσωματώθηκε σ' αυτήν χωρίς να επηρεάσει την σωστή λειτουργία των υπόλοιπων τμημάτων της. Η υλοποίησή μας χρησιμοποιεί τις υπάρχουσες συναρτήσεις ή νέες που ορίσαμε εμείς, πάντα όμως με την ίδια λογική αρχικοποίησης, λειτουργίας και αποθήκευσης που χρησιμοποιούν και οι άλλες παραλλαγές R-tree που υποστηρίζει η βιβλιοθήκη.

Παρακάτω θα παρουσιάσουμε την λειτουργικότητα της υλοποίησής μας, περιγράφοντας όπου χρειάζεται περισσότερο τις λεπτομέρειες υλοποίησης.

- ✓ **Αρχικοποίηση Packed Hilbert R-tree:**

  Η οντότητα του δέντρου αναπαρίσταται από την κλάση Rtree που υπάρχει στο πακέτο rtree της βιβλιοθήκης. Για την αρχικοποίησή της, ο χρήστης πρέπει να παρέχει ως όρισμα στη συνάρτηση-constructor δύο πράγματα: ένα στιγμιότυπο μιας κλάσης που να υλοποιεί την διεπαφή IstorageManager (που διαχειρίζεται την αποθήκευση του δέντρου σε κάποιο επιλεγμένο μέσο) και ένα στιγμιότυπο της κλάσης PropertySet για παραμετροποίηση του δέντρου.

  Μια κλάση που υλοποιεί την διεπαφή IstorageManager είναι η κλάση Buffer, η οποία παρέχει ένα ενδιάμεσο επίπεδο buffering και έναν storage manager. Στην

περίπτωσή μας, χρησιμοποιούμε ως storage manager την κλάση DiskStorageManager, που διαχειρίζεται την αποθήκευση του δέντρου στο δίσκο. Η παραμετροποίηση του DiskStorageManager γίνεται μέσω ενός στιγμιοτύπου της κλάσης PropertySet, όπου ο χρήστης ορίζει το όνομα του αρχείου που θα αποθηκευτεί (ή από όπου θα ανακτηθεί) το δέντρο, το μέγεθος σελίδας αποθήκευσης και την τιμή της παραμέτρου overwrite.

Για την αρχικοποίηση του Packed Hilbert R-tree, ο χρήστης παρέχει επίσης ένα στιγμιότυπο της κλάσης PropertySet που ορίζει τις τιμές ορισμένων παραμέτρων. Οι παράμετροι με τις τιμές τους φαίνονται στον Πίνακα 4.3 που ακολουθεί.

Όνομα Παραμέτρου	**Τύπος**	**Περιγραφή & προεπιλεγμένες τιμές**
IndexIndentifier	Integer	Εάν οριστεί τιμή, ο storage manager θα προσπαθήσει να ανακτήσει το υπάρχον Packed Hilbert R-tree που αντιστοιχεί στο δοσμένο id. Εάν δεν δοθεί η παράμετρος αυτή, ένα νέο Packed Hilbert R-tree φτιάχνεται και επιστρέφεται το id του σαν τιμή της παραμέτρου αυτής μετά το τέλος της επεξεργασίας.
Dimension	Integer	Η διάσταση των χωρικών δεδομένων στον k-διάστατο χώρο. Default τιμή:2.
IndexCapacity	Integer	Η συνολική χωρητικότητα των εσωτερικών κόμβων του δέντρου. Default τιμή:100.
LeafCapacity	Integer	Η συνολική χωρητικότητα των κόμβων φύλλων του δέντρου. Default τιμή:100.
TreeVariant	Integer	Η παραλλαγή του R-tree που θα χρησιμοποιηθεί. Στην περίπτωσή μας, χρησιμοποούμε τον αριθμό 4 που αντιστοιχεί στον τύπο Packed Hilbert R-tree.

writtenOnHDFS	Boolean	Σηματοδοτεί την χρήση του Hadoop ως framework και του HDFS ως αποθηκευτικό μέσο.

Πίνακας 4.3: Παραμέτροι που χρησιμοποιούνται για την αρχικοποίηση του Packed Hilbert R-tree.

Ανάλογα με τις επιλογές του χρήστη, γίνεται η αρχικοποίηση του δέντρου. Στην περίπτωση κατασκευής νέου R-tree, αρχικοποιείται ένα κενό δέντρο με τις προδιαγραφές που έδωσε ο χρήστης και δημιουργούνται τα δυο αρχεία με κατάληξη .idx και .dat στα οποία θα αποθηκευτεί η πληροφορία που σχετίζεται με το δέντρο. Στην περίπτωση που ο χρήστης θέλησε την ανάκτηση ενός δέντρου που ήδη υπάρχει στο δίσκο, το δέντρο αυτό ανακτάται με βάση τις πληροφορίες των αρχείων με κατάληξη .idx και .dat.

- ✓ **Μαζική εισαγωγή χωρικών δεδομένων στο δέντρο (bulk-loading):**

Για την μαζική εισαγωγή δεδομένων σε ένα κενό δέντρο Hilbert R-tree, ο χρήστης καλεί την παρακάτω συνάρτηση και το δέντρο χτίζεται από κάτω προς τα πάνω (bottom-up):

public void bulkLoadData(Vector<byte[]> data, Vector<Region> shapes, Vector<Long> ids) throws IllegalStateException, IOException;

Όπου:

- *Vector<Region> shapes* τα χωρικά αντικείμενα προς δεικτοδότηση, τα οποία εκφράζονται από το MBR τους (Minimum Bounding Rectangle) μέσω της κλάσης Region. Τα αντικείμενα στο Vector είναι ταξινομημένα κατά αύξουσα σειρά ανάλογα με την τιμή Hilbert του κέντρου τους.
- *Vector<Long> ids* τα αντίστοιχα ids που συνδέονται με τα χωρικά δεδομένα, και
- *Vector<byte[]> data* τυχόν επιπρόσθετα δεδομένα σε μορφή διανυσμάτων bytes που συσχετίζονται με κάθε χωρικό αντικείμενο (π.χ. επιπρόσθετα χαρακτηριστικά, σημειώσεις, κτλ). Φυσικά, μπορεί τα διανύσματα αυτά να έχουν τιμή null.

Θεωρούμε ότι πριν την κλήση της συνάρτησης, έχουμε υπολογίσει την τιμή Hilbert κάθε χωρικού αντικειμένου. Ο αλγόριθμος υπολογισμού της τιμής αυτή θα παρουσιαστεί στην ενότητα 4.3.2, στην ανάλυση της υλοποίησης Packed Hilbert R-tree με χρήση του framework Hadoop.

Παρακάτω παρουσιάζεται ο αλγόριθμος για την συνάρτηση *bulkLoadData* που χτίζει το δέντρο bottom-up εισάγοντας μαζικά τα χωρικά δεδομένα που δίνει ο χρήστης.

Αλγόριθμος *bulkLoadData(Vector<byte[]> data, Vector<Region> shapes, Vector<Long> ids)*: packing χωρικών δεδομένων σε ένα Rtree.

Βήμα 1. /* Σχηματισμός των κόμβων-φύλλων (επίπεδο δέντρου L=0) */

Κατασκεύασε ένα κενό Vector με όνομα CurrentLevel_Nodes.

While (υπάρχουν ακόμα ορθογώνια στο Vector shapes) {

Ομαδοποίησε τα επόμενα LeafCapacity ορθογώνια (ή λιγότερα εάν έχουμε φτάσει στο τέλος των Vectors) με τα αντίστοιχα ids και data και αποθήκευσέ τα σε νέα Vectors n_data, n_shapes, και n_ids.

Κατασκεύασε έναν νέο κόμβο-φύλλο newleaf.

Κάλεσε τη συνάρτηση newleaf.bulkLoadMakeLeaf(n_data,n_shapes,n_ids) για να αναθέσει τα ορθογώνια με τα αντίστοιχα ids και data σε αυτόν τον κόμβο-φύλλο.

Πρόσθεσε στο Vector με όνομα CurrentLevel_Nodes ένα στιγμιότυπο της κλάσης EntrySummury, που θα αποθηκεύει το id και το MBR του κόμβου-φύλλου που δημιουργήθηκε.

}

Βήμα 2. /* Σχηματισμός των εσωτερικών κόμβων (επίπεδο δέντρου L+1) */

While (υπάρχουν πάνω από 1 κόμβος στο προηγούμενο επίπεδο L) {

Αντέγραψε τα περιεχόμενα του CurrentLevel_Nodes σε ένα νέο κενό Vector PreviousLevel_Nodes.

Άδειασε τα περιεχόμενα του Vector CurrentLevel_Nodes.

While(υπάρχουν ακόμα ορθογώνια στο Vector PreviousLevel_Nodes){ //μπορώ να φτιάξω έναν ακόμα κόμβο που να τα συμπεριλαμβάνει

Ομαδοποίησε τα επόμενα IndexCapacity στοιχεία του PreviousLevel_Nodes (ή λιγότερα εαν έχουμε φτάσει στο τέλος του vector) και αποθήκευσέ τα σε νέο Vector<EntrySummury> n_ids.

Κατασκεύασε έναν νέο κόμβο newnode.

Κάλεσε τη συνάρτηση newnode.bulkLoadMakeNode(n_ids) για να αναθέσει τα MBRSs με τα αντίστοιχα ids σε αυτόν τον κόμβο.

Πρόσθεσε στο Vector με όνομα CurrentLevel_Nodes. ένα στιγμιότυπο της κλάσης EntrySummury, που θα αποθηκεύει το id και το MBR του κόμβου-φύλλου που δημιουργήθηκε.

}

Προχώρα στο επόμενο επίπεδο L+1 επαναλαμβάνοντας το Βήμα 2.

}

Βήμα 3. Εάν υπάρχει μόνο ένας κόμβος στο προηγούμενο επίπεδο L {

Κάνε αυτόν τον κόμβο ρίζα του δέντρου.

}

Πίνακας 4.4: Αλγόριθμος bulkLoadData

Οι συναρτήσεις bulkLoadMakeLeaf() και bulkLoadMakeNode() που αναφέρονται στον αλγόριθμο, αναλαμβάνουν την φόρτωση ενός συνόλου χωρικών δεδομένων σε ένα φύλλο ή σε έναν εσωτερικό κόμβο. Η συνάρτηση bulkLoadMakeLeaf() ανήκει στην κλάση Leaf. Όταν κληθεί, για κάθε χωρικό αντικείμενο του vector n_shapes που τις δίνεται σαν είσοδο, καλεί την συνάρτηση Node. insertEntry() που εισάγει το αντικείμενο στον κόμβο-φύλλο αποθηκεύοντας το id και το MBR του, ενημερώνει το συνολικό MBR του κόμβου ώστε να συμπεριλάβει το MBR του νέου χωρικού αντικειμένου, και εάν χρειαστεί αποθηκεύει και τα δεδομένα byte[] που συσχετίζονται με το αντικείμενο. Στο τέλος, καλεί την συνάρτηση writeNode() της κλάσης Rtree, που αποθηκεύει τον κόμβο (μαζί με τις αλλαγές) στο αποθηκευτικό μέσο (στον δίσκο στην περίπτωσή μας). Η συνάρτηση bulkLoadMakeNode() της κλάσης Index εκτελεί τις αντίστοιχες ενέργειες αποθηκεύοντας τα χωρικά δεδομένα σε εσωτερικό κόμβο του δέντρου και αποθηκεύοντάς τον μετά.

Φυσικά, κατά την εισαγωγή των χωρικών δεδομένων του δέντρου, ενημερώνονται τα στατιστικά στοιχεία του δέντρου, ώστε να είναι εύκολος ο έλεγχος της σωστής δομής του Index (συνάρτηση isIndexValid()) και η εμφάνιση στατιστικών στοιχείων (συνάρτηση getStatistics()). Έτσι, είναι πιο εύκολη η αποσφαλμάτωση (debugging) της υλοποίησης και ο έλεγχος της σωστής λειτουργίας.

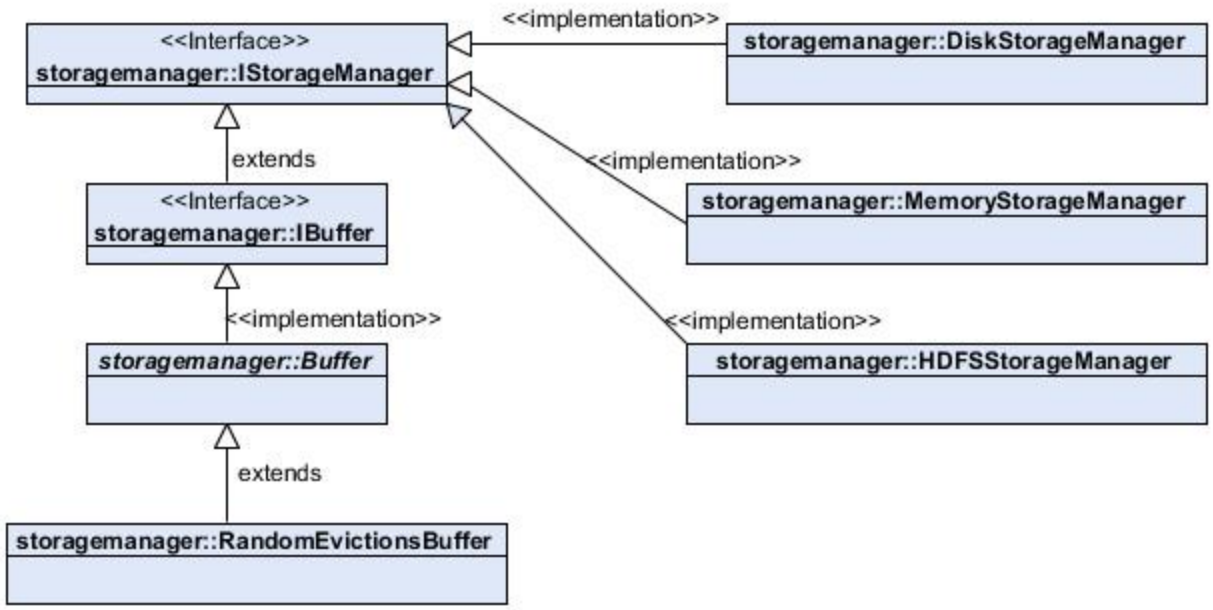
- ✓ **Επεξαργασία επερωτημάτων με χρήση του Packed Hilbert R-tree**

  Ο χρήστης μπορεί, αφού κατασκευάσει το δέντρο (ή το ανακτήσει από το δίσκο), να εκτελέσει μια σειρά από επερωτήματα. Η εκτέλεση είναι ακριβώς η ίδια με την εκτέλεση επερωτημάτων σε άλλες μορφές R-tree που υποστηρίζει η βιβλιοθήκη. Ο χειρισμός των επερωτημάτων γίνεται από την αντίστοιχη συνάρτηση της κλάσης Rtree, ανάλογα με τον τύπο του query. Παραδείγματος χάριν, η συνάρτηση rangeQuery() διαχειρίζεται τα επερωτήματα εύρους (range queries), και ανάλογα με τον τύπο τους π.χ. containment Query/ intersection Query δρομολογεί τις απαραίτητες ενέργειες. Η εύρεση των χωρικών αντικειμένων που ικανοποιούν τις συνθήκες των επερωτημάτων γίνεται χρησιμοποιώντας υλοποιήσεις των διεπαφών IVisitor και IQueryStrategy του πακέτου spatialindex.

4.2.5 Πρόσθετη Λειτουργικότητα: Υλοποίηση Storage Manager για αποθήκευση στο HDFS.

Για την ανάπτυξη ενός Packed Hilbert R-tree με χρήση του framework Hadoop, χρειαζόταν να υλοποιήσουμε έναν Storage Manager που να υλοποιεί την διεπαφή IStorageManager και να αναλαμβάνει την διαχείριση της αποθήκευσης/ανάκτησης του index στο HDFS. Η λειτουργικότητα εκφράζεται από την κλάση HDFSStorageManager. Στην Εικόνα 4.3 που ακολουθεί φαίνεται το διάγραμμα κλάσεων του πακέτου storagemanager της βιβλιοθήκης, μετά την προσθήκη της νέας κλάσης.

Η υλοποίηση του HDFSStorageManager έγινε σε πλήρη εναρμόνιση με τη υπόλοιπη βιβλιοθήκη Spatial Index. Ενσωματώθηκε σ' αυτήν χωρίς να επηρεάσει την σωστή λειτουργία των υπόλοιπων τμημάτων της. Η υλοποίησή μας χρησιμοποιεί τις υπάρχουσες υπογραφές συναρτήσεων για την διαχείριση της αποθήκευσης, ώστε να υπάρχει συμβατότητα και να μπορεί να χρησιμοποιηθεί στη θέση οποιουδήποτε storagemanager, για όλα τα είδη index που υποστηρίζει η βιβλιοθήκη.



Εικόνα 4.3: Διάγραμμα Κλάσεων του πακέτου storagemanager που απεικονίζει τις διεπαφές που διαχειρίζονται την αποθήκευση του Index, μετά την προσθήκη της κλάσης HDFSStorageManager.

Η νέα κλάση HDFSStorageManager φτιάχτηκε για να χρησιμοποιεί ως αποθηκευτικό μέσο το Hadoop Distributed FileSystem (HDFS), οπότε χρησιμοποιεί και συναρτήσεις και κλάσεις που ορίζονται στα πακέτα του project ανοικτού λογισμικού Hadoop.

Η διεπαφή αυτή παρέχει συναρτήσεις για αποθήκευση και ανάκτηση οντοτήτων. Οι οντότητες που διαχειρίζεται ο storage manager αντιμετωπίζονται ως απλά διανύσματα από bytes. Κατ΄ επέκταση, οι οντότητες αντιμετωπίζονται με τον ίδιο τρόπο, είτε είναι μια καταχώρηση ενός κόμβου R-tree, είτε είναι πρόσθετα δεδομένα που σχετίζονται με ένα χωρικό αντικείμενο του Index, ή οτιδήποτε άλλο θέλει να αποθηκεύσει ο χρήστης. Η ευθύνη για την μετατροπή ενός αντικειμένου σε bytestream (και αντίστροφα) δεν ανήκει στον storagemanager, αλλά στην κλάση που υλοποιεί την διεπαφή ISpatialIndex.

Η προσθήκη της HDFSStorageManager για διαχείριση αποθήκευσης στο HDFS, επέβαλλε κάποιες μικρές αλλαγές σε ορισμένα σημεία της βιβλιοθήκης. Συγκεκριμένα, προκειμένου να παρέχουμε τη δυνατότητα αποθήκευσης στο HDFS έπρεπε να εξασφαλίσουμε ότι θα υπηρετούνταν ορισμένες ιδιαιτερότητες του Hadoop χωρίς να επηρεάζεται η υπόλοιπη λειτουργικότητα της βιβλιοθήκης. Για το λόγο αυτό εισήχθη μια μεταβλητή με όνομα writtenOnHDFS, τύπου Boolean, που σηματοδοτεί την χρήση του HDFS ως αποθηκευτικό μέσο. Η μεταβλητή αυτή υπάρχει ως πεδίο στις κλάσεις Rtree του πακέτου rtree και στις κλάσεις Buffer και HDFSStorageManager του πακέτου storagemanager. Με τον τρόπο αυτό, όταν χρησιμοποιούμε το Hadoop και το HDFS μπορούμε να τροποποιήσουμε τη

στρατηγική ανάπτυξης του R-tree, χωρίς να επηρεαστεί η σωστή λειτουργία των R-tree σε άλλα περιβάλλοντα.

Παρακάτω θα παρουσιάσουμε την λειτουργικότητα της υλοποίησής μας, περιγράφοντας όπου χρειάζεται περισσότερο τις λεπτομέρειες υλοποίησης.

- ✓ **Η βασική λογική του HDFSStorageManager**

  Οι κλάσεις που υλοποιούν την διεπαφή IStorageManager καθορίζουν τον τρόπο με τον οποίο θα αποθηκεύονται οι οντότητες. Έτσι, ενώ ο storage manager δίσκου (κλάση DiskStorageManager) αποθηκεύει οντότητες σε αρχεία τύπου random access files, ο HDFSStorageManager χρησιμοποιεί τις κλάσεις FSDataOutputStream και FSDataInputStream που παρέχονται από το Hadoop για την εγγραφή/ανάγνωση μιας ροής bytes σε/από ένα δυαδικό αρχείο αποθηκευμένο στο HDFS.

  Ο λόγος για την επιλογή αυτή ήταν οι περιορισμοί του HDFS όσο αφορά την δημιουργία και την εγγραφή/ανάγνωση σε/από αρχεία. Μη μπορώντας να χρησιμοποιήσουμε τον τύπο αρχείων random access files, χρησιμοποιούμε αρχεία με δυαδική μορφή όπου επιτρέπεται ανάγνωση δεδομένων από τυχαία θέση (random read), αλλά όχι εγγραφή σε τυχαία θέση (random write). Ο storage manager μπορεί να γράψει δεδομένα μόνο σειριακά, εισάγοντας τα νέα δεδομένα στο τέλος του αρχείου. Ως αποτέλεσμα αυτού, τα δεδομένα που εγγράφονται μπορούν αργότερα να διαβαστούν (σε όποιο σημείο του αρχείου και αν βρίσκονται) πολλές φορές, αλλά δεν μπορούν να αλλαχθούν ή να διαγραφούν (μπορούμε να διαγράψουμε μόνο ολόκληρο το αρχείο).

  Η ιδιαιτερότητα αυτή έχει αλυσιδωτή επίδραση στο πώς και πότε μπορούμε να αποθηκεύουμε κάθε σύνολο δεδομένων (κόμβο δέντρου, επιπλέον δεδομένα, πληροφορίες για τη δομή του index). Καταλήγουμε ότι αφού αλλαγές στα περιεχόμενα του αρχείου δεν γίνεται, οι εφαρμογές index που χρησιμοποιούν τον HDFSStorageManager θα πρέπει να αναπροσδιορίσουν την στρατηγική με την οποία στέλνουν δεδομένα για αποθήκευση ή ζητούν δεδομένα για ανάκτηση. Αυτό θα φανεί πιο ξεκάθαρα στην ενότητα 4.3 του κεφαλαίου, όπου περιγράφεται η υλοποίηση του αλγορίθμου για κατασκευή Packed Hilbert R-tree σε περιβάλλον Hadoop.

  Είναι σημαντικό να τονιστεί ότι ο HDFSStorageManager υλοποιεί την δική του στρατηγική οργάνωσης των δεδομένων προς αποθήκευση σε σελίδες (paging) και ορθής εκμετάλλευσης του αποθηκευτικού μέσου, χωρίς να εμπλέκεται σε αυτή την διαδικασία αυτός που παρέχει τις οντότητες για αποθήκευση.

Μια χαρακτηριστική παράμετρος του HDFSStorageManager είναι το μέγεθος της σελίδας αποθήκευσης (page size) και καθορίζεται απ' τον χρήστη κατά τη αρχικοποίηση του storage manager. Ο HDFSStorageManager αποθηκεύει κάθε οντότητα δεδομένων σε μία σελίδα στο αρχείο, σε αντίθεση με άλλους storage managers της βιβλιοθήκης που επιτρέπουν σε ένα αντικείμενο να εκτείνεται σε περισσότερες. Αυτή η επιλογή έγινε αναγκαία λόγω του μεγάλου κόστους που θα είχε η διατήρηση από τον storage manager μιας δομής που θα αποθήκευε τα ids των σελίδων που αντιστοιχούν με το συγκεκριμένο id της οντότητας δεδομένων (mapping από id δεδομένων σε id/θέσεις σελίδων του αρχείου). Το ίδιο το HDFS εξ ορισμού είναι φτιαγμένο για να διαχειρίζεται πολύ μεγάλους όγκους δεδομένων. Οι εφαρμογές indexing λοιπόν που θα καλούνταν να εξυπηρετήσει ο HDFSStorageManager θα απαιτούσαν να διαχειρίζεται δισεκατομμύρια ή τρισεκατομμύρια ids, προκαλώντας ενδεχομένως προβλήματα έλλειψης μνήμης (Out Of Memory exceptions). Για το λόγο αυτό απλουστεύσαμε τον τρόπο που ο HDFSStorageManager αντιστοιχεί ids στα δεδομένα που αποθηκεύει, ώστε να μην χρειάζεται να κρατάμε στην κύρια μνήμη πληροφορίες mapping από id δεδομένων σε ids/θέσεις σελίδων στο αρχείο.

Ο HDFSStorageManager συσχετίζει κάθε οντότητα που αποθηκεύει (οποιουδήποτε τύπου) σε ένα μοναδικό id. Το id αυτό (μια μεταβλητή τύπου long) ισούται με την αρχή της σελίδας που αποθηκεύει τα δεδομένα στο αρχείο του HDFS. Συνεπώς, το id είναι η μοναδική πληροφορία που χρειαζόμαστε για να ανακτήσουμε μια οντότητα δεδομένων από το αρχείο. Ο HDFSStorageManager για την ανάθεση αυτού του id σε κάθε οντότητα που αποθηκεύει, διατηρεί μια μεταβλητή με όνομα m_nextPage (τύπου long), η οποία αποθηκεύει την θέση της επόμενης άδειας σελίδας στο αρχείο του HDFS.

Προσοχή στην εξής λεπτομέρεια: το id που αναθέτει ο storage manager είναι το id της της οντότητας του index, π.χ. το id ενός κόμβου ενός δέντρου. Αντίθετα, το id ενός χωρικού δεδομένου ανατίθεται από τον χρήστη και αποτελεί την ταυτότητα π.χ. του συγκεκριμένου ορθογωνίου. Για παράδειγμα, το ορθογώνιο με id=1987 που βρίσκεται αποθηκευμένο στον κόμβο-φύλλο του δέντρου με id=3.

- ✓ **Αποθήκευση δεδομένων στο HDFS**

Αναλαμβάνει την αποθήκευση του Index στο HDFS, χρησιμοποιώντας δύο δυαδικά αρχεία για την αποθήκευση δεδομένων και πληροφοριών για την δομή του Index. Η εγγραφή των δεδομένων στα αρχεία γίνεται με την βοήθεια δύο στιγμιότυπων της κλάσης FSDataOutputStream που παρέχεται από το Hadoop για την αποθήκευση μιας ροής bytes σε ένα δυαδικό αρχείο αποθηκευμένο στο HDFS.

Το ένα αρχείο έχει κατάληξη .dat και αποθηκεύει τα δεδομένα (π.χ. εάν το index είναι R-tree, τους κόμβους και τα χωρικά δεδομένα που δεικτοδοτούνται).

Το άλλο αρχείο έχει κατάληξη .idx και αποθηκεύει πληροφορίες για την δομή του index, δηλαδή το μέγεθος της σελίδας αποθήκευσης (page size) και την θέση της επόμενης άδειας σελίδας στο αρχείο με κατάληξη .dat του HDFS. Το αρχείο αυτό φορτώνεται στην μνήμη κατά την αρχικοποίηση του index, και γράφεται πάλι στο δίσκο (flush) στο τέλος της επεξεργασίας ώστε να σώσει τις πληροφορίες για τη δομή του index.

Η εγγραφή μιας οντότητας δεδομένων στο αρχείο του HDFS (με κατάληξη . dat όπως είπαμε), γίνεται με την κλήση της συνάρτησης

public long storeByteArray(final long id, final byte[] data) throws IOException.

Η συνάρτηση storeByteArray είναι ουσιαστικά η υλοποίηση της αντίστοιχης συνάρτησης της διεπαφής IStorageManager, προσαρμοσμένη στις ιδιαιτερότητες του HDFS. Δέχεται ως ορίσματα ένα διάνυσμα από bytes (τα οποία αποτελούν τα δεδομένα που θέλουμε να αποθηκεύσουμε) και ένα αναγνωριστικό id της οντότητας που αποθηκεύεται. Εάν ο χρήστης ορίσει ως τιμή id την τιμή NewPage (στον κώδικα αντιστοιχεί σε id=-1), ο storage manager φτιάχνει ένα νέο μοναδικό id που ισούται με την τιμή της μεταβλητής m_nextPage, αποθηκεύει την οντότητα/δεδομένα στο αρχείο .dat και στη συνέχεια συσχετίζει τα δεδομένα αυτά με το νέο id. Φυσικά, εάν ο χρήστης ζητήσει εγγραφή χωρίς να δώσει τιμή id ίση με NewPage, δημιουργείται εξαίρεση (Java Exception) αφού το HDFS δεν επιτρέπει αλλαγή στα περιεχόμενα ενός αρχείου, αλλά μόνο προσάρτηση νέων εγγραφών στο τέλος του.

Φυσικά, μια οντότητα που αποθηκεύεται καταλαμβάνει μόνο μια σελίδα, μπορεί ωστόσο να μην την καταλαμβάνη ολόκληρη, αλλά ένα μέρος της. Για να μπορέσει ο HDFSStorageManager να ανακτήσει σωστά τα δεδομένα από το HDFS, αποθηκεύει τη στιγμή της αποθήκευσης μιας οντότητας και τον αριθμό bytes που καταλαμβάνει (που μπορεί να είναι μικρότερος ή ίσος από PageSize). Έτσι, ενώ κάθε φορά αποθηκεύεται και ανακτάται σταθερός αριθμός από PageSize bytes, ξέρουμε πόσα από αυτά είναι όντως δεδομένα και πόσα στο τέλος του stream είναι άχρηστα bytes.

- ✓ **Ανάκτηση δεδομένων από το HDFS**

  Αναλαμβάνει την ανάκτηση του Index στο HDFS, ανοίγοντας τα δύο δυαδικά αρχεία που περιέχουν τα δεδομένα και τις πληροφορίες για την δομή του Index. Η ανάγνωση των δεδομένων από τα αρχεία γίνεται με την βοήθεια δύο στιγμιότυπων

της κλάσης FSDataInputStream που παρέχεται από το Hadoop για την ανάγνωση μιας ροής bytes από ένα δυαδικό αρχείο αποθηκευμένο στο HDFS.

Η ανάκτηση μιας οντότητας δεδομένων από το αρχείο του HDFS (με κατάληξη . dat όπως είπαμε), γίνεται με την κλήση της συνάρτησης

public long loadByteArray (final long id, final byte[] data) throws IOException.

Η συνάρτηση loadByteArray είναι ουσιαστικά η υλοποίηση της αντίστοιχης συνάρτησης της διεπαφής IStorageManager, προσαρμοσμένη στις ιδιαιτερότητες του HDFS. Δέχεται ως ορίσματα ένα διάνυσμα από bytes (στο οποίο θα αποθηκευτούν τα δεδομένα που θέλουμε να ααανακτήσουμε) και το αναγνωριστικό id της οντότητας που αναζητούμε. Εάν ο χρήστης ορίσει μια ήδη υπάρχουσα τιμή id, ο storage manager ανακτά τα δεδομένα που αντιστοιχούν στο id αυτό. Φυσικά, εάν ο χρήστης ζητήσει ανάκτηση χωρίς να δώσει έγγυρη τιμή id (NewPage ή id που ήδη υπάρχει αποθηκευμένο) δημιουργείται εξαίρεση (Java Exception).

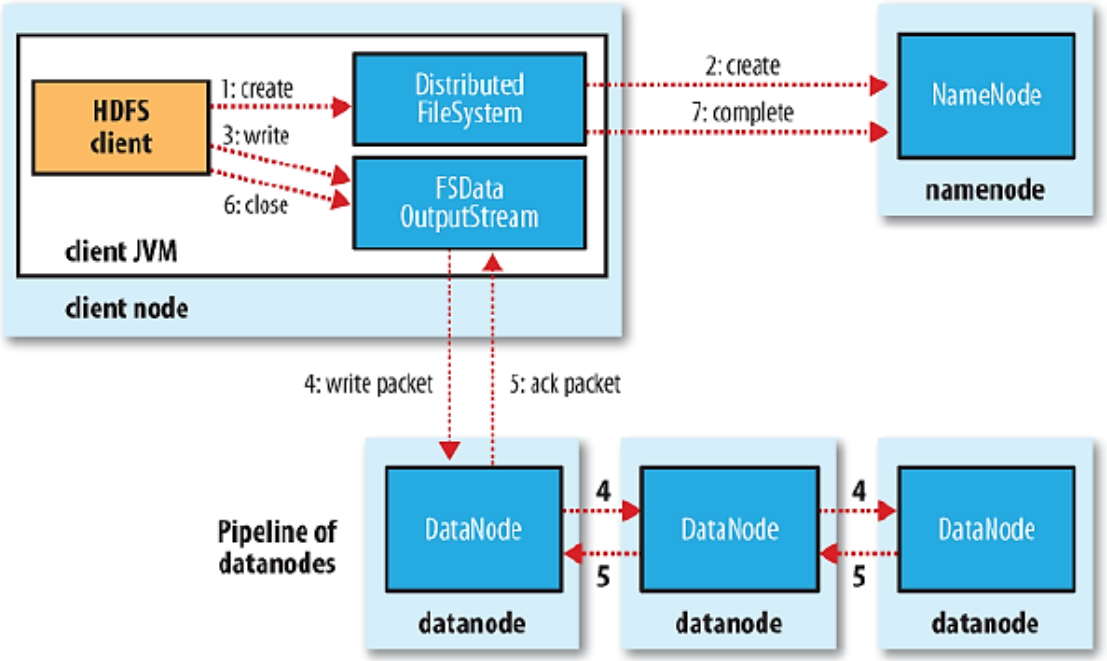
- ✓ **Κλείσιμο HDFSStorageManager: σύγχρονη εγγραφή της κατάστασης του Index και των δεδομένων στο HDFS.**

Η συνάρτηση close() της κλάσης HDFSStorageManager προκαλεί το κλείσιμο της ροής δεδομένων από τα αρχεία του HDFS στην περίπτωση που έχουμε ανακτήσει ένα index, δηλαδή κλείσιμο των FSDataInputStream από τα αρχεία με κατάληξη .idx και .dat . Η εντολή αυτή είναι απαραίτητο να εκτελείται κάθε φορά που μια διεργασία παύει να χρειάζεται πρόσβαση στο αρχείο του index στο HDFS. Σε άλλη περίπτωση, καθώς πολλές διεργασίες είναι πολύ πιθανό στα πλαίσια μιας εργασίας Hadoop να διαβάζουν από το ίδιο αρχείο, μπορεί να καταλήξουμε σε εξαίρεση λόγω πολλών ανοικτών συνδέσεων προς το αρχείο που ξεπερνούν το επιτρεπόμενο όριο.

Τέλος, σε περίπτωση που ο HDFSStorageManager γράφει σε ένα αρχείο του HDFS, θα πρέπει στο τέλος της διεργασίας να κληθεί η συνάρτηση HDFSStorageManager .flush(). Η συνάρτηση αυτή στόχο έχει να γράψει στο αρχείο με κατάληξη .idx όλες τις απαραίτητες πληροφορίες για την μελλοντική ανάκτηση του index. Συγκεκριμένα, αποθηκεύει την τιμή που αντιστοιχεί στο μέγεθος της σελίδας αποθήκευσης (παράμετρος page size) και την τιμή της μεταβλητής m_nextPage (η θέση της επόμενης κενής σελίδας στο αρχείο του HDFS). Τέλος, κλείνει την ροή εξόδου FSDataOutputStream προς τα αρχεία με καταλήξεις .idx και .dat, αφού πρώτα καλέσει την FSDataOutputStream.flush() για να είμαστε σίγουροι ότι όλα τα δεδομένα που έπρεπε να εγγραφούν όντως γράφτηκαν στο HDFS.

Πριν κλείσουμε την ενότητα του HDFSStorageManager, θα αναλύσουμε λίγο παραπάνω την έννοια της παραμέτρου PageSize, που εκφράζει τον αριθμό των bytes που διαβάζονται ή γράφονται στα αρχεία του HDFS κάθε φορά. Μπορεί ο χρήστης να επιλέγει την τιμή της παραμέτρου, υπάρχουν όμως κάποια στοιχεία της αρχιτεκτονικής του HDFS που επηρεάζουν την απόδοση του HDFSStorageManager και συνδέονται με την παράμετρο αυτή.

Ξέρουμε ότι θεμελιώδης ποσότητα του HDFS είναι το HDFS block, που εκφράζει την ελάχιστη ποσότητα δεδομένων διαβάζεται/γράφεται από/προς το HDFS. Παρόλα αυτά, ενώ όντως ο DataNode λαμβάνει ή δίνει δεδομένα ανά blocks (64 MB by default), η μεταφορά προς την διεργασία-πελάτη (ή από εάν έχουμε λειτουργία εγγραφής) γίνεται με ένα ενδιάμεσο επίπεδο buffering [29].



Εικόνα 4.4: Στάδια μιας λειτουργίας εγγραφής στο HDFS.

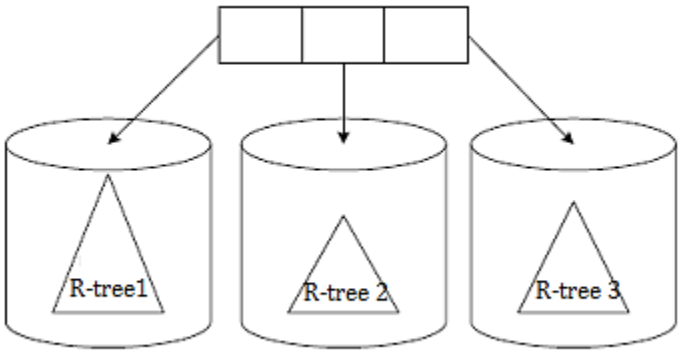
Σε κάθε αίτηση μιας διεργασίας για εγγραφή ή ανάγνωση δεδομένων, ανοίγεται ένα server socket για να γίνει η μεταφορά των δεδομένων ανάμεσα στον DataNode και στην διεργασία που ζητάει την I/O λειτουργία. Κατά τη μεταφορά λοιπόν των δεδομένων μέσω sockets, υπάρχουν ενδιάμεσα buffer που κάνουν την μεταφορά, συνήθως κατά πακέτα των 64KB (αυτό καθορίζεται από την Hadoop παράμετρο dfs.client-write-packet-size αλλά και από τους buffer των Input/Output streams) [29]. Έτσι, επιλέγοντας ως μέγεθος σελίδας PageSize του HDFSStorageManager τιμή ίση με την τιμή dfs.client-write-packet-size (64KB by default), θα έχουμε λιγότερα «άχρηστα» δεδομένα προς μεταφορά, ειδικά κατά τη

διαδικασία εγγραφής του δέντρου. Πειραματικές μετρήσεις στην επιστημονική δημοσίευση «Multi-dimensional Index on Hadoop Distributed File System» [30] που δημοσιεύτηκε όσο αυτή η εργασία ήδη εξελισσόταν, βεβαιώνουν την αποτελεσματικότητα αυτής της επιλογής και στην πράξη.

4.3 Υλοποίηση Κατασκευής Packed Hilbert R-tree σε περιβάλλον Hadoop

4.3.1 Εισαγωγή - Γενική περιγραφή του αλγορίθμου

Υπάρχουν αρκετές εναλλακτικές τεχνικές σχεδίασης ενός συστήματος δεικτοδότησης χωρικών δεδομένων ώστε να μπορεί να επωφεληθεί από κατανεμημένα συστήματα. Για κατασκευή R-tree σε κατανεμημένα συστήματα μια καλή τεχνική είναι η κατασκευή ενός αριθμού από ανεξάρτητα R-trees. Τα χωρικά δεδομένα διαμερίζονται (μοιράζονται) ανάμεσα σε έναν αριθμό διαθέσιμων κόμβων επεξεργασίας και ένα R-tree χτίζεται για κάθε κόμβο. Αυτή την τεχνική εφαρμόσαμε στην παρούσα διπλωματική εργασία, χρησιμοποιώντας το framework Hadoop για την κατασκευή ενός packed Hilbert R-tree.



Εικόνα 4.5: Ανεξάρτητα δέντρα R-trees σε κάθε κόμβο επεξεργασίας.

Το Packed Hilbert R-tree που θα κατασκευάσουμε δεικτοδοτεί δεδομένα που αντιπροσωπεύονται από το MBR τους. Η υλοποίησή μας δέχεται σαν είσοδο ένα αρχείο κειμένου με ορθογώνια (ορισμένα από 2 σημεία της διαγωνίου τους) και μοναδικά id που αντιστοιχούν σε κάθε ορθογώνιο. Κατά τη φάση Map, για κάθε ορθογώνιο που διαβάζουμε στην είσοδο, υπολογίζεται η τιμή Hilbert του κέντρου του και την συνέχεια δημιουργείται

ένα ζεύγος της μορφής <τιμή Hilbert, χωρικό αντικείμενο>, όπου η τιμή Hilbert είναι το κλειδί και το χωρικό αντικείμενο (ορθογώνιο και id) είναι η τιμή του ζεύγους.

Όπως προαναφέραμε στο κεφάλαιο 2, ο τρόπος που τα χωρικά δεδομένα θα διαμοιραστούν ώστε να κατασκευαστούν καλής ποιότητας R-trees σε κάθε κόμβο του cluster, είναι παράγοντας που επηρεάζει πολύ την απόδοση του αλγορίθμου.

Όσο αφορά την κατανομή των χωρικών δεδομένων για την κατασκευή των R-trees:

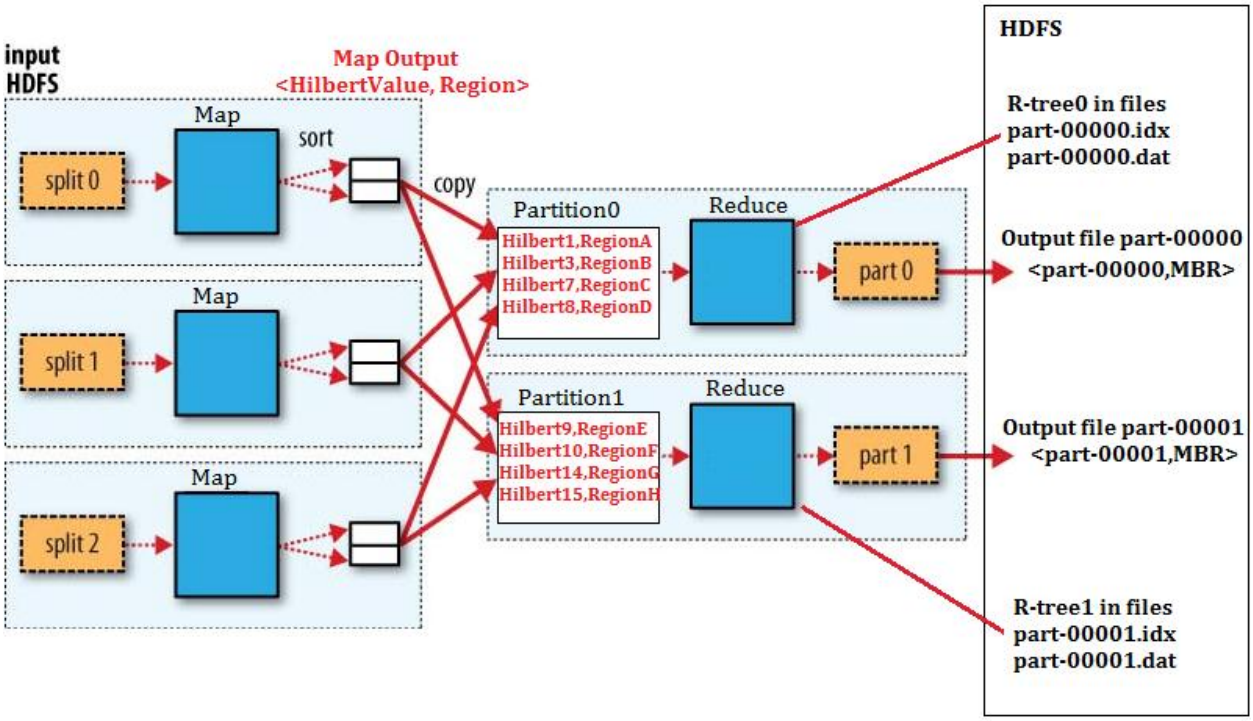
- ✓ Κατανομή με βάση τα δεδομένα (Data Distribution):
  Τα χωρικά δεδομένα διαμοιράζονται στους κόμβους επεξεργασίας με κριτήριο την ανάθεση ίσου περίπου αριθμού δεδομένων σε κάθε κόμβο, συνήθως χρησιμοποιώντας συναρτήσεις κατακερματισμού ή τεχνικές Round Robin. Η μέθοδος αυτή εξασφαλίζει ισορροπημένη κατανομή των δεδομένων, αλλά καθώς δεν υπάρχει κάποιο χωρικό κριτήριο, η εκτέλεση επερωτημάτων θα εξετάζει συνήθως όλους τους κόμβους.

- ✓ Κατανομή με βάση το χώρο (Space Distribution):
  Ο k-διάστατος χώρος χωρίζεται σε τόσες διαμερίσεις όσοι και οι διαθέσιμοι κόμβοι επεξεργασίας. Τα χωρικά δεδομένα διαμοιράζονται στους κόμβους επεξεργασίας με κριτήριο την τιμή τους στον k-διάστατο χώρο. Στην περίπτωση μη ομοιόμορφων δεδομένων ωστόσο, αυτή η μέθοδος θα δημιουργήσει άνισες διαμερίσεις, με συνέπεια ο φόρτος της εκτέλεσης επερωτημάτων να μην μπορεί να μοιραστεί ισορροπημένα ανάμεσα στους κόμβους επεξεργασίας.

Στην υλοποίησή μας εφαρμόσαμε μια τεχνική που συνδυάζει τα πλεονεκτήματα των 2 παραπάνω μεθόδων. Εάν R είναι ο αριθμός των διεργασιών Reduce (Reducers), διαχωρίζουμε το εύρος τιμών που παίρνει η τιμή Hilbert των δεδομένων μας σε R μέρη, ώστε σε κάθε διάστημα να υπάρχουν περίπου ίσα ζευγάρια με τιμή Hilbert σ' αυτό το διάστημα. Οι τιμές-όρια των R διαστημάτων έχουν προκύψει από δειγματοληψία της εισόδου που κάναμε πριν ξεκινήσει η εργασία Hadoop για την κατασκευή του δέντρου. Τα ζευγάρια εξόδου της φάσης Map διαμερίζονται λοιπόν με βάση το κλειδί τους (δηλαδή την τιμή Hilbert), ανάλογα με το εύρος τιμών/διάστημα στο οποίο ανήκουν (διαδικασία partitioning). Καθώς η τιμή Hilbert διατηρεί την τοπικότητα, και δεδομένου ότι η έξοδος της φάσης Map είναι ταξινομημένη ως προς την τιμή κλειδιού κάθε ζεύγους (δηλαδή την τιμή Hilbert), η διαδικασία partitioning θα μας δώσει R ίσες περίπου στο μέγεθος διαμερίσεις, με κοντινά στο χώρο σημεία η καθεμία, ταξινομημένα κατά αύξουσα τιμή κλειδιού, δηλαδή κατά αύξουσα τιμή Hilbert.

Ο κάθε Reducer θα αναλάβει μια από τις R διαμερίσεις, και θα κατασκευάσει ένα Packed Hilbert R-tree το οποίο θα αποθηκεύσει στο HDFS. Σε κάθε διεργασία Reduce, καλείται μόνο μια φορά η συνάρτηση reduce(), με είσοδο όλα τα ζευγάρια της διαμέρισης ώστε να χτιστεί το δέντρο χρησιμοποιώντας τα χωρικά αντικείμενα που αντιστοιχούν στις τιμές των ζευγαριών εισόδου. Τα επιμέρους δέντρα χτίζονται υλοποιώντας τον αλγόριθμο

packed Hilbert R-tree και χρησιμοποιώντας την βιβλιοθήκη spatial Index και τον HDFSStorageManager για αποθήκευση σε αρχεία του HDFS. Αποτέλεσμα είναι να έχουμε για κάθε Reducer 2 αρχεία (ένα με κατάληξη .idx και ένα με κατάληξη .dat) με μοναδικό όνομα που αντιστοιχεί σ' αυτόν τον Reducer και που έχουν αποθηκευμένο το R-tree. Η έξοδος του Reducer, τέλος, είναι ζευγάρια της μορφής <όνομα αρχείου με Rtree, MBR>. Με αυτόν τον τρόπο ο reducer αποθηκεύει σε ένα αρχείο εξόδου πληροφορία σχετικά με το δέντρο που παρήγαγε: αποθηκεύει αρχικά ένα ζεύγος <όνομα αρχείου με Rtree, MBR της ρίζας του δέντρου> και στη συνέχεια για κάθε κόμβο-παιδί της ρίζας ένα ζεύγος <όνομα αρχείου με Rtree, MBR(i) > όπου MBR(i) το MBR του i-οστού κόμβου παιδιού της ρίζας. Όταν η εργασία Hadoop ολοκληρώνεται, πριν τερματιστεί η εφαρμογή, κατασκευάζουμε από τα αρχεία εξόδου ένα επιπλέον αρχείο στο HDFS, στο οποίο γράφουμε συγκεντρωτικά ζεύγη της μορφής < όνομα αρχείου με Rtree(i), MBR της ρίζας του δέντρου(i) >, για κάθε R-tree (i) που χτίσαμε. Η πληροφορία αυτή είναι απαραίτητη για την εκτέλεση επερωτημάτων με τη βοήθεια του δέντρου αργότερα.



Εικόνα 4.6: Εποπτικό σχεδιάγραμμα των φάσεων εκτέλεσης της εργασίας Hadoop.

Η υλοποίηση έγινε χρησιμοποιώντας την έκδοση Hadoop 0.20.2, ενώ στην πορεία ο κώδικας ελέγχτηκε και χρησιμοποιώντας την έκδοση Hadoop 0.20.203 (stable). Ο κώδικας υιοθετεί το νέο API του Hadoop, το οποίο επιβάλλει αλλαγές σε αρκετές κλάσεις σε σχέση με το παλιό API. Καθώς η έκδοση Hadoop 0.20.2 δεν υποστήριζε εξ' ολοκλήρου το νέο API,

δηλαδή δεν παρείχε τις καινούριες εκδόσεις όλων των συναρτήσεων, αναγκαστήκαμε σε κάποιες περιπτώσεις να εισάγουμε τις καινούριες εκδόσεις συναρτήσεων στο project μας ως κοινές συναρτήσεις και να τις χρησιμοποιούμε κάνοντάς τις import από εκεί. Ευτυχώς, το Hadoop σαν open-source πρόγραμμα παρέχει μεγάλη ευελιξία στον χρήστη, που μπορεί στο πρόγραμμά του να χρησιμοποιεί δικές του custom κλάσεις στην θέση βασικών κλάσεων που παρέχονται υλοποιημένες από το Hadoop. Φυσικά, η ευθύνη για την τελική σωστή λειτουργία της υλοποίησης ανήκει στον προγραμματιστή!

Τέλος, μιλήσαμε ήδη στο κεφάλαιο 3 για την αξιοπιστία του Hadoop και την ικανότητά του να ανιχνεύει αποτυχημένες ή καθυστερημένες διεργασίες και να τις επαναδρομολογεί σε διαφορετικό κόμβο του cluster. Για την τελευταία περίπτωση, το φαινόμενο αυτό της επαναδρομολόγησης μιας διεργασίας-αντίγραφου όταν ένα task αργεί πολύ, ονομάζεται *Speculative execution.* Στην υλοποίηση της κατασκευής του R-tree, προβλέψαμε απενεργοποίηση της Speculative execution στους Reducers. Αυτό έγινε για να μην υπάρχουν εμπλοκές κατά την δημιουργία ενός αρχείου R-tree (όταν π.χ. μια διεργασία και το αντίγραφό της προσπαθήσουν να γράψουν το ίδιο αρχείο R-tree) και προκληθεί διακοπή της εγγραφής αυτού του κομματιού του Index.

4.3.2 Ανάλυση της υλοποίησης

Στην ενότητα αυτή θα αναλύσουμε περισσότερο κάθε βήμα της υλοποίησής μας.

1. **Δειγματοληψία και διάβασμα Εισόδου**

   Η διαδικασία της δειγματοληψίας γίνεται πριν την έναρξη της κύριας εργασίας Hadoop, χρησιμοποιώντας την κλάση InputSampler που παρέχεται από το Hadoop. Να σημειωθεί, ότι καθώς το Hadoop 0.20.2 δεν παρείχε έκδοση της κλάσης αυτής που να χρησιμοποιεί το νέο API, ενσωματώσαμε τον κώδικα της κλάσης με το νέο API (από την έκδοση Hadoop 0.21) στο project μας στο πακέτο new_Partitioner, και την χρησιμοποιήσαμε από εκεί.

   Η κλάση new_Partitioner.InputSampler παίρνει δείγματα από τα δεδομένα εισόδου με τη μορφή ζευγαριών <κλειδί, τιμή>, για να διαχωρίσει το εύρος τιμών που παίρνει το κλειδί των δεδομένων μας σε R μέρη (με R=αριθμός διαμερίσεων=αριθμός Reducers) ώστε σε κάθε εύρος τιμών να υπάρχουν περίπου ίσα ζευγάρια με τιμή κλειδιού σ' αυτό το εύρος. Ο χρήστης καθορίζει τον αριθμό των δειγμάτων που θα ληφθούν, τον μέγιστο αριθμό από input splits που θα εξετάσει ο sampler, αλλά κυρίως το InputFormat με το οποίο ο sampler θα διαβάσει τα ζευγάρια <κλειδί, τιμή> από τα αρχεία εισόδου. Η αρχικοποίηση του sampler στον κώδικά μας γίνεται ως εξής:

job.setInputFormatClass(MySamplerInputFormat.class);

InputSampler.Sampler <VLongWritable, Text> sampler = new InputSampler.RandomSampler <VLongWritable, Text> (0.1, Integer.parseInt(args[4]), 20);

Ο sampler θα επιλέξει με τυχαίο τρόπο (κάθε record έχει πιθανότητα 0.1% να επιλεγεί) τον αριθμό δειγμάτων που θα ορίσει ο χρήστης ως παράμετρο, χρησιμοποιώντας το πολύ 20 input splits από την είσοδο τα οποία διαβάζει με την βοήθεια της κλάσης MySamplerInputFormat.

Ο sampler θα παράξει ένα αρχείο με τις τιμές των ορίων των R διαμερίσεων, που θα διαβαστούν στην συνέχεια μετά την φάση Map από τον partitioner. Κάθε κλειδί ζευγαριού στην έξοδο των Mappers θα συγκρίνεται με τα όρια αυτών των διαμερίσεων για να βρεθεί σε ποιο εύρος ανήκει και να οδηγηθεί στο ανάλογο partition. Για να λειτουργήσει λοιπόν αυτό το σύστημα, το κλειδί από τα ζευγάρια <κλειδί_εισόδου,τιμή_εισόδου> που διαβάζει ο sampler από την είσοδο, θα πρέπει να είναι ίδιας μορφής με το κλειδί από τα ζευγάρια <κλειδί_map, τιμή_map> που βγάζει σαν έξοδο ο mapper. Για το λόγο αυτό ορίσαμε ένα custom InputFormat, την κλάση με όνομα MySamplerInputFormat που θα δημιουργεί από τα αρχεία εισόδου τα ζευγάρια <κλειδί, τιμή> που θα διαβάζει ο sampler. Το κλειδί θα πρέπει να είναι το ίδιο με το κλειδί που θα βγάζει ο Mapper, δηλαδή να ισούται με την τιμή Hilbert του χωρικού αντικειμένου που διαβάζουμε κάθε φορά από την είσοδο.

Η είσοδος είναι αρχεία text που σε κάθε γραμμή τους περιέχουν ένα id και ένα χωρικό αντικείμενο που αντιστοιχεί στο id αυτό. Ο MySamplerInputFormat διαβάζει το αρχείο, το διασπά σε input splits και στη συνέχεια παράγει records-ζευγάρια της μορφής:

<κλειδί=τιμή Hilbert του χωρικού αντικειμένου, τιμή=αδιάφορη>

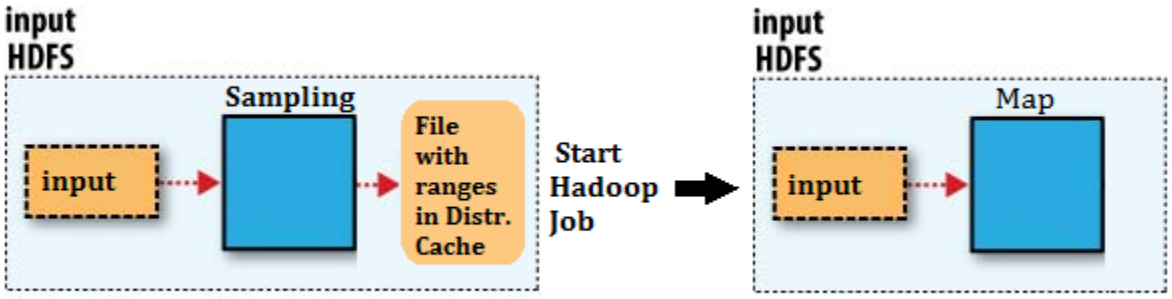
Ο sampler δειγματοληπτεί τα ζεύγη και παράγει το αρχείο με όνομα _partitions.lst που περιέχει τα όρια των (περίπου ίσων) R διαμερίσεων.

Στη συνέχεια, το αρχείο αυτό αντιγράφεται τοπικά σε όλους τους κόμβους του cluster, ώστε να είναι διαθέσιμο κατά την φάση partitioning. Αυτό γίνεται χρησιμοποιώντας τον μηχανισμό Distributed Cache που μας παρέχει το Hadoop. Ο μηχανισμός αυτός μας επιτρέπει να αντιγράψουμε μικρά αρχεία (λίγων Kilobytes ή MegaBytes) από το HDFS στο τοπικό σύστημα αρχείων κάθε κόμβου του cluster, ώστε να είναι άμεσα διαθέσιμα για διάβασμα με χαμηλό κόστος. Η τεχνική αυτή αυξάνει την απόδοση, καθώς οι διεργασίες που χρειάζονται read-only πρόσβαση σε κοινά δεδομένα μπορούν να τα διαβάσουν από τον τοπικά κόμβο, χωρίς να πρέπει να ανακτήσουν απομακρυσμένα blocks από το HDFS.

Μετά την διαδικασία δειγματοληψίας, αρχίζει η εκτέλεση της κύριας εργασίας Hadoop. Τα δεδομένα διαβάζονται από τα αρχεία εισόδου ώστε να προετοιμαστούν

τα ζευγάρια <κλειδί, τιμή> που θα αποτελέσουν είσοδο στις διεργασίες Map. Επιλέξαμε όπως αναφέραμε ήδη να θεωρήσουμε ως είσοδο της εργασίας απλά αρχεία κειμένου, ώστε να τηρηθεί η γενικότητα και να μην υπάρχει προαπαιτούμενη προεπεξεργασία. Έτσι, η είσοδος διαβάζεται χρησιμοποιώντας την κλάση TextInputFormat που δημιουργεί ζευγάρια από κάθε γραμμή του κειμένου εισόδου. Η μορφή των ζευγαριών που θα τροφοδοτηθούν στην φάση Map είναι η ακόλουθη:

<κλειδί = θέση γραμμής που διαβάσαμε, τιμή = ολόκληρη η γραμμή κειμένου>



Εικόνα 4.7: Η φάση Sampling πριν την φάση Map

2. Φάση Map

Η φάση Map λαμβάνει στην είσοδό της input splits, και δρομολογεί μια διεργασία Map για το καθένα για να τα επεξεργαστεί. Για κάθε record μέσα σε ένα input split, η διεργασία καλεί μια φορά τη συνάρτηση Map, με είσοδο το ζεύγος <τιμή, κλειδί> του record αυτού. Η συνάρτηση Map στην υλοποίησή μας, δέχεται είσοδο <κλειδί, τιμή> της μορφής

<VLongWritable:θέση γραμμής που διαβάσαμε, Text:ολόκληρη η γραμμή κειμένου>

Με την τιμή συγκεκριμένα να έχει την γενική μορφή:
id_αντικειμένου<\t>συντεταγμένες 2 σημείων του ορθογωνίου χωρισμένες με \t.

Η τιμή του κλειδιού ουσιαστικά μας είναι αδιάφορη.

Η συνάρτηση map διαβάζει την τιμή του ζεύγους και από αυτήν εξάγει το id του χωρικού αντικειμένου και τις συντεταγμένες των 2 σημείων του. Από αυτά κατασκευάζει ένα στιγμιότυπο new_region της κλάσης RegionWritable που αναπαριστά το χωρικό αντικείμενο.

Η custom writable κλάση RegionWritable περιέχει ένα πεδίο id τύπου VLongWritable που αποθηκεύει το id του χωρικού αντικειμένου και μια δομή Region τύπου TwoDArrayWritable που λειτουργεί σαν δισδιάστατο διάνυσμα που αποθηκεύει τις συντεταγμένες του ορθογωνίου. Η συνάρτηση map, μέσω της συνάρτησης RegionWritable.getHilbertValue() βρίσκει την τιμή Hilbert του κέντρου του χωρικού αντικειμένου και την αποθηκεύει στην μεταβλητή hilbertValue (τύπου VLongWritable). Ο αλγόριθμος που περιγράφει τον υπολογισμό της τιμής Hilbert για ένα αντικείμενο φαίνεται στο πλαίσιο που ακολουθεί, όπως αναφέρεται στην επιστημονική δημοσίευση του H. V. Jagadish "Linear Clustering of Objects with Multiple Attributes".[31]

Τμήμα του αλγορίθμου για την εξαγωγή της τιμής Hilbert: Η τιμές x και y αντιστοιχούν στις συντεταγμένες του κέντρου του ορθογωνίου του οποίου την τιμή Hilbert αναζητούμε. Η τιμή size είναι μια σταθερά με πολύ μεγάλη τιμή, ώστε να προσομοιώσουμε το fractal. Στην περίπτωσή μας, θεωρούμε size= 33554432.

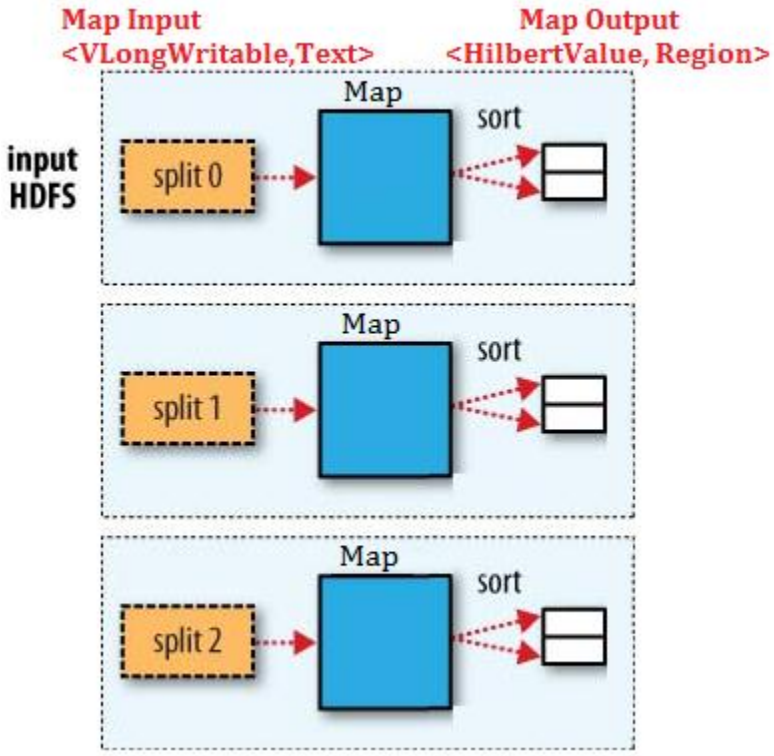
```
int rotation_table[4] = (3, 0, 0, 1) ;
int sense_table[4] = (-1, 1, 1, -1) ;
in quad_table[4][2][2]= {  {{0,1},{3,2}} , {{1,2},{0,3}} ,
{{2,3},{1,0}} , {{3,0},{2,1}}  } ;
rotation = 0  /* Initially no rotation */
sense = 1    /* Initially positive sense */
num=0

for(k=side/2 ,k>0, k=k/2)   {
        xbit = x/k       /*Get the msb of x*/
        ybit = y/k
        x -= k*xbit  /* Take away the current msb */
        y -= k*ybit
        quad = quad_table[rotation][xbit][ybit]
         /*Which quadrant am I in? */
        num += (sense == - 1 ) ? k*k*(3-quad) k*k*quad

        rotation += rotation_table[quad]
        /* Fix rotation value for next time */
        if (rotation >= 4) rotation -= 4
        /* Addition is modulo 4 */
        sense *= sense_table[quad] ,
        /* Fix sense value for next time */
}
```

Πίνακας 4.5: Αλγόριθμος για υπολογισμό της τιμής Hilbert ενός ορθογωνίου.

Ο παραπάνω αλγόριθμος υπολογίζει την τιμή Hilbert θεωρώντας ακέραιες τιμές συντεταγμένων του κέντρου του ορθογωνίου (x, y). Στην περίπτωσή μας, προκειμένου να υποστηρίξουμε τιμές double στις συντεταγμένες μας (όπως άλλωστε ήταν και οι συντεταγμένες του dataset που χρησιμοποιήσαμε για τα πειράματά μας αργότερα), απλά πολλαπλασιάσαμε κάθε συντεταγμένη με το 10 και πήραμε την ακέραια τιμή. Η παρέμβαση αυτή δεν επηρεάζει την τοπικότητα και δίνει και πάλι πολύ καλά αποτελέσματα.

Στο τέλος, αφού έχει υπολογιστεί η τιμή Hilbert του χωρικού αντικειμένου, η διεργασία Map δίνει έξοδο <κλειδί, τιμή> της μορφής

<VLongWritable: hilbertValue, RegionWritable: το χωρικό αντικείμενο new_region>



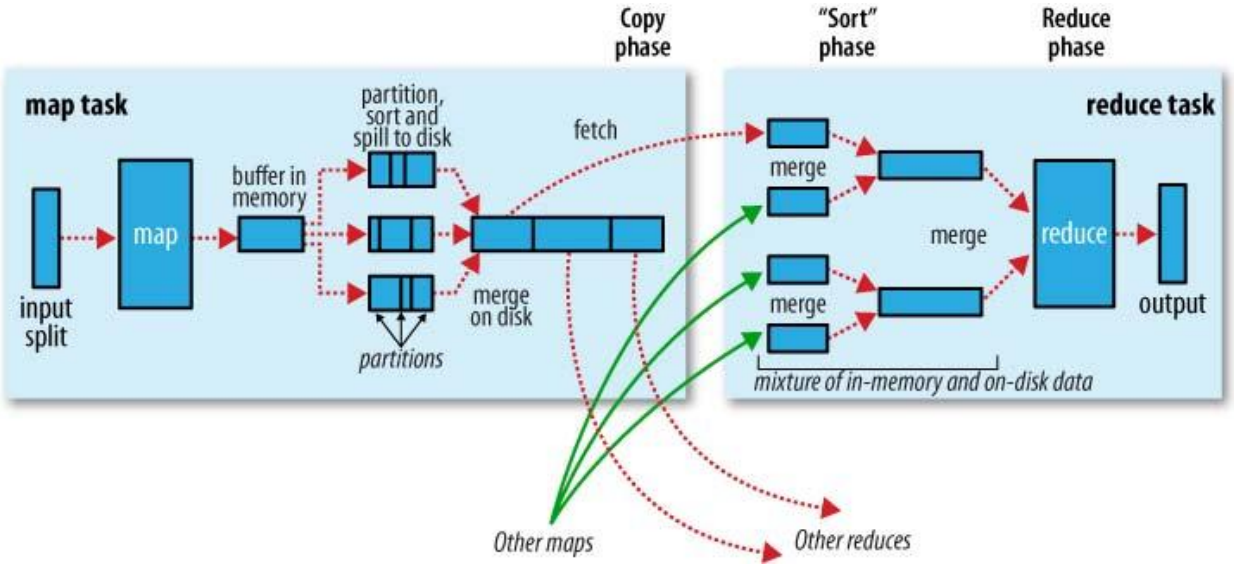
Εικόνα 4.8: Η φάση Map.

3. Φάση Partitioning

Όπως αναφέραμε παραπάνω, ακολουθήσαμε την εξής μέθοδο για την διαδικασία partitioning της εξόδου των Mappers που θα λάβει χώρα αργότερα: εάν R είναι ο αριθμός των διεργασιών Reduce (Reducers), διαχωρίζουμε το εύρος τιμών που

παίρνει η τιμή Hilbert των δεδομένων μας σε R μέρη, ώστε σε κάθε εύρος τιμών να υπάρχουν περίπου ίσα ζευγάρια με τιμή Hilbert σ' αυτό το εύρος. Οι τιμές-όρια των R διαστημάτων έχουν προκύψει από δειγματοληψία της εισόδου που κάναμε πριν ξεκινήσει η εργασία Hadoop για την κατασκευή του δέντρου. Η μέθοδος αυτή είναι βασισμένη στην Hadoop εφαρμογή TeraSort [32], η οποία χρησιμοποίησε δειγματοληψία στην είσοδο και διαμέριση με βάση το εύρος τιμών, ώστε να πετύχει μια υλοποίηση ταξινόμησης 1 TeraByte εισόδου σε μόλις 209 seconds!

Η κλάση new_Partitioner.TotalOrderPartitioner δέχεται ως είσοδο τα ζευγάρια που έχουν παραχθεί από κάθε Mapper και επιστρέφει για κάθε ζευγάρι έναν ακέραιο αριθμό από 0 έως R-1, που αντιστοιχεί στον αριθμό της διαμέρισης που θα οδηγηθεί το ζευγάρι αυτό. Ο διαμερισμός γίνεται ως εξής: ανακτώνται από το αρχείο όνομα _partitions.lst τα όρια των (περίπου ίσων) R διαμερίσεων, και ανάλογα σε ποιο εύρος ανήκει η τιμή κλειδιού του κάθε ζεύγους <VLongWritable:hilbertValue, RegionWritable: το χωρικό αντικείμενο new_region>, το ζεύγος οδηγείται στο αντίστοιχο partition.



Εικόνα 4.9: Λεπτομερέστερη απεικόνιση της φάσης Partition και Shuffle.

Η τιμή Hilbert διατηρεί την τοπικότητα και η έξοδος της φάσης Map είναι ταξινομημένη ως προς την τιμή κλειδιού κάθε ζεύγους. Άρα, η διαδικασία partitioning και η διαδικασία Shuffle (μεταφορά της διαμέρισης στον κόμβο που θα εκτελεστεί η διεργασία reduce και συνολική ταξινόμησή της) θα μας δώσουν R ίσες

περίπου στο μέγεθος διαμερίσεις, μια σε κάθε Reducer, με κοντινά στο χώρο σημεία η καθεμία, ταξινομημένα κατά αύξουσα τιμή Hilbert.

4. Φάση Reduce και εγγραφή αποτελεσμάτων

Αυτό που επιθυμούμε είναι στην φάση Reduce, κάθε Reducer να κατασκευάσει ένα δέντρο και να το αποθηκεύσει στο HDFS. Ωστόσο, κάθε Reducer αναλαμβάνει ένα partition, όπου μπορεί να περιέχει ζευγάρια με πολλές διαφορετικές τιμές κλειδιών (Hilbert values). Για κάθε διαφορετική τιμή κλειδιού, η κλάση Grouping Comparator στο Hadoop ομαδοποιεί τα ζευγάρια της διαμέρισης που έχουν την ίδια τιμή κλειδιού. Έτσι δημιουργούνται ζευγάρια της μορφής <κλειδί, λίστα τιμών>. Για κάθε τέτοιο ζεύγος δρομολογείται και μια νέα κλήση της συνάρτησης reduce(). Προφανώς, αυτό είναι κάτι που δεν θέλουμε να γίνει, καθώς κάθε Reducer θα προσπαθούσε να φτιάξει ένα δέντρο για κάθε διαφορετική τιμή κλειδιού! Επομένως, πρέπει να ορίσουμε την δικιά μας κλάση Grouping Comparator, που θα ομαδοποιεί όλα τα ζευγάρια της διαμέρισης σε ένα μόνο ζεύγος <κλειδί, λίστα τιμών>.

Ορίζουμε λοιπόν σαν κλάση Grouping Comparator την δική μας static κλάση AlwaysEqualGroupingComparator, η οποία για κάθε δύο κλειδιά που συγκρίνει, τα βρίσκει πάντα όμοια! Με αυτό τον τρόπο, δημιουργείται ένα ζεύγος <κλειδί, τιμή> της μορφής

<VLongWritable: hilbertValue, λίστα από(RegionWritable: τα χωρικά αντικείμενα από όλη την διαμέριση)>

Το ζεύγος αυτό είναι είσοδος της συνάρτησης reduce() η οποία θα κληθεί μια μόνο φορά σε κάθε Reducer.

Η συνάρτηση reduce() αρχικοποιεί ένα δέντρο χρησιμοποιώντας τις κλάσεις της βιβλιοθήκης spatial index, όπως περιγράψαμε στα κεφάλαια 4.2.4 «Πρόσθετη Λειτουργικότητα: Υλοποίηση Packed Hilbert R-tree στο δίσκο» και 4.2.4 Πρόσθετη Λειτουργικότητα: Υλοποίηση Storage Manager για αποθήκευση στο HDFS.

Πρώτα ρυθμίζονται οι παράμετροι και γίνεται η αρχικοποίηση του storage manager. Χρησιμοποιούμε ένα στιγμιότυπο του Buffer για ενδιάμεση αποθήκευση και τον HDFSStorageManager για αποθήκευση στο HDFS. Στον Πίνακα 4.6 φαίνονται οι παράμετροι που χρησιμοποιήσαμε για την αρχικοποίηση του storage manager.

Όνομα Παραμέτρου	Τύπος	Περιγραφή
FileName	String	Η βάση για το όνομα του αρχείου που θα αποθηκευτεί το Index (χωρίς καταλήξεις .index ή .dat). Έχει τη μορφή part-r-XXXXX όπου XXXXX ο αριθμός id του Reducer, π.χ. part-r-00000.
Overwrite	Boolean	True.
PageSize	Integer	Το μέγεθος σελίδας αποθήκευσης. Δίνουμε, όπως είπαμε, τιμή ίση με την παράμετρο dfs.client-write-packet-size του Hadoop, με default τιμή 64 Kilobytes. Αυτό γίνεται με σκοπό να ελαχιστοποιήσουμε την μεταφορά άχρηστων bytes στο δίκτυο του cluster.

Πίνακας 4.4: Παράμετροι που χρησιμοποιήσαμε για την αρχικοποίηση του storage manager.

Στον Πίνακα 4.7 φαίνονται οι παράμετροι που χρησιμοποιήσαμε για την αρχικοποίηση του Packed Hilbert R-tree.

Όνομα Παραμέτρου	Τύπος	Περιγραφή & προεπιλεγμένες τιμές
IndexIndentifier	Integer	Null (δεν ορίζεται στην αρχή, μετά την κατασκευή του δέντρου η τιμή αυτή θα έχει το id του νέου R-tree).
Dimension	Integer	Δίνουμε τιμή 2. Μπορεί να πάρει και τιμές για περισσότερες διαστάσεις.
IndexCapacity	Integer	Η συνολική χωρητικότητα των εσωτερικών κόμβων του δέντρου δεν δίνεται από τον χρήστη, αλλά καθορίζεται από το μέγεθος που έχει η παράμετρος PageSize του HDFSStorageManager, καθώς θέλουμε ένας κόμβος να καταλαμβάνει

		ακριβώς μια σελίδα. Υπολογίζεται βάσει τύπου. Για PageSize= dfs.client-write-packet-size=64 KiloBytes, έχουμε IndexCapacity=1489.
LeafCapacity	Integer	Ίση με την τιμή της IndexCapacity.
TreeVariant	Integer	Ίση με 4 που αντιστοιχεί στον τύπο Packed Hilbert R-tree.
writtenOnHDFS	Boolean	True.

Πίνακας 4.7:Παράμετροι που χρησιμοποιήσαμε για την αρχικοποίηση του R-tree.

Στη συνέχεια προσπελαύνονται τα χωρικά δεδομένα που βρίσκονται στην λίστα τιμών της εισόδου της reduce(), και φορτώνονται στο δέντρο. Ο αλγόριθμος είναι παρόμοιος με αυτόν που υλοποιήσαμε για κατασκευή packed Hilbert R-tree στην βιβλιοθήκη spatial index, με μικρές διαφορές λόγω ιδιαιτερότητας του περιβάλλοντος Hadoop. Τα βήματα που περιγράφουν την λογική φαίνονται παρακάτω.

Αλγόριθμος *BulkLoadData*: packing χωρικών δεδομένων σε ένα Rtree σε περιβάλλον Hadoop.

Βήμα 1. /* Σχηματισμός των κόμβων-φύλλων (επίπεδο δέντρου L=0) */

Κατασκεύασε ένα κενό Vector με όνομα CurrentLevel_Nodes.

While (υπάρχουν ακόμα χωρικά αντικείμενα RegionWritable στην είσοδο) {

Ομαδοποίησε τα επόμενα <LeafCapacity> στο πλήθος αντικείμενα RegionWritable (ή λιγότερα εάν έχουμε φτάσει στο τέλος της λίστας της εισόδου) .

Φτιάξε για κάθε αντικείμενο RegionWritable ένα στιγμιότυπο της κλάσης spatialindex.Region που να εκφράζει το ορθογώνιο, και φόρτωσέ το σε ένα νέο Vector< Region > n_shapes. Τα ids φόρτωσέ τα σε ένα νέο Vector<Long> ids.

Κατασκεύασε έναν νέο κόμβο-φύλλο newleaf.

Κάλεσε τη συνάρτηση newleaf.bulkLoadMakeLeaf(null,n_shapes,n_ids) για να αναθέσει τα ορθογώνια με τα αντίστοιχα ids σε αυτόν τον κόμβο-φύλλο.

Πρόσθεσε στο Vector με όνομα CurrentLevel_Nodes ένα στιγμιότυπο της κλάσης EntrySummury, που θα αποθηκεύει το id και το MBR του κόμβου-φύλλου που δημιουργήθηκε.

}

Βήμα 2. /* Σχηματισμός των εσωτερικών κόμβων (επίπεδο δέντρου L+1) */

While (υπάρχουν πάνω από 1 κόμβος στο προηγούμενο επίπεδο L) {

Αντέγραψε τα περιεχόμενα του CurrentLevel_Nodes σε ένα νέο κενό Vector PreviousLevel_Nodes.

Άδειασε τα περιεχόμενα του Vector CurrentLevel_Nodes.

While(υπάρχουν ακόμα ορθογώνια στο Vector PreviousLevel_Nodes){ //μπορώ να φτιάξω έναν ακόμα κόμβο που να τα συμπεριλαμβάνει

Ομαδοποίησε τα επόμενα IndexCapacity στοιχεία του PreviousLevel_Nodes (ή λιγότερα εάν έχουμε φτάσει στο τέλος του vector) και αποθήκευσέ τα σε νέο Vector<EntrySummury> n_ids.

Κατασκεύασε έναν νέο κόμβο newnode.

Κάλεσε τη συνάρτηση newnode.bulkLoadMakeNode(n_ids) για να αναθέσει τα MBRSs με τα αντίστοιχα ids σε αυτόν τον κόμβο.

Πρόσθεσε στο Vector με όνομα CurrentLevel_Nodes. ένα στιγμιότυπο της κλάσης EntrySummury, που θα αποθηκεύει το id και το MBR του κόμβου-φύλλου που δημιουργήθηκε.

}

Προχώρα στο επόμενο επίπεδο L+1 επαναλαμβάνοντας το Βήμα 2.

}

Βήμα 3. Εάν υπάρχει μόνο ένας κόμβος στο προηγούμενο επίπεδο L {

Κάνε αυτόν τον κόμβο ρίζα του δέντρου. }

Πίνακας 4.8: Packing χωρικών δεδομένων σε ένα Rtree σε περιβάλλον Hadoop.

Μετά το τέλος της φόρτωσης των αντικειμένων στο R-tree, καλείται η συνάρτηση RTree.flush() για να αποθηκεύσει τις πληροφορίες της δομής του R-tree και να κλείσει τις ανοιχτές ροές προς τα αρχεία του δέντρου στο HDFS.

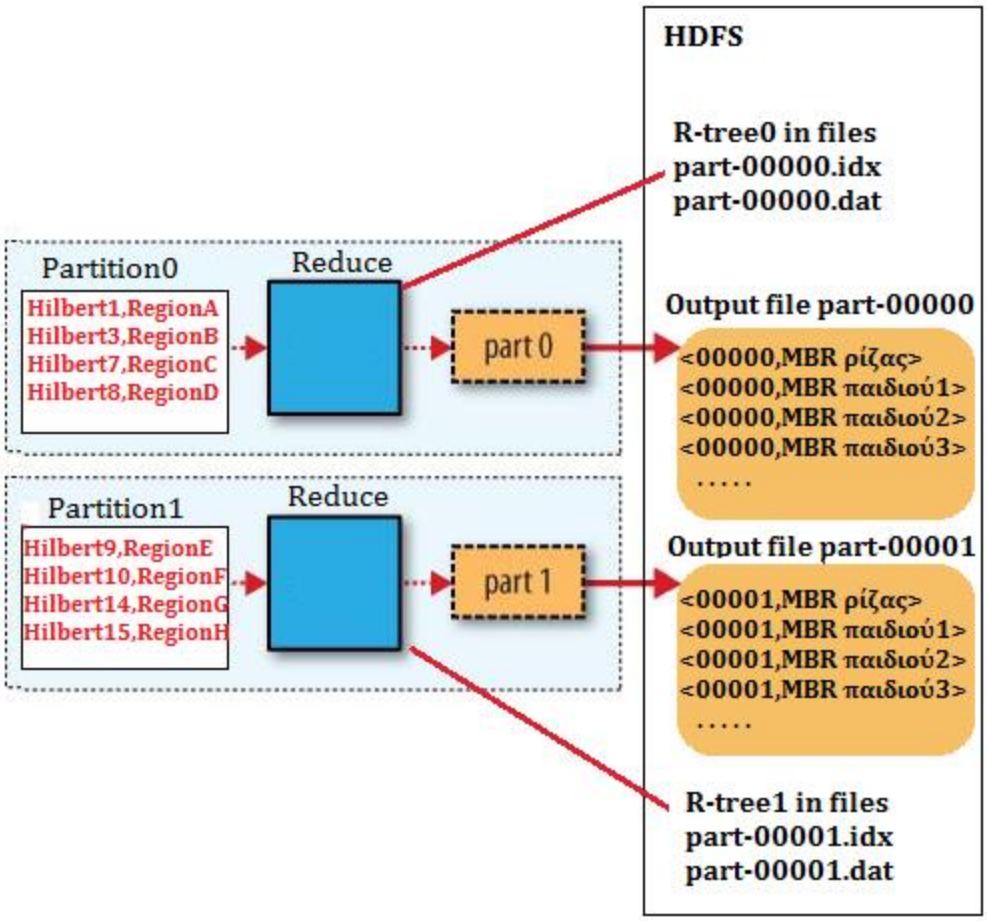
Αφού φτιαχτεί το δέντρο στα αρχεία με όνομα που αντιστοιχεί στον Reducer και κατάληξη .idx και .dat, ο Reducer γράφει στο αρχείο εξόδου του πληροφορίες σχετικά με το δέντρου που έφτιαξε. Συγκεκριμένα, ο Reducer πρέπει να αποθηκεύσει το id του δέντρου ώστε να μπορεί να το ανακτήσει, και πληροφορίες σχετικά με το MBR της ρίζας του δέντρου (και τα MBR των παιδιών της) που θα βοηθήσουν την εκτέλεση επερωτημάτων αργότερα με χρήση του R-tree που παράχθηκε.

Για να αποθηκεύσει τις πληροφορίες αυτές η reduce() έχει έξοδο ζευγάρια της <κλειδί, τιμή> μορφής

<VLongWritable: Id του Reducer, RegionWritable: πληροφορίες για IndexId και MBRs ρίζας>

Η τιμή του κλειδιού μας είναι ουσιαστικά αδιάφορη.

Συγκεκριμένα, η reduce() γράφει την έξοδο σε αρχείο τύπου SequenceFile, ώστε να μπορεί να γράψει τα δεδομένα σε δυαδική μορφή και να μπορεί να ανακτήσει αργότερα τα δεδομένα κατευθείαν ως στιγμιότυπα κλάσεων VLongWritable και RegionWritable. Στην αρχή του αρχείου εξόδου γράφεται ως τιμή μια μεταβλητή RegionWritable με RegionWritable.id=IndexIdentifier και RegionWritable.Region=(το συνολικό MBR ολόκληρης της ρίζας του δέντρου). Στη συνέχεια γράφεται για καθένα από τους κόμβους-παιδιά της ρίζας (τους κόμβους του επιπέδου ακριβώς κάτω από τη ρίζα) ένα ζεύγος <VLongWritable, RegionWritable> με RegionWritable.id=IndexIdentifier και RegionWritable.Region=(το MBR του συγκεκριμένου κόμβου-παιδιού).



Εικόνα 4.10: Η φάση Reduce σχηματικά.

5. Εγγραφή αρχείου με πρόσθετες πληροφορίες για τα R-trees στο HDFS

Πριν τον τερματισμό της εφαρμογής, δημιουργούμε ένα επιπλέον αρχείο SequenceFile με όνομα rootsMBRs, όπου αποθηκεύουμε ζευγάρια <κλειδί, τιμή> της μορφής

<Text: όνομα αρχείου του εν λόγω R-tree, RegionWritable: IndexId και MBR ρίζας>

Για να δημιουργηθεί το αρχείο, ανοίγουμε ένα-ένα τα αρχεία εξόδου της εργασίας Hadoop, διαβάζουμε το πρώτο ζεύγος που περιέχει τις πληροφορίες που χρειαζόμαστε για τη ρίζα του εν λόγω δέντρου και τις γράφουμε στο νέο αρχείο με τη μορφή που αναφέραμε. Το αρχείο αυτό θα μας είναι χρήσιμο κατά την εκτέλεση επερωτημάτων με χρήση των R-trees.

5. Αλγόριθμοι εκτέλεσης Χωρικών Επερωτημάτων με χρήση του Packed Hilbert R-tree

5.1 Ανάλυση Σχεδιασμού: Λειτουργικότητα και απαιτήσεις

Στα πλαίσια αυτής της διπλωματικής εργασίας, εξετάσαμε μεθόδους για ταυτόχρονη εκτέλεση πολλών επερωτημάτων παράλληλα χρησιμοποιώντας τη δομή δεικτοδότησης Packed Hilbert R-tree που κατασκευάσαμε σε πρώτη φάση στο σύστημα αρχείων HDFS. Στόχος μας ήταν να ερευνήσουμε το πεδίο εφαρμογής που μπορεί να έχει το framework Hadoop σε περιπτώσεις εκτέλεσης επερωτημάτων που στοχεύουν σε ένα μικρό μέρος ενός συνόλου χωρικών δεδομένων.

Η εργασία αυτή επικεντρώθηκε στην ανάπτυξη αλγορίθμων που απαντούν μια δέσμη επερωτημάτων κατανεμημένα σε περιβάλλον Hadoop MapReduce, χρησιμοποιώντας ως βοηθητική δομή τα Packed Hilbert R-trees που βρίσκονται αποθηκευμένα στο HDFS. Ωστόσο, ο HDFSStorageManager που αναπτύξαμε και παρουσιάσαμε στο Κεφάλαιο 4, μας δίνει τη δυνατότητα να έχουμε πρόσβαση στο δέντρο και για ερωτήματα που θέτουμε μεμονωμένα. Για κάτι τέτοιο θα μπορούσαμε να χρησιμοποιήσουμε εφαρμογές που προσαρτούν εικονικά το HDFS σαν ένα οποιοδήποτε σύστημα αρχείων, παρέχοντας έτσι πρόσβαση στα αρχεία και τα δεδομένα του HDFS (αναφέρουμε ως παράδειγμα το project Hadoop Fuse).

Τέλος, θέλαμε οι αλγόριθμοι εκτέλεσης επερωτήσεων να χρησιμοποιούσαν τα δέντρα έχοντας πρόσβαση σ' αυτά απευθείας στο HDFS, χωρίς να απαιτείται η αποθήκευση των R-trees στον δίσκο ενός μηχανήματος. Αυτό το στοιχείο μπορεί να κάνει τους αλγορίθμους που θα αναπτυχθούν να μπορούν (στην ιδανική περίπτωση) να χρησιμοποιούν R-trees μεγάλου μεγέθους, πέρα από τους περιορισμούς ενός τοπικού δίσκου.

Στην εργασία αυτή θα παρουσιαστούν 3 διαφορετικές στρατηγικές για εκτέλεση μιας δέσμης χωρικών επερωτήσεων σε περιβάλλον Hadoop MapReduce. Ο λόγος για την ανάπτυξη 3 αλγορίθμων ήταν η προσπάθειά μας να προτείνουμε αλγορίθμους που να μπορούν να ανταπεξέλθουν ικανοποιητικά σε διαφορετικές συνθήκες, ανάλογα με την φύση των επερωτημάτων, το μέγεθος του cluster, και τον βαθμό ομοιομορφίας των χωρικών δεδομένων στα οποία στοχεύει το κάθε query.

Κάθε αλγόριθμος εκτέλεσης επερωτήσεων, δέχεται σαν είσοδο ένα αρχείο κειμένου, που σε κάθε γραμμή του περιέχει τα στοιχεία ενός query, στην εξής μορφή:

query id<\t>συντεταγμένες 2 σημείων του ορθογωνίου του query χωρισμένες με \t.

Οι αλγόριθμοι υποστηρίζουν την εκτέλεση τοπολογικών επερωτημάτων, και συγκεκριμένα ερωτημάτων εύρους ή Range Queries. Τα επερωτήματα εύρους χρησιμοποιούν τον χωρικό τελεστή επικάλυψης (intersection operator), που μπορεί να θεωρηθεί ως η γενική μορφή ενός συνόλου πιο εξειδικευμένων χωρικών τοπολογικών τελεστών. Ο τελεστής επικάλυψης ουσιαστικά εκφράζει εάν δυο χωρικά αντικείμενα επικαλύπτονται, είτε μερικώς είτε ολικώς. Εξειδικεύσεις αυτού του τελεστή είναι οι ακόλουθες χωρικές σχέσεις: συναντά (meet), ισούται με (equal), επικαλύπτει μερικώς (overlap), περιέχει (contains) και καλύπτει (covers), που παρουσιάσαμε στο Κεφάλαιο 1.3.1 . Στην υλοποίησή μας, στην κλάση spatialindex.Region που εκφράζει ένα χωρικό αντικείμενο, έχουμε υλοποιήσει όλους τους παραπάνω χωρικούς τελεστές. Έτσι, παρόλο που για τις ανάγκες της παρούσας εργασίας μελετήσαμε την απόδοση των υλοποιήσεων χρησιμοποιώντας τον ευρύτερο τελεστή *intersect*, η εφαρμογή μπορεί να υποστηρίξει και τις άλλες μορφές Range Queries με την κατάλληλη παραμετροποίηση.

Στην περίπτωση χρήσης R-trees ως δομής δεικτοδότησης, η πρόβλεψη της απόδοσης μιας εκτέλεσης ενός χωρικού επερωτήματος σχετίζεται άμεσα με τον αριθμό των κόμβων και των καταχωρήσεων του R-tree που θα χρειαστεί να προσπελαστούν. Σαν γενικό κανόνα, επιδιώκει κανείς το σχέδιο εκτέλεσης που θα απαιτεί πρόσβαση στους λιγότερους κόμβους του δέντρου, από τη στιγμή που ένας κόμβος αντιστοιχεί σε μια σελίδα του μέσου αποθήκευσης. Βέβαια, το πραγματικό κόστος μπορεί να είναι μικρότερο, όταν χρησιμοποιείται μια τεχνική ενδιάμεσης αποθήκευσης στη μνήμη με την χρήση ενός buffer. Μια τέτοια τεχνική buffering προβλέπεται και στους αλγορίθμους που υλοποιήθηκαν, με απλοϊκή μορφή.

Στις ενότητες του κεφαλαίου αυτού που ακολουθούν, θα παρουσιάσουμε τον σχεδιασμό και την υλοποίηση των τριών αλγορίθμων. Σε κάθε βήμα ανάπτυξης του εκάστοτε αλγορίθμου υπήρξε κριτήριο η όσο το δυνατόν καλύτερη δρομολόγηση των επερωτημάτων στους κόμβους επεξεργασίας, και η προσαρμογή της ιεραρχικής δομής των R-tree στις ιδιαιτερότητες του Hadoop και του HDFS.

5.2 Στρατηγική 1η: Εκτέλεση επερωτημάτων με κατανομή τους στο δέντρο που τα απαντά

5.2.1 Κίνητρα και γενική ιδέα του αλγορίθμου

Κατά την σχεδίαση του αλγορίθμου εκτέλεσης μιας δέσμης επερωτημάτων κατανεμημένα, επιδιώξαμε την όσο το δυνατόν πιο ισορροπημένη κατανομή του φόρτου εργασίας στους κόμβους επεξεργασίας. Στην αντίθετη περίπτωση, ο κόμβος που επεξεργάζεται την πιο χρονοβόρα διεργασία θα προκαλέσει καθυστέρηση σε όλο το σύστημα (bottleneck).

Κατά την σχεδίαση του 1ου αλγορίθμου, θέλαμε να κατανέμουμε τα επερωτήματα στους κόμβους με χωρικό κριτήριο, ανάλογα με το ποιο (ή ποια) από τα επιμέρους R-trees που βρίσκονταν αποθηκευμένα σε αρχεία του HDFS απαντούσε το κάθε ερώτημα. Σε περίπτωση που δεν εντοπίζαμε εξ' αρχής τα δέντρα που πιθανόν απαντούσαν το κάθε επιμέρους επερώτημα, το κάθε query θα έπρεπε να εκτελεστεί αναζητώντας όλα τα δέντρα, γεγονός που θα προκαλούσε αρχικοποίηση όλων των R-trees σε κάθε κόμβο επεξεργασίας, ακόμα και αν στην πραγματικότητα τα δεδομένα στα δέντρα αυτά δεν σχετίζονται με το query. Επιπλέον, η απλοϊκή προσέγγιση της δρομολόγησης όλων των επερωτημάτων σε κάθε κόμβο επεξεργασίας, ώστε ο κάθε κόμβος να εξετάζει μόνο ένα R-tree, θα προκαλούσε τεράστια επιβάρυνση στο δίκτυο του cluster χωρίς λόγο.

Ο αλγόριθμος που υλοποιήσαμε, κατά την φάση Map βρίσκει για κάθε επερώτημα της εισόδου το αρχείο (ή τα αρχεία) R-tree που απαντάει εν δυνάμει το επερώτημα, κρίνοντας από το MBR της ρίζας κάθε δέντρου, αλλά και από τα MBR των παιδιών της κάθε ρίζας για μεγαλύτερη αξιοπιστία. Στα τοπολογικά query, ο έλεγχος της συνθήκης που απαιτείται στο MBR της ρίζας είναι ένα πρώτο βήμα να βρούμε ποια δέντρα είναι πιθανό να απαντάνε το query: εάν το MBR της ρίζας δεν καλύπτει την συνθήκη, μπορούμε να αποκλείσουμε τα δεδομένα του δέντρου, εάν το MBR της ρίζας ικανοποιεί την συνθήκη τότε ίσως κάποια δεδομένα να την ικανοποιούν εξίσου μιας και το MBR της ρίζας περικλείει όλα τα δεδομένα του δέντρου. Επιλέξαμε να συμπεριλάβουμε στην αναζήτηση ικανοποίησης της συνθήκης και τα MBRs των άμεσων παιδιών της ρίζας για μεγαλύτερη αξιοπιστία και αποφυγή των false positive απαντήσεων: καθώς το MBR της ρίζας των δέντρων είναι γενίκευση των κάτω επιπέδων του δέντρου, και συχνά υπάρχει επικάλυψη μεταξύ των MBRs ριζών από διαφορετικά δέντρα, προχωράμε σε περεταίρω διερεύνηση για να μειώσουμε την επακάλυψη μεταξύ των δέντρων και να έχουμε πιο ασφαλή αποφάνσεις σχετικά με την πιθανότητα ενός δέντρου να απαντάει ένα query. Όπως θα εξηγήσουμε και στο κεφάλαιο 6.3 η πειραματική αξιολόγηση αυτής της στρατηγικής έδειξε ότι δίνει ασφαλή αποφάνσεις σε ποσοστό πάνω από 85%.

Η έξοδος της φάσης Map της εργασίας είναι ζευγάρια της μορφής <όνομα αρχείου που απαντάει το επερώτημα, επερώτημα>, τα οποία διαμοιράζονται στην φάση Reduce

χρησιμοποιώντας ως συνάρτηση partitioning την συνάρτηση κατακερματισμού του κλειδιού. Έτσι, κάθε Reducer αναλαμβάνει να εξετάσει 1 ή περισσότερα δέντρα, λαμβάνοντας στην είσοδό του ζεύγη της μορφής

<όνομα αρχείου του R-tree, λίστα με όλα τα queries που προορίζονται γι' αυτό το δέντρο>.

Για κάθε όνομα αρχείου η διεργασία Reduce ξεκινάει μια κλήση της συνάρτησης reduce(), η οποία αρχικοποιεί το R-tree που υπάρχει στο αντίστοιχο αρχείο της τιμής κλειδιού, εκτελεί τα επερωτήματα διατρέχοντας το δέντρο και τέλος, αποθηκεύει τα αποτελέσματα για κάθε επερώτημα στο αρχείο εξόδου ως ένα ή περισσότερα ζευγάρια με τη μορφή

<query_id, id χωρικού αντικειμένου που απαντά στο query>

Η παραπάνω στρατηγική υλοποιείται στην κλάση Rtree_quering_FilenamePartitioning.

5.2.2 Περιγραφή της υλοποίησης

Στην ενότητα αυτή θα αναλύσουμε περισσότερο κάθε βήμα της υλοποίησής μας.

1. **Προετοιμασία εργασίας και Διάβασμα Εισόδου**

   Για την εκτέλεση της εργασίας που , ο χρήστης παρέχει στην συνάρτηση main της κλάσης Rtree_quering_FilenamePartitoning τις παρακάτω παραμέτρους:

Όνομα Παραμέτρου	**Περιγραφή**
args[0]	InputPath: Το αρχείο με τα επερωτήματα εισόδου. Σε περίπτωση που δωθεί directory, ως είσοδος λαμβάνονται όλα τα αρχεία του directory.
args[1]	OutputPath: Το directory όπου θα αποθηκευτούν τα αρχεία εξόδου της εργασίας Hadoop με τα αποτελέσματα των queries.
args[2]	numberOfReducers: Αριθμός των διεργασιών Reduce.
args[3]	TreeINFOfiles.directory: Το directory όπου βρίσκονται αποθηκευμένα τα αρχεία με τις πληροφορίες για τα R-trees (δηλ. το directory εξόδου της εργασίας Hadoop που

	κατασκεύασε το Index). Η τιμή του φορτώνεται στην παράμετρο *Rtree_test_epikalypsi.TreeINFOfiles.directory* του αρχείου Configuration της εργασίας, ώστε να μπορεί να ανακτηθεί από όλες τις διεργασίες (Map ή Reduce).
args[4]	Treefiles.directory: Το directory όπου βρίσκονται αποθηκευμένα τα αρχεία με τα R-trees. Η τιμή του φορτώνεται στην παράμετρο *Rtree_test_epikalypsi.Treefiles.directory* του αρχείου Configuration της εργασίας, ώστε να μπορεί να ανακτηθεί από όλες τις διεργασίες (Map ή Reduce).
args[5]	number_of_Rtrees: Ο αριθμός των επιμέρους R-trees που απαρτίζουν το Index. Η τιμή του φορτώνεται στην παράμετρο *Rtree_test_epikalypsi.numberOfTrees.constructed* του αρχείου Configuration της εργασίας, ώστε να μπορεί να ανακτηθεί από όλες τις διεργασίες (Map ή Reduce).
args[6]	allMBRsInMemory: Εάν έχει τιμή true, για την εύρεση των αρχείων R-tree που απαντάνε κάθε επερώτημα κρατάμε τα MBRs όλων των παιδιών των ριζών στη μνήμη. Αλλιώς, τα διαβάζουμε μέσα από τα αρχεία του TreeINFOfiles.directory κάθε φορά που χρειάζεται. Η τιμή της φορτώνεται στην παράμετρο *Rtree_test_epikalypsi.loadTreeRootMBRs.InMemory* του αρχείου Configuration της εργασίας, ώστε να μπορεί να ανακτηθεί από όλες τις διεργασίες (Map ή Reduce).

Πίνακας 5.1: Παράμετροι που δίνει ο χρήστης στην συνάρτηση main της εργασίας Hadoop.

Πριν δρομολογηθεί η εκτέλεση της εργασίας Hadoop, η συνάρτηση main() αναλαμβάνει να τοποθετήσει όλα τα αρχεία με τις πληροφορίες για τα επιμέρους R-tree που υπάρχουν στο TreeINFOfiles.directory στην Distributed Cache, ώστε να βρίσκονται στο τοπικό σύστημα αρχείων κάθε κόμβου του cluster, άμεσα διαθέσιμα για διάβασμα με χαμηλό κόστος. Να σημειώσουμε, ότι το συνολικό μέγεθος του κάθε αρχείου που περιέχεται στο φάκελο έχει μέγεθος της τάξης των 100 KB ή και μικρότερο (σε περίπτωση που η ρίζα έχει λιγότερα από IndexCapacity παιδιά). Έτσι, ακόμα και εάν έχουμε λίγες εκατοντάδες R-trees (δηλ. εάν το Index δημιουργήθηκε χρησιμοποιώντας μερικές εκατοντάδες Reducers), όλα τα αρχεία που θα μεταφέρουμε δεν θα ξεπερνούν τα λίγα MegaBytes, μέγεθος αποδεκτό για μεταφορά στην Distributed Cache. Επιπλέον, η main() τοποθετεί στην Distributed

Cache και το αρχείο με όνομα rootsMBRs που περιέχει τα MBRs των ριζών όλων των επιμέρους R-trees. Τα αρχεία αυτά, θα χρησιμοποιηθούν κατά την φάση Map για τον καθορισμό των δέντρων που απαντάνε κάθε επερώτημα.

Έπειτα αρχίζει η εκτέλεση της κύριας εργασίας Hadoop, και τα επερωτήματα προς εκτέλεση διαβάζονται από τα αρχεία εισόδου ώστε να προετοιμαστούν τα ζευγάρια <κλειδί, τιμή> που θα αποτελέσουν είσοδο στις διεργασίες Map. Επιλέξαμε όπως αναφέραμε ήδη να θεωρήσουμε ως είσοδο της εργασίας απλά αρχεία κειμένου, ώστε να τηρηθεί η γενικότητα και να μην υπάρχει προαπαιτούμενη προεπεξεργασία των επερωτημάτων. Έτσι, η είσοδος διαβάζεται χρησιμοποιώντας την κλάση TextInputFormat που δημιουργεί ζευγάρια από κάθε γραμμή του κειμένου εισόδου. Η μορφή των ζευγαριών που θα τροφοδοτηθούν στην φάση Map είναι η ακόλουθη:

<κλειδί=θέση γραμμής που διαβάσαμε, τιμή=ολόκληρη η γραμμή κειμένου>

2. Φάση Map

Η φάση Map λαμβάνει στην είσοδό της input splits, και δρομολογεί μια διεργασία Map για το καθένα για να τα επεξεργαστεί.

Για κάθε διεργασία Map εκτελείται μια φορά στην αρχή της η συνάρτηση

protected void setup(Context context) throws IOException;

στην οποία μπορούμε να ορίσουμε κάποιες ενέργειες που θέλουμε να εκτελεστούν μια φορά, πριν την έναρξη των κλήσεων της συνάρτησης map για κάθε record μέσα στο input split. Στην υλοποίησή μας, έχουμε ορίσει στην συνάρτηση αυτή να φορτώνονται κάποιες πληροφορίες για τα R-trees σε δομές της μνήμης, ώστε να γίνεται πιο εύκολη και γρήγορη η διαδικασία εύρεσης του δέντρου που απαντά κάθε επερώτημα (η διαδικασία αυτή θα πραγματοποιείται αργότερα στην συνάρτηση map).

Κατά την κλήση της setup στην αρχικοποίηση κάθε διεργασίας Map, φορτώνονται στην δομή rootMBRs (τύπου HashMap<Text, Region>) ζεύγη με το όνομα του αρχείου που περιέχει το επιμέρους R-tree και το συνολικό MBR της ρίζας του, για κάθε R-tree. Τα στοιχεία αυτά διαβάζονται από το αρχείο τύπου SequenceFile με όνομα rootsMBRs που έχουμε βάλει στην Distributed Cache. Έπειτα, εξετάζεται η τιμή της παραμέτρου *Rtree_test_epikalypsi.loadTreeRootMBRs.InMemory* του αρχείου Configuration της εργασίας: εάν έχει τιμή true κρατάμε τα MBRs όλων των παιδιών των ριζών των R-trees στη μνήμη, αποθηκεύοντας στην δομή allMBRs (τύπου ArrayList<Region>) το συνολικό MBR κάθε παιδιού της ρίζας, για κάθε R-tree. Τέλος, η δομή HashMap<Text,TwoIntegerLimits> fileLimitIndex μας βοηθάει

να εντοπίσουμε την θέση αρχής και τέλους των MBRs που αφορούν το κάθε επιμέρους δέντρο στο διάνυσμα allMBRs.

Αφού τελειώσει η εκτέλεση της συνάρτησης setup, η διεργασία καλεί μια φορά τη συνάρτηση map για κάθε record μέσα στο input split,, με είσοδο το ζεύγος <τιμή, κλειδί> του record αυτού. Η συνάρτηση map στην υλοποίησή μας, δέχεται είσοδο <κλειδί, τιμή> της μορφής

<VLongWritable:θέση γραμμής που διαβάσαμε, Text:ολόκληρη η γραμμή κειμένου>
Με την τιμή συγκεκριμένα να έχει την γενική μορφή:
query_id <\t> συντεταγμένες 2 σημείων του ορθογωνίου του query χωρισμένες με \t.
Η τιμή του κλειδιού ουσιαστικά μας είναι αδιάφορη.

Η συνάρτηση map διαβάζει την τιμή του ζεύγους και από αυτήν εξάγει το id του χωρικού αντικειμένου και τις συντεταγμένες των 2 σημείων του ορθογωνίου του επερωτήματος. Από τις συντεταγμένες κατασκευάζεται ένα στιγμιότυπο new_region της κλάσης spatialindex.Region που αναπαριστά το ορθογώνιο-«παράθυρο» του query. Στην συνέχεια εκτελείται ο παρακάτω αλγόριθμος, ο οποίος εντοπίζει το αρχείο με το δέντρο (ή τα αρχεία με τα δέντρα) που εν γένει απαντά/απαντούν το επερώτημα, και τα βάζει σε μια λίστα.

Αλγόριθμος *Find_Treefiles_That_Potentially_Answer_The_Query:* εντοπίζει το αρχείο με το δέντρο (ή τα αρχεία με τα δέντρα) που εν γένει απαντά/απαντούν το επερώτημα Q με παράθυρο new_region, και τα βάζει σε 2 ArrayList μαζί με το IndexIdentifier του αντίστοιχου δέντρου.

Βήμα 1. /* αρχή */

Κατασκεύασε ένα κενό ArrayList με όνομα filenames.

Κατασκεύασε ένα κενό ArrayList με όνομα headerIDs.

Βήμα 2. /* Έλεγχος αν το MBR της ρίζας του δέντρου L απαντά στο query */

While (υπάρχουν ακόμα δέντρα) {

Διάβασε από την δομή rootMBRs το συνολικό MBR της ρίζας του επόμενου δέντρου L σε μια μεταβλητή Region r.

Αν (r. intersects(new_region){

/* Έλεγχος αν κάποιο MBR παιδιού της ρίζας του δέντρου L απαντά στο query */

Αν(*Rtree_test_epikalypsi.loadTreeRootMBRs.InMemory==true*){

Για το δέντρο L, έλενξε με τη βοήθεια της δομής

allMBRs εάν για κάποιο MBR παιδιού της ρίζας του δέντρου, έστω memory_region, ισχύει memory_region.intersects(new_region).

Εάν ναι{

Σταμάτα την αναζήτηση σ' αυτό το δέντρο.

Πρόσθεσε στο filenames το όνομα του αρχείου με το R-tree L.

Πρόσθεσε στο headerIDs το IndexIdentifier του R-tree L.

}

}

Αλλιώς
Αν(*Rtree_test_epikalypsi.loadTreeRootMBRs.InMemory==false*){

/* πραγματοποίησε τους ελέγχους διαβάζοντας από το αρχείο που βρίσκονται οι πληροφορίες του R-tree L */

Άνοιξε το αρχείο (τύπου Sequence File) που περιέχει τις πληροφορίες για το δέντρο L.

Για το δέντρο L, έλενξε με τη βοήθεια του αρχείου εάν για κάποιο MBR παιδιού της ρίζας του δέντρου, έστω file_region, ισχύει file_region.intersects(new_region).

Εάν ναι{

Σταμάτα την αναζήτηση σ' αυτό το δέντρο.

Πρόσθεσε στο filenames το όνομα του αρχείου με το R-tree L.

Πρόσθεσε στο headerIDs το IndexIdentifier του R-tree L.

}

}

}

Συνέχισε για το επόμενο δέντρο L=L+1.

}

Πίνακας 5.2: Αλγόριθμος που εντοπίζει τα αρχεία με τα δέντρα που εν δυνάμει απαντούν ένα επερώτημα.

Στη συνέχεια, η συνάρτηση map για κάθε i-οστό μέλος της λίστας headerIDs σχηματίζει ένα στιγμιότυπο queryInfo της κλάσης QueryWritable και αποθηκεύει σ' αυτό το id του query προς απάντηση, το ορθογώνιο-«παράθυρο» του query και το IndexIdentifier του R-tree που βρίσκεται στη θέση i της λίστας headerIDs. Τέλος, αποθηκεύει το αντίστοιχο μέλος i της λίστας filenames σε μια μεταβλητή τύπου Text. Στο τέλος, για κάθε i-οστό ζεύγος που δημιούργησε δίνει έξοδο <κλειδί, τιμή> της μορφής

<*Text:* όνομα αρχείου που περιέχει το R-tree, QueryWritable*:* το επερώτημα queryInfo προς εκτέλεση>

3. Φάση Partitioning

Η εργασία Hadoop χρησιμοποιεί ως partitioning συνάρτηση την default συνάρτηση κατακερματισμού του κλειδιού κάθε ζεύγους.

Άρα, η διαδικασία partitioning και η διαδικασία Shuffle (μεταφορά της διαμέρισης στον κόμβο που θα εκτελεστεί η διεργασία reduce και συνολική ταξινόμησή της) θα μας δώσουν R διαμερίσεις, μια σε κάθε Reducer, που μπορεί η καθεμιά να περιέχει ζευγάρια με διαφορετικές τιμές κλειδών (αρχεία με διαφορετικά R-trees), αλλά για κάθε τιμή κλειδιού περιέχει όλα τα ζευγάρια της φάσης Map που έχουν αυτή την τιμή για κλειδί τους (όλα τα επερωτήματα που απαντά το κάθε επιμέρους R-tree).

4. Φάση Reduce και εγγραφή αποτελεσμάτων

Κάθε Reducer αναλαμβάνει μια διαμέριση. Τα ζευγάρια της διαμέρισης ταξινομούνται και ομαδοποιούνται στη συνέχεια με βάση το κλειδί. Έτσι, δημιουργούνται ζευγάρια της μορφής

< Text: όνομα αρχείου που περιέχει το R-tree, Iterable<QueryWritable>: λίστα με όλα τα επερωτήματα που εν δυνάμει απαντά το συγκεκριμένο δέντρο>

και για κάθε ζεύγος της παραπάνω μορφής εκτελείται μια φορά η συνάρτηση reduce.

Η συνάρτηση reduce αρχικοποιεί αρχικά τον HDFSStorageManager, περνώντας ως παράμετρο *FileName* το όνομα του αρχείου που περιέχει το Index. Στη συνέχεια το νέο στιγμιότυπο χρησιμοποιείται για την αρχικοποίηση ενός στιγμιότυπου της κλάσης RandomEvictionsBuffer, που αναλαμβάνει τη δημιουργία ενός buffer σαν ενδιάμεσο επίπεδο αποθήκευσης. Η RandomEvictionsBuffer αποθηκεύει μια σελίδα που ανακτά από το HDFS αν δεν υπάρχει στο buffer ήδη. Για την αντικατάσταση μιας υπάρχουσας σελίδας ακολουθεί μια replacement policy βασισμένη σε γεννήτρια τυχαίων αριθμών. Αρχικοποιούμε τον buffer έχει 10 θέσεις αποθήκευσης,

καθώς δεδομένου του υψηλού fan-out των κόμβων του R-tree που κατασκευάσαμε, το δέντρο αναμένεται να έχει χαμηλό ύψος, άρα οι 10 θέσεις είναι μια καλή ισορροπία ανάμεσα στο κόστος επιβάρυνσης της μνήμης και στο κέρδος από τη χρήση του για μείωση των λειτουργιών ανάγνωσης από το HDFS.

Σειρά έχει η αρχικοποίηση του R-tree που βρίσκεται αποθηκευμένο στο αρχείο με όνομα την τιμή κλειδιού του ζεύγους εισόδου της συνάρτησης reduce. Για την αρχικοποίησή του αρκεί να δώσουμε στην παράμετρο *IndexIdentifier* το id του δέντρου, που το ανακτάμε από μια μεταβλητή της λίστας τιμών που λαμβάνει η reduce ως είσοδο (χρησιμοποιώντας την συνάρτηση *QueryWritable*.getTreeHeaderId()).

Στην συνέχεια, για κάθε επερώτημα της λίστας τιμών εισόδου φτιάχνεται ένα στιγμιότυπο r της κλάσης Region και εκτελείται το query με την κλήση της συνάρτησης Rtree.intersectionQuery(r, vis). Η παράμετρος vis είναι στιγμιότυπο της εσωτερικής κλάσης MyVisitor, που αναλαμβάνει την εγγραφή των αποτελεσμάτων κάθε query στο αρχείο εξόδου. Η MyVisitor υλοποιεί την διεπαφή IVisitor, αρχικοποιείται για κάθε επερώτημα και λαμβάνει κατά την αρχικοποίησή της ως παράμετρο το στιγμιότυπο της κλάσης Context που έχει λάβει και η συνάρτηση reduce(). Μέσω της συνάρτησης context.write(κλειδί, τιμή) η MyVisitor μπορεί να εκτυπώνει τα αποτελέσματα του κάθε query στο αρχείο εξόδου της εργασίας Hadoop, εξάγοντας ένα ζεύγος ανά αποτέλεσμα στη μορφή <κλειδί, τιμή>:

< VLongWritable: query_id , VLongWritable :id χωρικού αντικειμένου που απαντά στο query >

Να σημειωθεί ότι τα χωρικά αποτελέσματα που απαντούν ένα επερώτημα βρίσκονται με την βοήθεια της συνάρτησης Rtree.rangeQuery που καλείται από την Rtree.intersectionQuery. Η συνάρτηση αυτή υλοποιείται στην βιβλιοθήκη spatial index και λαμβάνει ως παραμέτρους τον τύπο του επερωτήματος, την περιοχή («παράθυρο») του επερωτήματος, και ένα στιγμιότυπο κλάσης που υλοποιεί την διεπαφή IVisitor που θα αναλάβει την εξαγωγή/εκτύπωση των αποτελεσμάτων. Ανάλογα με τον τύπο του επερωτήματος, η συνάρτηση Rtree.rangeQuery δρομολογεί τις απαραίτητες ενέργειες. Για τους κοινούς τύπους queries (όπως στην περίπτωσή μας το intersection query) ξεκινά την σάρωση του R-tree από τη ρίζα του και προχωρά προς τα κάτω επίπεδα αναδρομικά, ελέγχοντας κάθε φορά όλα τα παιδιά του τρέχοντος κόμβου εάν πληρούν μια συνθήκη που επιβάλει το εκάστοτε επερώτημα. Όταν η συνάρτηση φτάσει σε επίπεδο κόμβων-φύλλων και βρει ένα χωρικό αντικείμενο που ικανοποιεί το query, καλεί την συνάρτηση IVisitor.visitData() για να εκτυπωθεί/εξαχθεί το συγκεκριμένο αντικείμενο ανάμεσα στα αποτελέσματα.

5.3 Στρατηγική 2η: Εκτέλεση επερωτημάτων με ισοκατανομή τους στις διεργασίες επεξεργασίας.

5.3.1 Κίνητρα και γενική ιδέα του αλγορίθμου

Όπως προαναφέραμε, κατά την σχεδίαση του αλγορίθμου εκτέλεσης μιας δέσμης επερωτημάτων κατανεμημένα, επιδιώξαμε την όσο το δυνατόν πιο ισορροπημένη κατανομή του φόρτου εργασίας στους κόμβους επεξεργασίας. Στην αντίθετη περίπτωση, ο κόμβος που επεξεργάζεται την πιο χρονοβόρα διεργασία θα προκαλέσει καθυστέρηση σε όλο το σύστημα (bottleneck).

Κατά την σχεδίαση του 1ου αλγορίθμου, θέλαμε να κατανέμουμε τα επερωτήματα στους κόμβους με χωρικό κριτήριο, ανάλογα με το ποιο (ή ποια) από τα επιμέρους R-trees που βρίσκονταν αποθηκευμένα σε αρχεία του HDFS απαντούσε το κάθε ερώτημα. Η στρατηγική αυτή, αναμένουμε να έχει καλά αποτελέσματα στην περίπτωση μιας δέσμης επερωτημάτων που στοχεύει σε δεδομένα απ' όλα τα δέντρα (ή απ' τα περισσότερα), σχετικά ομοιόμορφα. Επιπλέον, η στρατηγική αυτή είναι επεκτάσιμη μόνο μέχρι ο αριθμός των Reducers να φτάσει τον αριθμό των επιμέρους R-trees που απαρτίζουν το Index, αφού λόγω της κατανομής των επερωτημάτων βάσει των αρχείων που περιέχουν ένα R-tree, οι επιπλέον Reducers θα παραμείνουν ουσιαστικά αδρανείς.

Για το λόγο αυτό προχωρήσαμε στην σχεδίαση και υλοποίηση ενός εναλλακτικού αλγόριθμου για εκτέλεση των επερωτημάτων, ακριβώς για την περίπτωση που τα επερωτήματα στοχεύουν σε ένα υποσύνολο μόνο του δέντρου, ή στην γενική περίπτωση δεν στοχεύουν «ομοιόμορφα» σε όλα τα R-trees που απαρτίζουν το συνολικό index μας. Επιπλέον, θέλαμε ο νέος αλγόριθμος να μπορεί να εκμεταλλευτεί αριθμό Reducers που θα ξεπερνούσαν τον αριθμό των επιμέρους R-trees.

Το σημείο που διαφοροποιεί την τεχνική αυτή από την προηγούμενη είναι ο τρόπος διαμοιρασμού των επερωτημάτων στις διεργασίες Reduce. Αυτή τη φορά, επιδιώκουμε ίση κατανομή των επερωτημάτων στους Reducers, ανεξάρτητα από το R-tree που εν δυνάμει απαντά το κάθε επερώτημα.

Συνοπτικά, ο αλγόριθμος που υλοποιήσαμε, κατά την φάση Map βρίσκει για κάθε επερώτημα της εισόδου το αρχείο (ή τα αρχεία) R-tree που απαντάει εν δυνάμει το επερώτημα, κρίνοντας από το MBR της ρίζας κάθε δέντρου, αλλά και από τα MBR των παιδιών της κάθε ρίζας για μεγαλύτερη αξιοπιστία. Η έξοδος της φάσης Map της εργασίας είναι ζευγάρια της μορφής *<όνομα αρχείου που απαντάει το επερώτημα, επερώτημα>*. Μέχρι το σημείο αυτό, δεν υπάρχει διαφοροποίηση με τον αλγόριθμο της 1ης στρατηγικής.

Τα ζεύγη από τη φάση Map διαμοιράζονται στην φάση Reduce με τυχαίο τρόπο (βάσει ομοιόμορφης κατανομής), ώστε κάθε Reducer να λάβει ίσο περίπου αριθμό

επερωτημάτων. Κάθε Reducer λαμβάνει στην γενική περίπτωση ζεύγη με έναν αριθμό από διαφορετικές τιμές κλειδιού, τα οποία ομαδοποιούνται στη συνέχεια ανάλογα με την τιμή κλειδιού. Έτσι, κάθε διεργασία Reduce αναλαμβάνει να εξετάσει 1 ή περισσότερα δέντρα, λαμβάνοντας στην είσοδό του ζεύγη της μορφής

<όνομα αρχείου του R-tree, λίστα με όλα τα queries που προορίζονται γι' αυτό το δέντρο>.

Για κάθε όνομα αρχείου η διεργασία Reduce ξεκινάει μια κλήση της συνάρτησης reduce(), η οποία αρχικοποιεί το R-tree που υπάρχει στο αντίστοιχο αρχείο της τιμής κλειδιού, εκτελεί τα επερωτήματα διατρέχοντας το δέντρο και τέλος, αποθηκεύει τα αποτελέσματα για κάθε επερώτημα στο αρχείο εξόδου ως ένα ή περισσότερα ζευγάρια με τη μορφή

<query_id, id χωρικού αντικειμένου που απαντά στο query>

Η παραπάνω στρατηγική υλοποιείται στην κλάση Rtree_quering_RandomPartitioning_clever.

5.3.2 Περιγραφή της υλοποίησης

Ακολουθεί η ανάλυση κάθε μέρους της υλοποίησής μας.

1. **Προετοιμασία εργασίας και Διάβασμα Εισόδου**

   Για την εκτέλεση της εργασίας που , ο χρήστης παρέχει στην συνάρτηση main της κλάσης Rtree_quering_FilenamePartitoning τις ίδιες παραμέτρους που απαιτούσε και η 1η στρατηγική.

   Η προετοιμασία για την εκτέλεση της εργασίας Hadoop γίνεται ακριβώς με τον ίδιο τρόπο όπως και στον αλγόριθμο που περιγράψαμε στην ενότητα 5.2.2: η συνάρτηση main() αναλαμβάνει να τοποθετήσει όλα τα αρχεία με τις πληροφορίες για τα επιμέρους R-tree (που υπάρχουν στο TreeINFOfiles.directory) και το αρχείο με όνομα rootsMBRs που περιέχει τα MBRs των ριζών τους στην Distributed Cache, ώστε να βρίσκονται στο τοπικό σύστημα αρχείων κάθε κόμβου του cluster, άμεσα διαθέσιμα για διάβασμα με χαμηλό κόστος. Έπειτα αρχίζει η εκτέλεση της κύριας εργασίας Hadoop, και τα επερωτήματα προς εκτέλεση διαβάζονται από τα αρχεία κειμένου της εισόδου ώστε να προετοιμαστούν τα ζευγάρια <κλειδί, τιμή> που θα αποτελέσουν είσοδο στις διεργασίες Map. Η μορφή των ζευγαριών που θα τροφοδοτηθούν στην φάση Map είναι η ακόλουθη:

   <κλειδί=θέση γραμμής που διαβάσαμε, τιμή=ολόκληρη η γραμμή κειμένου>

2. **Φάση Map**

Η φάση Map λαμβάνει στην είσοδό της input splits, και δρομολογεί μια διεργασία Map για το καθένα για να τα επεξεργαστεί.

Η κάθε διεργασία Map εκτελείται με τον ίδιο τρόπο που περιγράψαμε στην ενότητα 5.2.2: καλείται μια φορά στην αρχή της η συνάρτηση *setup* στην οποία φορτώνονται κάποιες πληροφορίες για τα R-trees σε δομές της μνήμης (δομές rootMBRs, allMBRs, fileLimitIndex), ώστε να γίνεται πιο εύκολη και γρήγορη η διαδικασία εύρεσης του δέντρου που απαντά κάθε επερώτημα. Στη συνέχεια καλείται μια φορά η συνάρτηση map για κάθε record μέσα στο input split,, με είσοδο το ζεύγος <τιμή, κλειδί> του record αυτού. Η συνάρτηση κατασκευάζει ένα στιγμιότυπο new_region της κλάσης spatialindex.Region που αναπαριστά το ορθογώνιο-«παράθυρο» του query και εντοπίζει το αρχείο με το δέντρο (ή τα αρχεία με τα δέντρα) που εν γένει απαντά/απαντούν το επερώτημα. Στο τέλος, για κάθε R-tree που εν δυνάμει απαντά το query δίνει έξοδο <κλειδί, τιμή> της μορφής

<*Text:* όνομα αρχείου που περιέχει το R-tree, QueryWritable*:* το επερώτημα queryInfo προς εκτέλεση>

3. **Φάση Partitioning**

Η εργασία Hadoop αυτή χρησιμοποιεί ως partitioning κλάση την custom κλάση MyRandomPartitioner που υλοποιήσαμε και κληρονομεί την γενική κλάση Partitioner. Σ' αυτήν, η συνάρτηση getPartition διαμοιράζει τα ζευγάρια σε διαμερίσεις με τυχαίο τρόπο (βάσει ομοιόμορφης κατανομής), ώστε κάθε Reducer να λάβει ίσο περίπου αριθμό επερωτημάτων.

Άρα, η διαδικασία partitioning και η διαδικασία Shuffle (μεταφορά της διαμέρισης στον κόμβο που θα εκτελεστεί η διεργασία reduce και συνολική ταξινόμησή της) θα μας δώσουν R ίσες περίπου στο μέγεθος διαμερίσεις, μια σε κάθε Reducer, που μπορεί η καθεμιά να περιέχει ζευγάρια με διαφορετικές τιμές κλειδών (αρχεία με διαφορετικά R-trees).

4. **Φάση Reduce και εγγραφή αποτελεσμάτων**

Κάθε Reducer αναλαμβάνει μια διαμέριση. Τα ζευγάρια της διαμέρισης ταξινομούνται και ομαδοποιούνται στη συνέχεια με βάση το κλειδί. Έτσι, δημιουργούνται ζευγάρια της μορφής

< Text: όνομα αρχείου που περιέχει το R-tree, Iterable<QueryWritable>: λίστα με όλα τα queries που εν δυνάμει απαντά το συγκεκριμένο R-tree σ' αυτή τη διαμέριση>

και για κάθε ζεύγος της παραπάνω μορφής εκτελείται μια φορά η συνάρτηση reduce.

Η συνάρτηση reduce έχει ακριβώς την ίδια υλοποίηση με την αντίστοιχη συνάρτηση reduce της 1ης στρατηγικής που περιγράψαμε στην ενότητα 5.2.2. Συνοπτικά υπενθυμίζουμε: Αρχικοποιείται αρχικά ο HDFSStorageManager και ο RandomEvictionsBuffer για την διαχείριση των λειτουργιών I/O από το HDFS και τη δημιουργία buffer ενδιάμεσης αποθήκευσης. Σειρά έχει η αρχικοποίηση του R-tree που βρίσκεται αποθηκευμένο στο αρχείο με όνομα την τιμή κλειδιού του ζεύγους εισόδου της συνάρτησης reduce. Για κάθε επερώτημα της λίστας τιμών εισόδου φτιάχνεται ένα στιγμιότυπο r της κλάσης Region και εκτελείται το query με την κλήση της συνάρτησης Rtree.intersectionQuery(r, vis). Η παράμετρος vis είναι στιγμιότυπο της εσωτερικής κλάσης MyVisitor, που αναλαμβάνει την εγγραφή των αποτελεσμάτων κάθε query στο αρχείο εξόδου. Μέσω της συνάρτησης context.write(κλειδί, τιμή) η MyVisitor μπορεί να εκτυπώνει τα αποτελέσματα του κάθε query στο αρχείο εξόδου της εργασίας Hadoop, εξάγοντας ένα ζεύγος ανά αποτέλεσμα στη μορφή <κλειδί, τιμή>:

< VLongWritable: query_id , VLongWritable :id χωρικού αντικειμένου που απαντά στο query >

5.4 Στρατηγική 3η: Εκτέλεση επερωτημάτων με προεπεξεργασία

5.4.1 Κίνητρα και γενική ιδέα του αλγορίθμου

Η δεύτερη στρατηγική εκτέλεσης δέσμης επερωτημάτων που σχεδιάσαμε και υλοποιήσαμε όντως δίνει μια λύση για την περίπτωση που τα επερωτήματα στοχεύουν σε ένα υποσύνολο μόνο του δέντρου, ή στην γενική περίπτωση δεν στοχεύουν «ομοιόμορφα» σε όλα τα R-trees που απαρτίζουν το συνολικό index μας. Επιπλέον, ο νέος αλγόριθμος φαίνεται να μπορεί να εκμεταλλευτεί έναν αριθμό Reducers που θα ξεπερνάει τον αριθμό των επιμέρους R-trees.

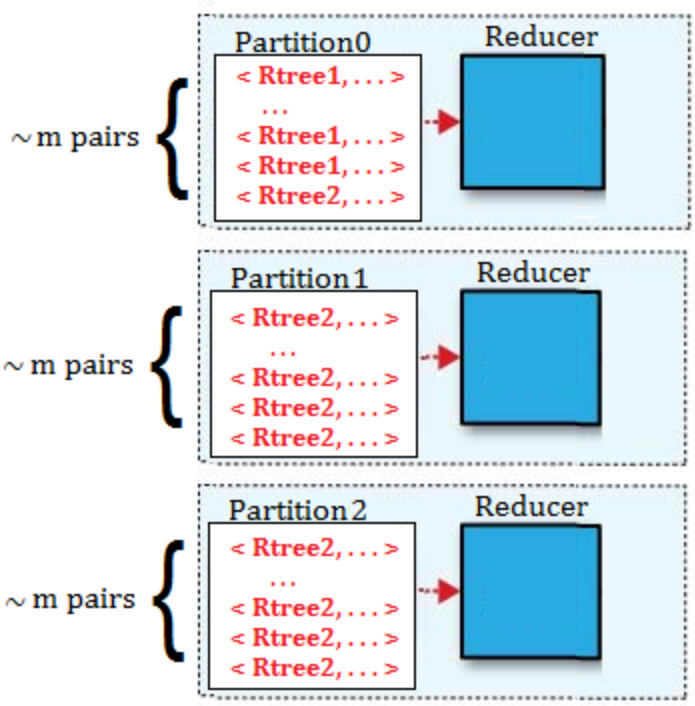
Ωστόσο, παραμένει πρόβλημα το γεγονός ότι σε κάθε Reducer αναλογεί ένα partition που περιέχει επερωτήματα που στην γενική περίπτωση απευθύνονται στο σύνολο των δέντρων που απαρτίζουν το Index και απαντούν τα επερωτήματα. Έτσι, αν υποθέσουμε π.χ. ότι τα επερωτήματα στο σύνολό τους στοχεύουν σε δεδομένα που δεικτοδοτούνται

από 5 επιμέρους δέντρα, κάθε διεργασία Reduce θα πρέπει ενδεχομένως να καλέσει τη συνάρτηση reduce έως και 5 φορές και να αρχικοποιήσει 5 δέντρα και να εκτελέσει τα ανάλογα επερωτήματα, κάτι που σίγουρα καθυστερεί την συνολική απόδοση του συστήματος.

Είναι κατανοητό, ότι η ιδανική λύση θα ήταν αυτή που θα μπορούσε να διαμοιράσει τα επερωτήματα στους Reducers με χωρικό κριτήριο (ανάλογα π.χ. με το R-tree που εν δυνάμει απαντάει το εκάστοτε επερώτημα), αλλά θα είχε επίσης τη δυνατότητα οι διαμερίσεις αυτές να έχουν τον ίδιο περίπου αριθμό επερωτημάτων. Επιπλέον, επιδιώκουμε η στρατηγική αυτή να ελαχιστοποιεί τον αριθμό διαφορετικών δέντρων που θα πρέπει να σαρώσει ο κάθε Reducer.

Στην ενότητα αυτή θα παρουσιάσουμε τον σχεδιασμό και την υλοποίηση ενός αλγορίθμου που φαίνεται να πληροί τις παραπάνω απαιτήσεις. Ο αλγόριθμος δανείζεται την ιδέα της δειγματοληψίας της εισόδου και της χρήσης του TotalOrderPartitioner που χρησιμοποιήσαμε κατά την κατασκευή του Packed Hilbert R-tree στο HDFS. Όπως είδαμε στο Κεφάλαιο 4.3.1 με τη μέθοδο αυτή μπορούμε να διαχωρίσουμε το εύρος τιμών που παίρνει το κλειδί των ζευγαριών στην έξοδο των Mappers μας σε R μέρη, ώστε σε κάθε διάστημα να υπάρχουν περίπου ίσα ζευγάρια με τιμή κλειδιού σ' αυτό το εύρος. Οι τιμές-όρια των R διαστημάτων έχουν προκύψει από δειγματοληψία της εισόδου που κάναμε πριν ξεκινήσει η εργασία Hadoop για την κατασκευή του δέντρου.

Η Εικόνα 5.1 δείχνει σε γενικές γραμμές αυτό που θέλουμε να πετύχουμε, το «άπλωμα» κατά μια έννοια της τιμής που παίρνει το κλειδί κατά μήκος των partitions. Η εικόνα δείχνει τον τρόπο που θα θέλαμε να διαμοιράζουμε τα ζευγάρια ανάλογα με το ποιο R-tree τα απαντά, στην περίπτωση που έχουμε 3 Reducers και επερωτήματα που απαντώνται από 2 αρχεία με επιμέρους R-trees.



Εικόνα 5.1: Partitioning ζευγαριών ανάλογα με το ποιο R-tree τα απαντά, στην περίπτωση που έχουμε 3 Reducers και επερωτήματα που απαντώνται από 2 αρχεία με επιμέρους R-trees.

Οι διαμερίσεις που φτιάχνει η υλοποίησή μας είναι λίγο διαφορετικές από αυτό που δείχνει η Εικόνα 5.1 για τον εξής λόγο: Για να δημιουργήσει σωστά τα διαστήματα τιμών, και να μπορεί να «ταξινομήσει» κατά μια έννοια τις διαφορετικές τιμές κλειδιού, ο TotalOrderPartitioner χρειάζεται κλειδιά που θα παίρνουν περισσότερες από R διακριτές τιμές. Για να το πετύχουμε, επιλέγουμε το κλειδί κάθε ζευγαριού που εξάγεται από τη συνάρτηση map να είναι μια συμβολοσειρά (μεταβλητή τύπου Text) που παράγεται ως εξής:

<όνομα αρχείου που απαντά το query><KENO><query_ID>
Όπου query_ID το μοναδικό id που χαρακτηρίζει κάθε επερώτημα.

Ωστόσο, τα διαστήματα που θα χρησιμοποιήσει ο TotalOrderPartitioner δημιουργούνται κατά τη φάση δειγματοληψίας της εισόδου, ενώ εμείς βρίσκαμε τα αρχεία με τα R-trees που απαντούσαν κάθε επερώτημα στη φάση Map, αφού είχε ξεκινήσει η εργασία Hadoop.

Ο γενικός αλγόριθμος που δίνει την λύση και αποτελεί την 3η στρατηγική εκτέλεσης επερωτημάτων είναι λοιπόν ο εξής: Προεπεξεργαζόμαστε τα επερωτήματα πριν ξεκινήσει η εργασία Hadoop. Η προεπεξεργασία αναλαμβάνει για κάθε επερώτημα του αρχείου κειμένου στην είσοδο να βρει ποιο αρχείο με επιμέρους R-tree το απαντάει. Η διαδικασία αυτή είναι απλή και μπορεί να γίνει με μια εφαρμογή Java χωρίς χρήση του περιβάλλοντος Hadoop. Ο αλγόριθμος για το κομμάτι αυτό είναι ακριβώς ίδιος με τον αλγόριθμο που

ακολουθούσαμε στην φάση Map των άλλων 2 στρατηγικών εκτέλεσης επερωτημάτων. Τα δεδομένα/αρχεία που χρειαζόμαστε απλά τα έχουμε αντιγράψει από το HDFS στο τοπικό σύστημα αρχείων που τρέχει η εφαρμογή. Στο τέλος η εφαρμογή παράγει ένα Sequence File με ζευγάρια της μορφής

<Text: *όνομα αρχείου που απαντά το query><KENO><query_ID , QueryQritable: το αντίστοιχο επερώτημα>*

Το αρχείο αυτό, με όνομα preprocessed_queries, φορτώνεται στην συνέχεια από το τοπικό σύστημα αρχείων στο HDFS, και δρομολογείται η δειγματοληψία και η εκτέλεση μιας εργασίας Hadoop με είσοδο το αρχείο αυτό. Η εργασία Hadoop εκτελεί τα επερωτήματα και αποθηκεύει τα αποτελέσματα στο HDFS.

Η παραπάνω στρατηγική υλοποιείται από την εφαρμογή Java με όνομα Rtree_queryPreparation, και από την κλάση με όνομα Rtree_quering_PreprocessedInput της εφαρμογής Hadoop MapReduce.

Στις δύο ενότητες που ακολουθούν θα περιγράψουμε λεπτομερέστερα την υλοποίηση του παραπάνω αλγορίθμου.

5.4.2 Περιγραφή πρώτου σταδίου της υλοποίησης: Προεπεξεργασία των επερωτημάτων

Για την υλοποίηση του 3ου αλγορίθμου εκτέλεσης επερωτημάτων, πρέπει η εργασία Hadoop που θα εκτελεστεί να λαμβάνει σαν είσοδο ένα αρχείο με προεπεξεργασμένα επερωτήματα. Η προεπεξεργασία των queries γίνεται από μια εφαρμογή Java που λαμβάνει σαν είσοδο το αρχείο κειμένου με τα επερωτήματα, καθώς και μια σειρά παραμέτρων εκτέλεσης που απεικονίζονται στον Πίνακα 5.3 που ακολουθεί.

Όνομα Παραμέτρου	**Περιγραφή**
args[0]	query_file: Το αρχείο με τα επερωτήματα εισόδου.
args[1]	rootMBR_file: Το αρχείο με τα MBRs των ριζών όλων των επιμέρους R-trees.
args[2]	treesPath: Το directory όπου βρίσκονται αποθηκευμένα τα αρχεία με τις πληροφορίες για τα R-trees.
args[3]	number_of_Rtrees: Ο αριθμός των επιμέρους R-trees που

	απαρτίζουν το Index. Θα πρέπει να ισούται με το πλήθος των αρχείων μέσα στο directory *treesPath*.
args[4]	loadTreeRootMBRs_InMemory: Εάν έχει τιμή true, για την εύρεση των αρχείων R-tree που απαντάνε κάθε επερώτημα κρατάμε τα MBRs όλων των παιδιών των ριζών στη μνήμη. Αλλιώς, τα διαβάζουμε μέσα από τα αρχεία του treesPath κάθε φορά που χρειάζεται.

Πίνακας 5.3: Οι παράμετροι που ορίζονται κατά την εκτέλεση της εφαρμογής Java για προεπεξεργασία των επερωτημάτων.

Πριν δρομολογηθεί η εκτέλεση της εφαρμογής, ο χρήστης θα πρέπει να έχει αντιγράψει τα αρχεία με τις πληροφορίες των R-trees (directory treesPath), το αρχείο με τα MBRs των ριζών όλων των επιμέρους R-trees (παράμετρος rootMBR_file) και το αρχείο με τα επερωτήματα εισόδου από το HDFS στο τοπικό σύστημα αρχείων του υπολογιστεί που θα τρέξει το πρόγραμμα. Η διαδικασία αυτή είναι πολύ γρήγορη, καθώς όπως αναλύσαμε στο κεφάλαιο 5.2.2 τα δύο πρώτα δεν ξεπερνούν σε μέγεθος τα λίγα MegaBytes.

Το πρόγραμμα αναλαμβάνει για κάθε επερώτημα του αρχείου κειμένου στην είσοδο να βρει ποιο αρχείο (ή ποια αρχεία) με επιμέρους R-tree το απαντάει, ακολουθώντας τον ίδιο ακριβώς αλγόριθμο που ακολούθησαν και οι προηγούμενοι δυο αλγόριθμοι εκτέλεσης queries κατά τη φάση Map.

Αρχικά, φορτώνονται στην δομή rootMBRs (τύπου HashMap<Text, Region>) ζεύγη με το όνομα του αρχείου που περιέχει το επιμέρους R-tree και το συνολικό MBR της ρίζας του, για κάθε R-tree. Τα στοιχεία αυτά διαβάζονται από το αρχείο τύπου SequenceFile που αντιστοιχεί στην παράμετρο rootMBR_file. Έπειτα, εξετάζεται η τιμή της παραμέτρου loadTreeRootMBRs_InMemory: εάν έχει τιμή true κρατάμε τα MBRs όλων των παιδιών των ριζών των R-trees στη μνήμη, αποθηκεύοντας στην δομή allMBRs (τύπου ArrayList<Region>) το συνολικό MBR κάθε παιδιού της ρίζας, για κάθε R-tree. Τέλος, η δομή HashMap<Text,TwoIntegerLimits> fileLimitIndex μας βοηθάει να εντοπίσουμε την θέση αρχής και τέλους των MBRs που αφορούν το κάθε επιμέρους δέντρο στο διάνυσμα allMBRs.

Έπειτα, η εφαρμογή διαβάζει ένα-ένα τα επερωτήματα από την κάθε γραμμή του αρχείου εισόδου, με τη βοήθεια της κλάσης BufferedReader. Από κάθε γραμμή εξάγει το id του χωρικού αντικειμένου καθώς και οι συντεταγμένες των 2 σημείων του ορθογωνίου του επερωτήματος απ' όπου κατασκευάζεται ένα στιγμιότυπο new_region της κλάσης spatialindex.Region που αναπαριστά το ορθογώνιο-«παράθυρο» του query. Στην συνέχεια για κάθε επερώτημα εκτελείται ο παρακάτω αλγόριθμος (παρουσιάστηκε και στην

ενότητα 5.2.2), ο οποίος εντοπίζει το αρχείο με το δέντρο (ή τα αρχεία με τα δέντρα) που εν γένει απαντά/απαντούν το query, και τα βάζει σε μια λίστα.

Αλγόριθμος *Find_Treefiles_That_Potentially_Answer_The_Query:* εντοπίζει το αρχείο με το δέντρο (ή τα αρχεία με τα δέντρα) που εν γένει απαντά/απαντούν το επερώτημα Q με παράθυρο new_region, και τα βάζει σε 2 ArrayList μαζί με το IndexIdentifier του αντίστοιχου δέντρου.

Βήμα 1. /* αρχή */

Κατασκεύασε ένα κενό ArrayList με όνομα filenames.

Κατασκεύασε ένα κενό ArrayList με όνομα headerIDs.

Βήμα 2. /* Έλεγχος αν το MBR της ρίζας του δέντρου L απαντά στο query */

While (υπάρχουν ακόμα δέντρα) {

Διάβασε από την δομή rootMBRs το συνολικό MBR της ρίζας του επόμενου δέντρου L σε μια μεταβλητή Region r.

Αν (r. intersects(new_region){

/* Έλεγχος αν κάποιο MBR παιδιού της ρίζας του δέντρου L απαντά στο query */

Αν(loadTreeRootMBRs_InMemory == *true*){

Για το δέντρο L, έλενξε με τη βοήθεια της δομής allMBRs εάν για κάποιο MBR παιδιού της ρίζας του δέντρου, έστω memory_region, ισχύει memory_region.intersects(new_region).

Εάν ναι{

Σταμάτα την αναζήτηση σ' αυτό το δέντρο.

Πρόσθεσε στο filenames το όνομα του αρχείου με το R-tree L.

Πρόσθεσε στο headerIDs το IndexIdentifier του R-tree L.

}

}

Αλλιώς Αν(loadTreeRootMBRs_InMemory == *false*){

/* πραγματοποίησε τους ελέγχους διαβάζοντας από το αρχείο που

βρίσκονται οι πληροφορίες του R-tree L */

Άνοιξε το αρχείο (τύπου Sequence File) που περιέχει τις πληροφορίες για το δέντρο L.

Για το δέντρο L, έλενξε με τη βοήθεια του αρχείου εάν για κάποιο MBR παιδιού της ρίζας του δέντρου, έστω file_region, ισχύει file_region.intersects(new_region).

Εάν ναι{

Σταμάτα την αναζήτηση σ' αυτό το δέντρο.

Πρόσθεσε στο filenames το όνομα του αρχείου με το R-tree L.

Πρόσθεσε στο headerIDs το IndexIdentifier του R-tree L.

}

}

}

Συνέχισε για το επόμενο δέντρο L=L+1.

}

Πίνακας 5.4: Αλγόριθμος που εντοπίζει τα αρχεία με τα δέντρα που εν δυνάμει απαντούν ένα επερώτημα.

Στη συνέχεια, η εφαρμογή για κάθε i-οστό μέλος της λίστας headerIDs σχηματίζει ένα στιγμιότυπο queryInfo της κλάσης QueryWritable και αποθηκεύει σ' αυτό το id του query προς απάντηση, το ορθογώνιο-«παράθυρο» του query και το IndexIdentifier του R-tree που βρίσκεται στη θέση i της λίστας headerIDs. Επίσης, συγχωνεύει σε μια μεταβλητή τύπου Text το αντίστοιχο μέλος i της λίστας filenames και το id του query διαχωρισμένα με έναν κενό χαρακτήρα (space).

Στο τέλος, για κάθε i-οστό ζεύγος που δημιούργησε με την παραπάνω διαδικασία δίνει έξοδο <κλειδί, τιμή> της μορφής

<Text: όνομα αρχείου που περιέχει το R-tree<KENO>Query_ID , QueryWritable: το επερώτημα queryInfo προς εκτέλεση>

προς το αρχείο εξόδου με όνομα preprocessed_queries και τύπο Sequence File.

Το αρχείο αυτό, ,μετά τον τερματισμό του προγράμματος, αντιγράφεται στο HDFS και δίνεται ως είσοδος προς την εργασία Hadoop που θα τρέξει για την εκτέλεση των επερωτημάτων. Να σημειωθεί, ότι επιλέξαμε το αρχείο με τα μετασχηματισμένα queries να είναι τύπου Sequence File, καθώς το Hadoop παρέχει βελτιστοποιημένη ανάγνωση των δεδομένων εισόδου εάν παρέχονται σε αρχείο Sequence File, μέσω της κλάσης SequenceFileInputFormat.

5.4.3 Περιγραφή δεύτερου σταδίου της υλοποίησης: Εκτέλεση των επερωτημάτων

Για την έναρξη της εκτέλεσης, ο χρήστης δίνει στην συνάρτηση main της εργασίας Hadoop τις παρακάτω παραμέτρους που εμφανίζονται στον Πίνακα 5.5 .

Όνομα Παραμέτρου	**Περιγραφή**
args[0]	InputPath: Το αρχείο με τα προεπεξεργασμένα επερωτήματα εισόδου. Σε περίπτωση που δωθεί directory, ως είσοδος λαμβάνονται όλα τα αρχεία του directory.
args[1]	OutputPath: Το directory όπου θα αποθηκευτούν τα αρχεία εξόδου της εργασίας Hadoop με τα αποτελέσματα των queries.
args[2]	numberOfReducers: Αριθμός των διεργασιών Reduce.
args[3]	Treefiles.directory: Το directory όπου βρίσκονται αποθηκευμένα τα αρχεία με τα R-trees. Η τιμή του φορτώνεται στην παράμετρο *Rtree_test_epikalypsi.Treefiles.directory* του αρχείου Configuration της εργασίας, ώστε να μπορεί να ανακτηθεί από όλες τις διεργασίες (Map ή Reduce).
args[4]	Αριθμός δειγμάτων που θα ληφθούν κατά τη διαδικασία δειγματοληψίας της εισόδου.

Πίνακας 5.5: Παράμετροι που δίνονται για την εκτέλεση της εργασίας Hadoop.

Στην συνέχεια παρουσιάζονται τα βασικά βήματα εκτέλεσης.

1. Δειγματοληψία και διάβασμα Εισόδου

Η διαδικασία της δειγματοληψίας γίνεται πριν την έναρξη της κύριας εργασίας Hadoop, χρησιμοποιώντας την κλάση new_Partitioner.InputSampler.

Η αρχικοποίηση του sampler στον κώδικά μας γίνεται ως εξής:

InputSampler.Sampler < Text, QueryWritable > sampler =

new InputSampler.RandomSampler <Text,QueryWritable >(0.1, Integer.parseInt(args[4]), 20);

Ο sampler επιλέγει με τυχαίο τρόπο (κάθε record έχει πιθανότητα 0.1% να επιλεγεί) τον αριθμό δειγμάτων που θα ορίσει ο χρήστης ως παράμετρο, χρησιμοποιώντας το πολύ 20 input splits από την είσοδο τα οποία διαβάζει με την βοήθεια της κλάσης SequenceFileInputFormat. Στη συνέχεια παράγει το αρχείο με όνομα _partitions.lst που περιέχει τα όρια των (περίπου ίσων) R διαμερίσεων. Τέλος, χρησιμοποιώντας τον μηχανισμό Distributed Cache το αρχείο αυτό αντιγράφεται τοπικά σε όλους τους κόμβους του cluster, ώστε να είναι διαθέσιμο κατά την φάση partitioning.

Μετά την διαδικασία δειγματοληψίας, αρχίζει η εκτέλεση της κύριας εργασίας Hadoop και διαβάζεται η είσοδος χρησιμοποιώντας την κλάση SequenceFileInputFormat. Η μορφή των ζευγαριών <κλειδί, τιμή> που θα τροφοδοτηθούν στην φάση Map είναι η ακόλουθη:

<Text: όνομα αρχείου που περιέχει το R-tree<KENO>Query_ID , QueryWritable: το επερώτημα queryInfo προς εκτέλεση>

2. Φάση Map

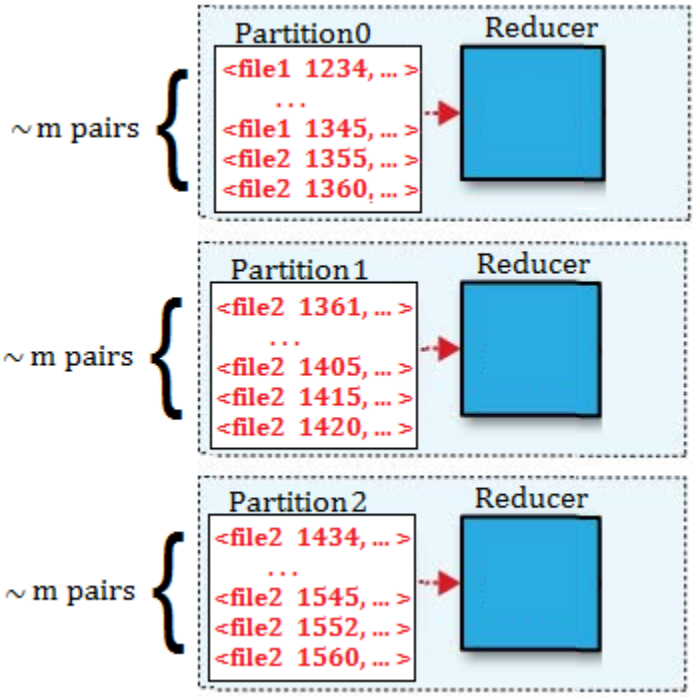
Η συνάρτηση map στην υλοποίησή μας, δέχεται είσοδο ένα ζεύγος <κλειδί, τιμή> της μορφής που αναφέραμε, και απλά το δρομολογεί αυτούσιο στην έξοδο. Έτσι, η φάση Map δίνει έξοδο ζευγάρια <κλειδί, τιμή> της μορφής

<Text: όνομα αρχείου που περιέχει το R-tree<KENO>Query_ID , QueryWritable: το επερώτημα queryInfo προς εκτέλεση>

3. Φάση Partitioning

Όπως αναφέραμε παραπάνω, ορίζουμε ως κλάση Partitioner την κλάση new_partitioner.TotalOrderPartitioner. Η συνάρτηση TotalOrderPartitioner. getPartition δέχεται ως είσοδο ένα-ένα ζευγάρι που έχει παραχθεί από κάθε Mapper και επιστρέφει για κάθε ζευγάρι έναν ακέραιο αριθμό από 0 έως R-1, που

αντιστοιχεί στον αριθμό της διαμέρισης που θα οδηγηθεί το ζευγάρι αυτό. Ο διαμερισμός γίνεται ως εξής: ανακτώνται από το αρχείο όνομα _partitions.lst τα όρια των (περίπου ίσων) R διαμερίσεων, και ανάλογα σε ποιο εύρος ανήκει η τιμή κλειδιού (*Text: όνομα αρχείου που περιέχει το R-tree<KENO>Query_ID*), το ζεύγος οδηγείται στο αντίστοιχο partition.



Εικόνα 5.2:Η διαδικασία Partitioning.

4. Φάση Reduce και εγγραφή αποτελεσμάτων

Κάθε Reducer αναλαμβάνει ένα partition, όπου περιέχει ζευγάρια με πολλές διαφορετικές τιμές κλειδιών, λόγω της ύπαρξης του query_ID στο δεύτερο μέρος της συμβολοσειράς του κάθε κλειδιού. Για κάθε διαφορετική τιμή κλειδιού, η κλάση Grouping Comparator ομαδοποιεί τα ζευγάρια της διαμέρισης που έχουν την ίδια τιμή κλειδιού. Έτσι δημιουργούνται ζευγάρια της μορφής <κλειδί, λίστα τιμών>. Για κάθε τέτοιο ζεύγος δρομολογείται και μια νέα κλήση της συνάρτησης reduce(). Προφανώς, αυτό είναι κάτι που δεν θέλουμε να γίνει, καθώς θα κάθε Reducer θα προσπαθούσε να φτιάξει ένα δέντρο για κάθε διαφορετική τιμή κλειδιού! Επομένως, πρέπει να ορίσουμε την δικιά μας κλάση Grouping Comparator, που θα ομαδοποιεί όλα τα ζευγάρια της διαμέρισης που αναφέρονται στο ίδιο όνομα αρχείου R-tree σε ένα μόνο ζεύγος <κλειδί, λίστα τιμών>.

Ορίζουμε λοιπόν σαν κλάση Grouping Comparator την δική μας static κλάση EqualFilenameGroupingComparator, η οποία για κάθε δύο κλειδιά που συγκρίνει, τα βρίσκει όμοια αν έχουν ως πρώτο συνθετικό (πριν τον κενό χαρακτήρα) το ίδιο όνομα αρχείου R-tree! Με αυτό τον τρόπο, δημιουργείται ένα ζεύγος <κλειδί, τιμή> της μορφής

< Text: όνομα αρχείου που περιέχει το R-tree<KENO>ένα τυχαίο Query_ID, λίστα από(QueryWritable: όλα τα επερωτήματα queryInfo της διαμέρισης που απαντώνται εν δυνάμει από το συγκεκριμένο R-tree)>

Κάθε ζεύγος της παραπάνω μορφής που ανήκει στη διαμέριση είναι είσοδος για μια νέα κλήση της συνάρτησης reduce().

Η συνάρτηση reduce() ξεχωρίζει από την τιμή του κλειδιού του ζεύγους εισόδου της το πρώτο μέρος που αναφέρεται στο όνομα του αρχείου που περιέχει αποθηκευμένο το R-tree. Στη συνέχεια, η υλοποίηση είναι ίδια με την αντίστοιχη συνάρτηση reduce της 1ης και 2ης στρατηγικής που περιγράψαμε στην ενότητα 5.2.2 και 5.2.3 . Συνοπτικά υπενθυμίζουμε: Αρχικοποιείται αρχικά ο HDFSStorageManager και ο RandomEvictionsBuffer για την διαχείριση των λειτουργιών I/O από το HDFS και τη δημιουργία buffer ενδιάμεσης αποθήκευσης. Σειρά έχει η αρχικοποίηση του R-tree ώστε να εκτελεστούν τα επερωτήματα με τη βοήθειά του. Για κάθε επερώτημα της λίστας τιμών εισόδου φτιάχνεται ένα στιγμιότυπο r της κλάσης Region και εκτελείται το query με την κλήση της συνάρτησης Rtree.intersectionQuery(r, vis). Η παράμετρος vis είναι στιγμιότυπο της εσωτερικής κλάσης MyVisitor, που αναλαμβάνει την εγγραφή των αποτελεσμάτων κάθε query στο αρχείο εξόδου. Μέσω της συνάρτησης context.write(κλειδί, τιμή) η MyVisitor μπορεί να εκτυπώνει τα αποτελέσματα του κάθε query στο αρχείο εξόδου της εργασίας Hadoop, εξάγοντας ένα ζεύγος ανά αποτέλεσμα στη μορφή <κλειδί, τιμή>:

< VLongWritable: query_id , VLongWritable :id χωρικού αντικειμένου που απαντά στο query >

6. Πειραματική Αξιολόγηση

6.1 Μεθοδολογία των πειραμάτων και δεδομένα εισόδου

Στο κεφάλαιο αυτό θα παρουσιάσουμε τα αποτελέσματα των πειραμάτων που πραγματοποιήσαμε για την αξιολόγηση της υλοποίησης των διαφόρων τμημάτων της εργασίας.

Τα πειράματα έγιναν χρησιμοποιώντας τον cluster που έχει εγκατασταθεί στο Εργαστήριο Softnet του τμήματος Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης. Ο cluster απαρτίζεται από 12 διασυνδεδεμένους κόμβους, σε αρχιτεκτονική Master/Slave.

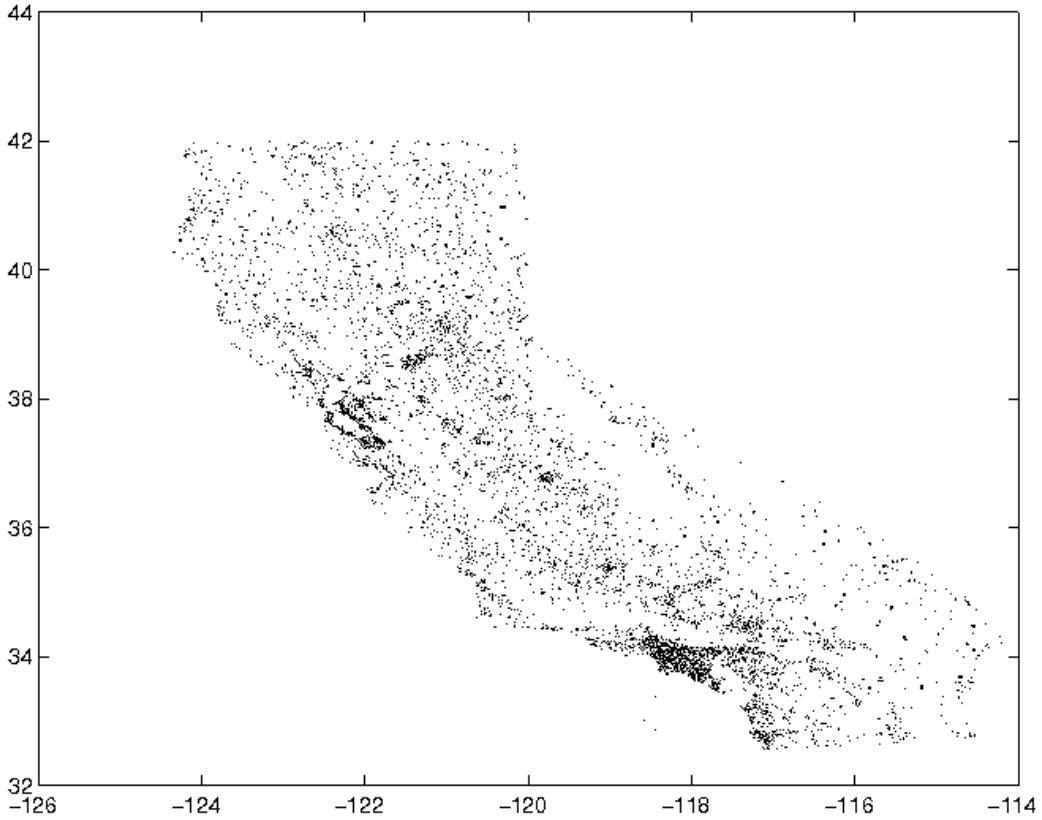
Στον cluster υπήρχε κατά την ανάπτυξη της εργασίας αυτής εγκατεστημένη η έκδοση Hadoop 0.20.2 ενώ αργότερα εγκαταστάθηκε η νεώτερη stable έκδοση Hadoop 0.20.203 . Οι υλοποιήσεις των διαφόρων αλγορίθμων της εργασίας αυτής εκτελέστηκαν επιτυχώς και στις 2 εκδόσεις, οι πειραματικές μετρήσεις που παρουσιάζονται ωστόσο αντιστοιχούν στις εκτελέσεις των πειραμάτων χρησιμοποιώντας την νεώτερη έκδοση Hadoop 0.20.203 .

Η καταχώρηση εργασιών προς εκτέλεση στον cluster γινόταν μέσω απομακρυσμένης σύνδεσης SSH με τον server, όπου και μεταφέραμε τις εντολές για την εκτέλεση των εργασιών Hadoop. Τα πακέτα .jar που περιείχαν τον κώδικα της εργασίας προς εκτέλεση βρίσκονταν στον τοπικό δίσκο του server, ενώ τα αρχεία εισόδου (και άλλα απαραίτητα αρχεία) βρίσκονταν αποθηκευμένα στο HDFS.

Τα πειράματά μας έγιναν με τρόπο τέτοιο, ώστε να μετρηθεί η απόδοση της κάθε υλοποίησης ως προς το χρόνο εκτέλεσης, την εγκυρότητα των αποτελεσμάτων εξόδου και την επεκτασιμότητα (παράγοντες scale-up και size-up). Η απόδοση στο πεδίο του χρόνου κάθε φορά εξαρτάται από διάφορες παραμέτρους, ανάμεσά τους το μέγεθος της εισόδου και ο αριθμός των κόμβων επεξεργασίας που τρέχουν παράλληλα την εφαρμογή. Πέρα από αυτά, η απόδοση επηρεάζεται πολύ από εγγενή χαρακτηριστικά του εκάστοτε αλγορίθμου της υλοποίησης που εκτελείται, τα οποία θα αναλύσουμε παρακάτω εξηγώντας την συμπεριφορά της υλοποίησης στα πειράματά μας.

Τα δεδομένα εισόδου που χρησιμοποιήσαμε είχαν ως βάση ένα πραγματικό σύνολο δεδομένων, το σύνολο Tiger/Line που παρέχεται από την MAF/TIGER® (Master Address File/Topologically Integrated Geographic Encoding and Referencing) βάση δεδομένων του U.S. Census Bureau [33]. Το αρχικό σύνολο δεδομένων περιέχει 2.249.727 μικρά

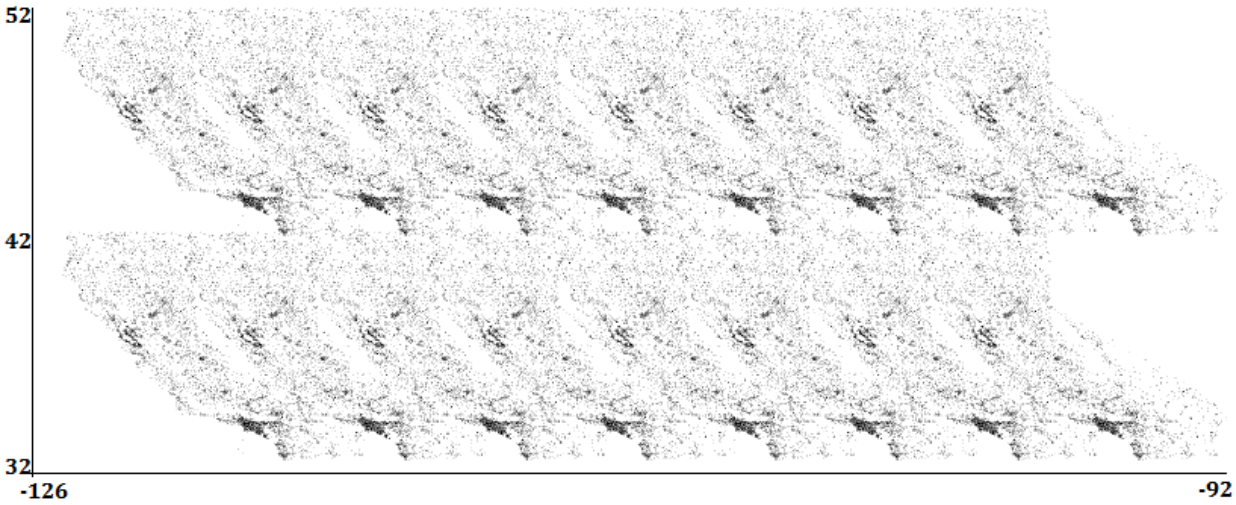
ορθογώνια MBRs δρόμων (polylines) της California. Στην Εικόνα 6.1 φαίνεται το αρχικό σύνολο δεδομένων.



Εικόνα 6.1: Απεικόνιση των MBRs του συνόλου πραγματικών δεδομένων CAR.

Το αρχικό σύνολο είχε μέγεθος σχετικά μικρό, και για το λόγο αυτό το μεγαλώσαμε 100, ή και διακόσιες φορές με την γνωστή μέθοδο μετατόπισης ενός στοιχείου μιας γραφικής παράστασης (προσθέτοντας/αφαιρώνταςς έναν σταθερό αριθμό στις κατάλληλες συντεταγμένες). Στα αρχεία κειμένου που δημιουργήθηκαν προσθέσαμε επίσης και ένα μοναδικό id για κάθε ορθογώνιο. Με τον τρόπο αυτό δημιουργήσαμε ένα σύνολο χωρικών δεδομένων μεγέθους 5GB (μετακινώντας κάθε ορθογώνιο του αρχικού συνόλου 50 φορές κατά σταθερές μονάδες), ένα με μέγεθος 7.5 GB (μετακινώντας κάθε σημείο του αρχικού συνόλου 75 φορές κατά σταθερές μονάδες) και ένα μεγέθους 10GB (μετακινώντας κάθε σημείο του αρχικού συνόλου 100 φορές κατά σταθερές μονάδες). Τέλος, ένα σύνολο χωρικών δεδομένων μεγέθους 20GB προέκυψε από αντιγραφή κάθε σημείου του συνόλου των 10GB μια φορά και μετακίνησή του. Καθώς δεν είναι εφικτή η απεικόνιση όλου του Dataset, η Εικόνα 6.2 που ακολουθεί δείχνει πως γίνεται για παράδειγμα η μετακίνηση και

αντιγραφή των σημείων του αρχικού dataset κατά 8 φορές προς τα δεξιά και μια φορά πάνω.



Εικόνα 6.2: Απεικόνιση ενός συνόλου δεδομένων που θα προέκυπτε από μετακίνηση του αρχικού Dataset κατά 8 φορές δεξιά και 1 φορά πάνω.

Όπως είναι φανερό, τα Dataset που προέκυψαν δεν είναι ομοιόμορφα, αλλά περιέχουν και λίγο πιο πυκνές ή πιο αραιές περιοχές. Η δομή του υπόλοιπου κεφαλαίου είναι η ακόλουθη: Στην ενότητα 6.2 παρουσιάζουμε τα πειράματα για τον αλγόριθμο κατασκευής του Packed Hilbert R-tree και στην ενότητα 6.3 βρίσκονται τα πειραματικά αποτελέσματα για τους 3 αλγορίθμους εκτέλεσης χωρικών επερωτημάτων με τη βοήθεια του R-tree.

6.2 Πειραματική αξιολόγηση του αλγορίθμου κατασκευής Packed Hilbert R-tree στο HDFS

Αρχικά η υλοποίησή μας εκτελέστηκε με χρήση του Pseudodistributed Mode που παρέχει το Hadoop, δηλαδή σε έναν απλό υπολογιστή όπου είχαμε εγκαταστήσει έναν «Single-Node Hadoop cluster». Για την ακρίβεια, προκειμένου να ελέγξουμε εάν ο αλγόριθμός μας έφτιαχνε άρτια R-tree, χρησιμοποιήσαμε αρχικά για τα πειράματα σε Pseudodistributed Mode τον DiskStorageManager της βιβλιοθήκης spatial index. Σε πρώτη φάση, φτιάξαμε ένα Packed Hilbert R-tree στο δίσκο με μικρή είσοδο, που αντιστοιχούσε στο πρωτότυπο CAR dataset που περιγράψαμε παραπάνω (αφού το είχαμε εμπλουτίσει με ids για κάθε

ορθογώνιο χωρικό αντικείμενο). Το σύνολο δεδομένων της εισόδου αντιστοιχούσε σε 2.249.727 ορθογώνια, από τα οποία κτίσαμε το R-tree**.** Έτσι, μπορέσαμε να εκτελέσουμε κάποια πρώιμα επερωτήματα (με τη βοήθεια συναρτήσεων της βιβλιοθήκης spatial index) στο νέο μας δέντρο. Τα αποτελέσματα αυτών των επερωτημάτων τα συγκρίναμε με αυτά που έβγαζαν άλλες παραλλαγές R-tree που είχαμε φτιάξει: είτε χρησιμοποιώντας την παραλλαγή packed Hilbert R-tree στο δίσκο που είχαμε φτιάξει εμείς, είτε χρησιμοποιώντας την παραλλαγή R*-tree που παρείχε η spatial index. Αφού βεβαιωθήκαμε ότι ο αλγόριθμός μας δεικτοδοτούσε σωστά τα χωρικά δεδομένα, μπορούσαμε να προχωρήσουμε στην προσαρμογή του αλγορίθμου στο περιβάλλον HDFS (όπως περιγράφηκε στο κεφάλαιο 4) και την αξιολόγηση της απόδοσής του.

Η νέα υλοποίηση εκτελέστηκε και στον cluster του εργαστηρίου Softnet σε πραγματικό κατανεμημένο περιβάλλον. Να σημειωθεί πληροφοριακά, ότι το σύνολο δεδομένων 10GB (που χρησιμοποιήθηκε στα περισσότερα πειράματα για την κατασκευή του R-tree, εκτός εάν δηλώνεται διαφορετικά) περιέχει 227.222.427 χωρικά αντικείμενα (και τα αντίστοιχα ids τους).

Το R-tree κατασκευάστηκε με τις παρακάτω προδιαγραφές:

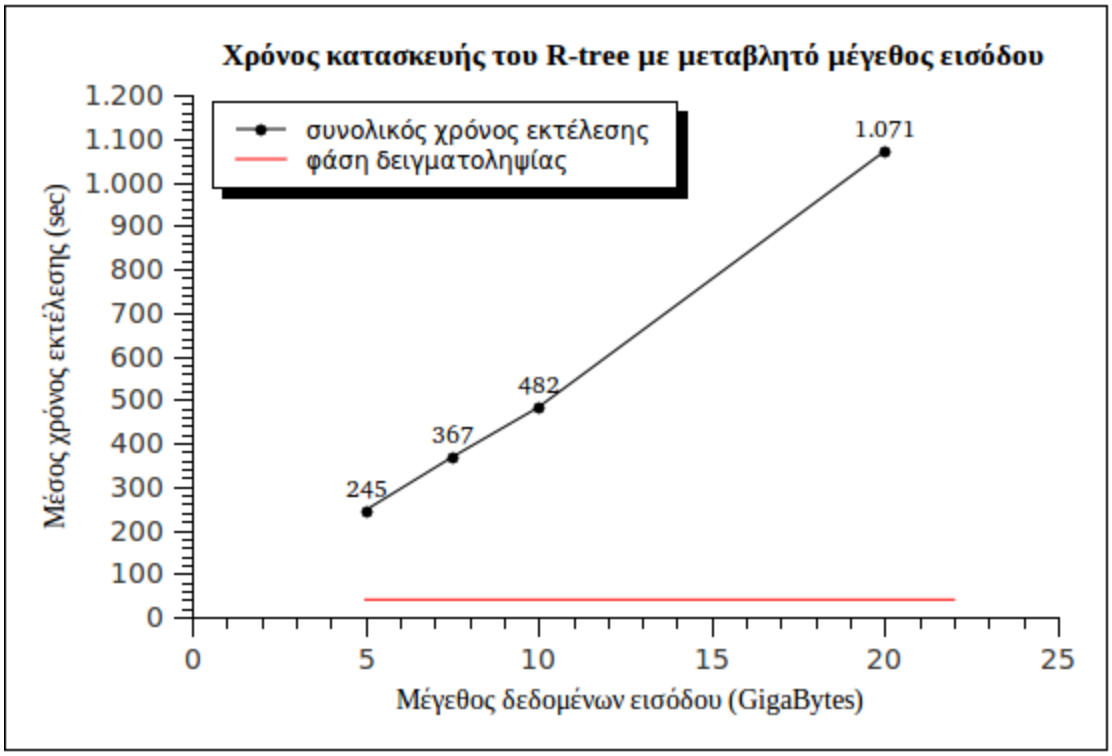
Όνομα Παραμέτρου	**Τύπος**	**Περιγραφή**
Αριθμός Δειγμάτων για την φάση δειγματοληψίας	Integer	Πραγματοποιήσαμε πειράματα χρησιμοποιώντας ως δείγματα το 1-3% της συνολικής εισόδου. Ο RandomSampler και ο TotalOrderPartitioner λειτουργούν πολύ καλά και με το 1% της εισόδου ως δείγμα (και στην περίπτωσή μας που έχουμε όχι πολύ ομοιόμορφα δεδομένα), καταλήγοντας σε ίσου περίπου μεγέθους διαμερίσεις. Ορίσαμε τέλος, μέγιστο αριθμό input splits για δειγματοληψία το 20 για μείωση του κόστους.
Αριθμός Reduce Tasks	Integer	Για τα επόμενα πειράματα θεωρούμε αριθμό Reducers=10, εκτός εάν πρόκειται γι πείραμα με αύξηση του αριθμού Reducers, όπου ο αριθμός τους φαίνεται στην γραφική παράσταση.

PageSize	Integer	Το μέγεθος σελίδας αποθήκευσης. Δίνουμε τιμή ίση με την παράμετρο dfs.client-write-packet-size του Hadoop, με default τιμή 64 Kilobytes. Αυτό γίνεται με σκοπό να ελαχιστοποιήσουμε την μεταφορά άχρηστων bytes στο δίκτυο του cluster. Για περισσότερες λεπτομέρειες, δείτε στο Παράρτημα Α τον μηχανισμό ανάγνωσης/εγγραφής του HDFS.
Διάσταση δεδομένων	Integer	Επιλέξαμε τιμή 2. Μπορεί να πάρει και τιμές για περισσότερες διαστάσεις.
IndexCapacity = LeafCapacity	Integer	Η συνολική χωρητικότητα των εσωτερικών κόμβων του δέντρου δεν δίνεται από τον χρήστη, αλλά καθορίζεται από το μέγεθος που έχει η παράμετρος PageSize του HDFSStorageManager, καθώς θέλουμε ένας κόμβος να καταλαμβάνει ακριβώς μια σελίδα. Υπολογίζεται βάσει τύπου. Για PageSize= dfs.client-write-packet-size=64 KiloBytes, έχουμε IndexCapacity=1489.

Πίνακας 6.1: Προδιαγραφές κατασκευής του R-tree.

Κατά την εκτέλεση των πειραμάτων, επιχειρήσαμε να μειώσουμε τον αριθμό των διεργασιών Map που δημιουργούνταν, αυξάνοντας το μέγεθος κάθε Input Split μέσω 2 ειδικών παραμέτρων που προσφέρει το Hadoop, την mapred.min.split.size και mapred.max.split.size. Προσαρμόσαμε το μέγεθος του input split έτσι ώστε να έχουμε 40 περίπου Map Tasks συνολικά (λαμβάνοντας υπόψιν μας την ικανότητα του cluster μας να εκτελεί 48 tasks παράλληλα). Απ΄ότι είδαμε, η αλλαγή δεν έφερε τα αναμενόμενα αποτελέσματα, καθώς για μεγάλες εισόδους τα κάθε Input Split κατέληγε πολύ μεγάλο, πράγμα που έκανε την επαναδρομολόγηση αποτυχημένων ή καθυστερημένων tasks συχνή και δαπανηρή. Έτσι, επαναφέραμε τις default ρυθμίσεις.

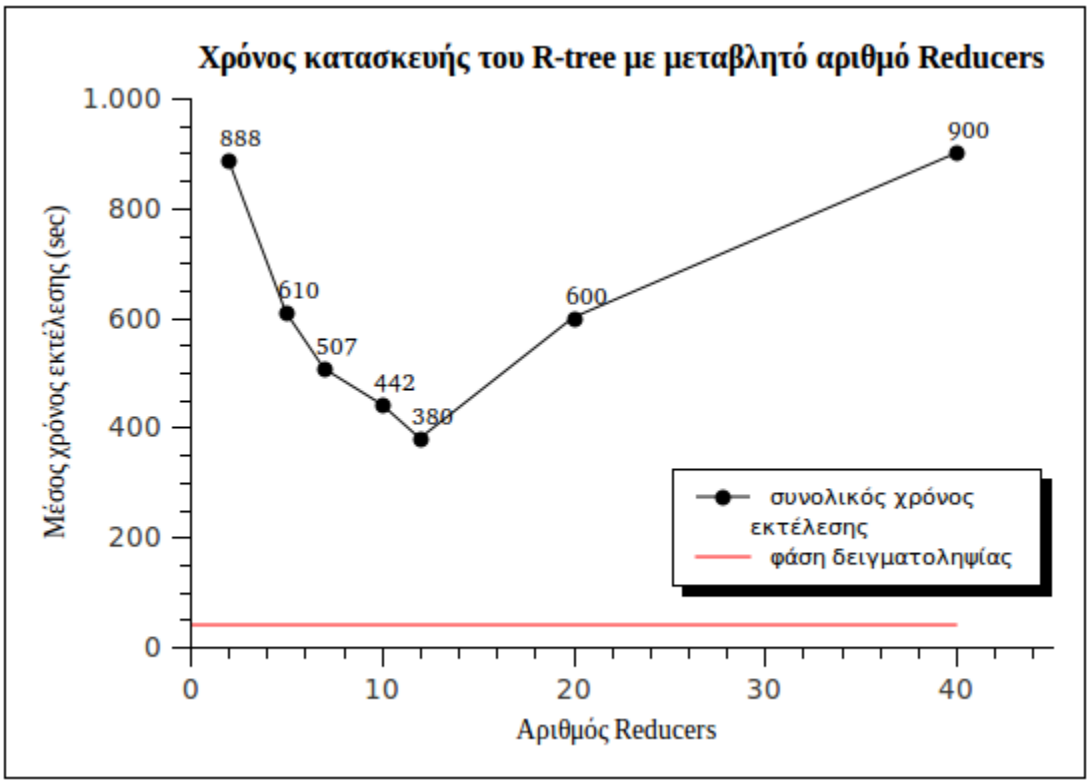
Η Εικόνα 6.3 που ακολουθεί απεικονίζει τα αποτελέσματα εκτέλεσης του αλγορίθμου κατασκευής ενός Packed Hilbert R-tree στο HDFS χρησιμοποιώντας 10 Reduce tasks, για μεταβλητό μέγεθος εισόδου (παράγοντας size-up).



Εικόνα 6.3: Εκτέλεση του αλγορίθμου κατασκευής ενός Packed Hilbert R-tree στο HDFS χρησιμοποιώντας 10 Reduce tasks, για μεταβλητό μέγεθος εισόδου.

Παρατηρούμε ότι ο αλγόριθμος έχει πολύ καλή επεκτασιμότητα ως προς το μέγεθος των χωρικών δεδομένων εισόδου, ακόμα και για μη-ομοιόμορφα πραγματικά χωρικά δεδομένα εισόδου. Ωστόσο, ο χρόνος εκτέλεσης όταν διπλασιάζουμε το μέγεθος εισόδου υπερ-διπλασιάζεται Αυτό συμβαίνει γιατί η μεγαλύτερη είσοδος προκαλεί μεγαλύτερο αριθμό διεργασιών Map που την διαβάζουν. Οι mappers αυτοί μπορούν να εκτελεστούν παράλληλα σε ομάδες των 48 (όσοι και οι πυρήνες επεξεργασίας των μηχανημάτων του cluster), ενώ εάν υπάρχουν περισσότερες από μία ομάδες των 48 tasks, οι επιπλέον mappers εκτελούνται με τη σειρά τους όταν ελευθερωθεί κάποια επεξεργαστική μονάδα. Επιπλέον, για μεγαλύτερες εισόδους η φάση αντιγραφής (Shuffle) και μετά ταξινόμησης της εξόδου των Mappers προς τους Reducers παίρνει χρόνο που δεν αυξάνεται γραμμικά (λόγω φόρτωσης του δικτύου και λόγω τακτικής εξωτερικής ταξινόμισης που απαιτεί πρόσβαση στο δίσκο).

Τέλος, βλέπουμε ότι η διάρκεια της φάσης δειγματοληψίας (περίπου 35 sec) παρέμενε κατά μέσο όρο σταθερή, ανεξάρτητα από το μέγεθος της εισόδου. Αυτό είναι αποτέλεσμα του περιορισμού που θέσαμε στον Sampler να διαλέγει με τυχαίο τρόπο έως 20 input splits από την είσοδο και να λαμβάνει τα δείγματα, πράγμα που μειώνει το κόστος της δειγματοληψίας διατηρώντας όμως πολύ καλά αποτελέσματα.



Εικόνα 6.4:Χρόνος εκτέλεσης του αλγορίθμου κατασκευής του R-tree με 10GB είσοδο, καθώς αυξάνεται ο αριθμός των διεργασιών Reduce.

Στην Εικόνα 6.4 παρατηρούμε την συμπεριφορά του αλγορίθμου κατασκευής του R-tree καθώς αυξάνεται ο αριθμός των διεργασιών Reduce. Για το πείραμα χρησιμοποιήθηκε ως είσοδος σύνολο δεδομένων 10GB.

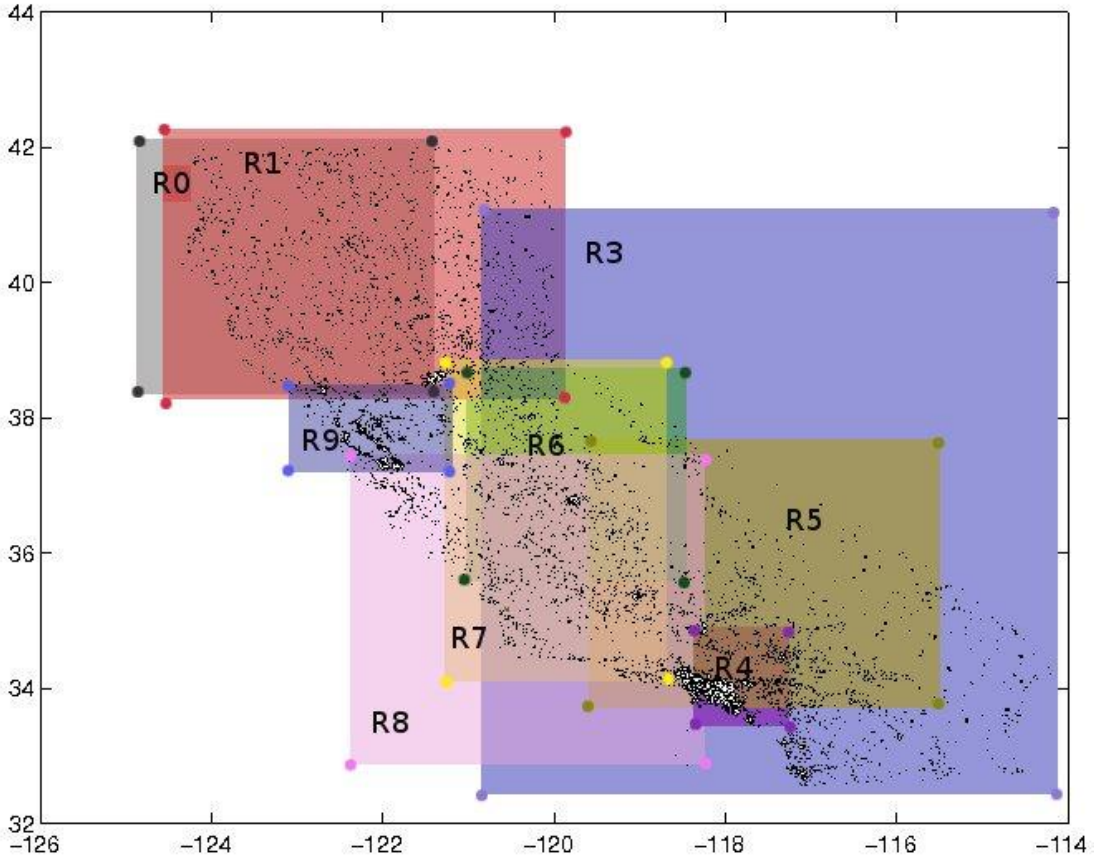
Βλέπουμε ότι ο χρόνος εκτέλεσης της εργασίας Hadoop (σ' αυτόν δεν συμπεριλαμβάνεται η φάση δειγματοληψίας) μειώνεται όσο αυξάνουμε τους Reducers, μέχρι να φτάσουμε σε αριθμό διεργασιών Reduce τον αριθμό 12, που ισούται με το αριθμό των κόμβων του cluster. Η μείωση αυτή είναι αναμενόμενη, αφού τα χωρικά δεδομένα διασπώνται σε περισσότερα μέρη (όσα και ο αριθμός των Reducers), άρα κάθε φάση Reduce έχει μικρότερο αριθμό δεδομένων για επεξεργασία καθώς αυξάνονται οι Reducers. Η μείωση δε αναμέναμε να μην είναι γραμμική, καθώς οι φάσεις Map, Partitioning και με μικρή διαφορά η Shuffle παίρνουν περίπου ίδιο χρόνο ανεξάρτητα από τον αριθμό των Reducers. Επιπλέον, η αρχικοποίηση περισσότερων tasks είναι σχετικά χρονοβόρα, πράγμα που πρέπει να αντισταθμιστεί με το κέρδος που έχουμε (αυτό θα φανεί σε επόμενα πειράματα, όπου έχουμε μικρό μέγεθος εισόδου).

Από εκεί και έπειτα όσο οι Reducers αυξάνονται για τιμές πάνω από 12, η απόδοση του αλγορίθμου χειροτερεύει δραματικά. Αυτό συμβαίνει λόγω των πολλαπλών Reducers που τρέχουν στον ίδιο κόμβο, καθώς ο αριθμός των διεργασιών Reduce είναι μεγαλύτερος από τον αριθμό των διαθέσιμων υπολογιστών του cluster. Οι διεργασίες Reduce προσπαθούν να φτιάξουν τα δέντρα και απαιτούν συχνές προσβάσεις στον τοπικό δίσκο του ίδιου υπολογιστή αλλά και άλλων υπολογιστών, ώστε να αποθηκεύσουν τα νέα δεδομένα του δέντρου στο HDFS (λόγω του παράγοντα replication 3, που αντιγράφει 3 φορές κάθε block δεδομένων για αξιοπιστία). Αυτό προκαλεί «άτακτες» μετακινήσεις της κεφαλής του κάθε δίσκου σε διάφορα σημεία του, αλλά και συχνές καθυστερήσεις στην εγγραφή δεδομένων λόγω write-lock , φαινόμενα γνωστά με την ορολογία *Thrasing*. Αυτό είναι ένα εγγενές χαρακτηριστικό των περιπτώσεων κοινής χρήσης των δίσκων και δεν οφείλεται στον σχεδιασμό του αλγορίθμου. Σε clusters με πολλούς κόμβους επεξεργασίας, ένα τέτοιο φαινόμενο θα εμφανιζόταν για πολύ μεγάλες τιμές αριθμού Reducers, κάτι που δεν θα επηρέαζε αρνητικά την εκτέλεση. Εάν σκεφτούμε ότι το Hadoop υποστηρίζει αξιόπιστη εκτέλεση εργασιών σε πολύ μεγάλα clusters κοινών υπολογιστών (με μικρό κόστος κατασκευής των clusters), είναι κοινό φαινόμενο οι εργασίες Hadoop να εκτελούνται σε clusters με εκατοντάδες υπολογιστές. Σε τέτοιους clusters λοιπόν, η κατασκευή R-trees με λίγες εκατοντάδες Reducers, θα έφερνε εξαιρετικά αποτελέσματα από άποψη χρονικής απόδοσης, καταφέρνοντας να διαχειριστεί μεγάλα σύνολα χωρικών δεδομένων.

6.3 Πειραματική αξιολόγηση των αλγορίθμων εκτέλεσης επερωτημάτων

Αρχικά, φτιάξαμε όπως προείπαμε ένα Packed Hilbert R-tree με μικρή είσοδο, που αντιστοιχούσε στο πρωτότυπο CAR dataset που περιγράψαμε παραπάνω (αφού το είχαμε εμπλουτίσει με ids για κάθε ορθογώνιο χωρικό αντικείμενο). Το σύνολο δεδομένων της εισόδου αντιστοιχούσε σε 2.249.727 ορθογώνια, από τα οποία κτίσαμε το R-tree.

Ανακτώντας τα MBRs των ριζών όλων των επιμέρους R-trees που δημιουργήθηκαν, για διάφορετικούς αριθμούς Reducers, προσπαθήσουμε να τα απεικονίσουμε ώστε να έχουμε μια εικόνα σχετικά με την επικάλυψη των επιμέρους δέντρων που απαρτίζουν το Index. Στην Εικόνα 6.5 φαίνονται τα MBRs των ριζών κάθε επιμέρους R-tree, για αριθμό Reducers 10. Φαίνεται ότι οι ρίζες των δέντρων έχουν μεγάλη επικάλυψη, πράγμα που δικαιολογείται και από την «μη ομοιόμορφη» φύση των δεδομένων που απαρτίζουν το συνολικό Index.



Εικόνα 6.5: Τα MBRs των ριζών κάθε επιμέρους R-tree, για αριθμό Reducers 10 και είσοδο το αρχικό μικρό dataset.

Αυτός ήταν ο λόγος που μας οδήγησε να εξετάσουμε τον βαθμό επικάλυψης των MBRs των άμεσων κόμβων-παιδιών της ρίζας. Για να το κάνουμε αυτό, ουσιαστικά εκτελέσαμε επερωτήματα με τα ίδια ακριβώς ορθογώνια που απάρτιζαν το R-tree, ώστε να δούμε από πόσα δέντρα «επικαλύπτονται» κρίνοντας αποκλειστικά από τα MBRs των ριζών και των άμεσων παιδιών τους. Τα αποτελέσματα ήταν πολύ καλά, όπως φαίνεται και στην Εικόνα 6.6. Πάνω από 80% των χωρικών δεδομένων επικαλύπτονταν (έστω και μερικώς) μόνο από 1 δέντρο (βάσει των πρώτων 2 επιπέδων του), ενώ το ποσοστό των αντικειμένων που επικαλύπτονταν πάνω από 3 δέντρα ήταν μηδενικό.

Στην συνέχεια, εκτελέσαμε intersection επερωτήματα για να αξιολογήσουμε την απόδοση της υλοποίησής μας. Στην ενότητα που ακολουθεί θα σχολιάσουμε τα αποτελέσματα της εκτέλεσης μιας δέσμης 22.497 επερωτημάτων με χρήση των 3 μεθόδων εκτέλεσης χωρικών επερωτημάτων που παρουσιάσαμε στο κεφάλαιο 5. Η εκτέλεση χρησιμοποίησε ένα R-tree του HDFS με 10 επιμέρους αρχεία (είχε φτιαχτεί δηλαδή από 10 Reducers) και με 10GB χωρικών δεδομένων δεικτοδοτούμενα απ’ αυτό. Τα επερωτήματα intersection είχαν παράθυρο αναζήτησης ίδιας τάξης μεγέθους περίπου με τα ορθογώνια που υπήρχαν

στο R-tree και επέστρεφαν το καθένα 10-20 δεδομένα ως αποτέλεσμα (είναι δηλαδή επερωτήματα με υψηλή επιλεκτικότητα -High selectivity).

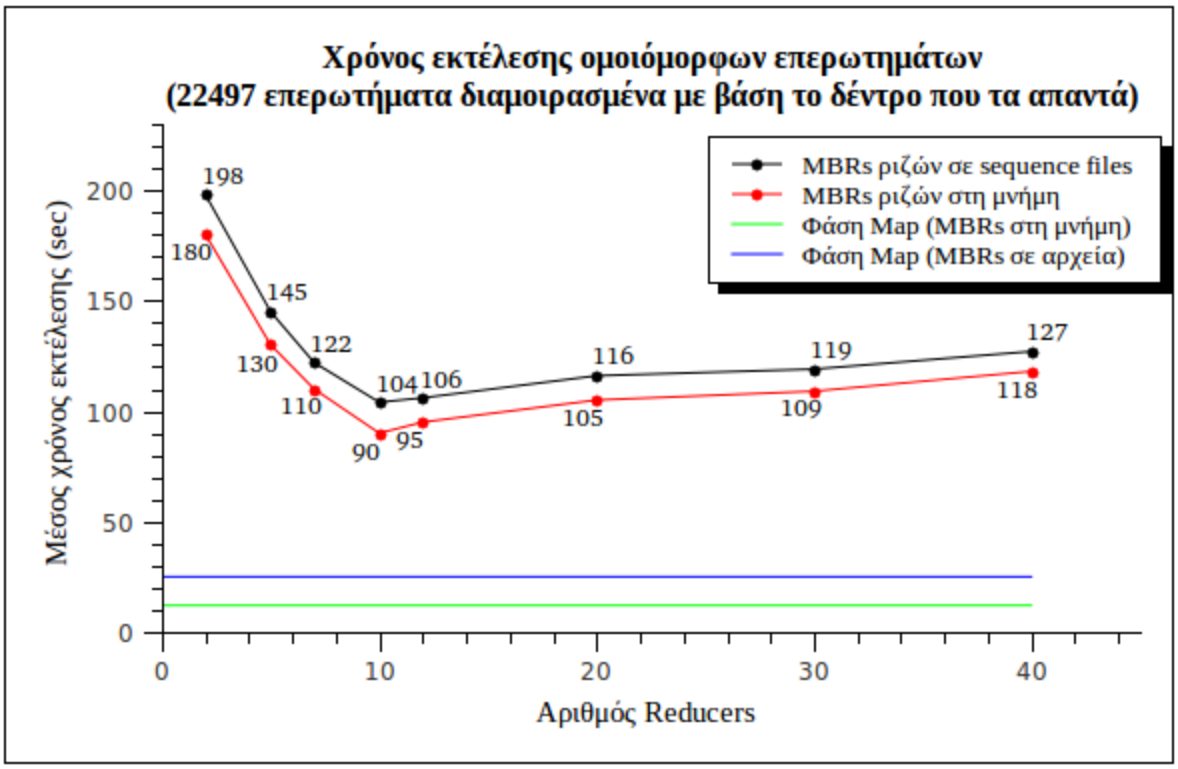
Είναι σημαντικό να πούμε, ότι κατά την διάρκεια των πάρα πολλών πειραμάτων, είχαμε μηδενικές αστοχίες. Αυτό αποδεικνύει την αξιοπιστία του Hadoop και την ικανότητά του να ανιχνεύει αποτυχημένες ή καθυστερημένες διεργασίες και να τις επαναδρομολογεί σε διαφορετικό κόμβο του cluster.

6.3.1 Εκτέλεση μιας δέσμης «ομοιόμορφων», ως προς τα R-trees, επερωτημάτων

Αρχικά, εκτελέσαμε επερωτήματα που στόχευαν όσο το δυνατόν πιο ομοιόμορφα σε αντικείμενα μοιρασμένα σε όλα τα 10 επιμέρους R-trees. Προσπαθήσαμε έτσι να δούμε την συμπεριφορά κάθε μιας από τις 3 μεθόδους εκτέλεσης επερωτημάτων σε queries που αφορούσαν «ομοιόμορφα» σχεδόν και τα 10 δέντρα.

Σε πρώτη φάση εκτελέσαμε την δέσμη «ομοιόμορφων» επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 1η μέθοδο.

Η μέθοδος αυτή θυμίζουμε ότι κατανέμει για εκτέλεση τα επερωτήματα στους Reducers με χωρικό κριτήριο, ανάλογα με το ποιο (ή ποια) από τα επιμέρους R-trees που βρίσκονται αποθηκευμένα σε αρχεία του HDFS απαντούν εν δυνάμει το κάθε ερώτημα. Πιο συγκεκριμένα, ο αλγόριθμος που υλοποιήσαμε για την 1η τεχνικά, κατά την φάση Map βρίσκει για κάθε επερώτημα της εισόδου το αρχείο (ή τα αρχεία) R-tree που απαντάει εν δυνάμει το επερώτημα, κρίνοντας από το MBR της ρίζας κάθε δέντρου, αλλά και από τα MBR των παιδιών της κάθε ρίζας για μεγαλύτερη αξιοπιστία. Τα επερωτήματα διαμοιράζονται στην φάση Reduce χρησιμοποιώντας ως συνάρτηση partitioning την συνάρτηση κατακερματισμού του ονόματος του αρχείου με το R-tree που απαντάει κάθε query. Δημιουργείται έτσι μια ομάδα για κάθε αρχείο R-tree που περιέχει όλα τα επερωτήματα για το δέντρο αυτό. Κάθε κλήση της reduce αρχικοποιεί ένα δέντρο, απαντά τα επερωτήματα που «στοχεύουν» σ' αυτό το δέντρο και αποθηκεύει τα αποτελέσματα στο αρχείο εξόδου.



Εικόνα 6.6: Μέσος χρόνος εκτέλεσης της δέσμης «ομοιόμορφων» επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 1η μέθοδο.

Στην Εικόνα 6.6 φαίνονται οι χρόνοι εκτέλεσης της δέσμης «ομοιόμορφων» επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 1η στρατηγική. Οι χρόνοι εκτέλεσης είναι οι μέσες τιμές που προέκυψαν από πολλές εκτελέσεις των πειραμάτων. Οι δύο γραμμές με τελείες απεικονίζουν τον συνολικό χρόνο εκτέλεσης, ενώ οι δύο γραμμές χαμηλά δείχνουν τον χρόνο που κάνει μόνο η φάση Map. Στη γραφική παράσταση φαίνονται πειράματα και με χρήση της μεθόδου όπου τα MBRs όλων των ριζών φορτώνονται σε δομές της μνήμης για να βρεθούν τα R-trees που απαντάνε κάθε query, αλλά με την περίπτωση τα MBRs αυτά διαβάζονται από τα αρχεία.

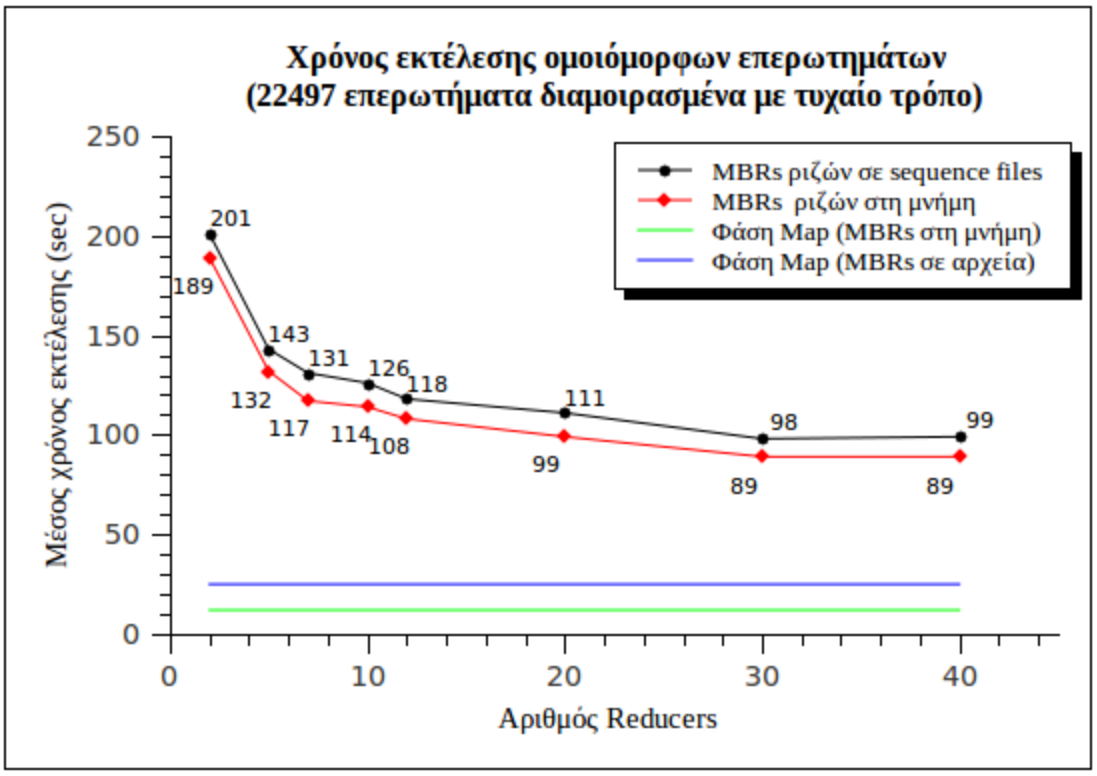
Παρατηρούμε ότι ο χρόνος εκτέλεσης της εργασίας Hadoop μειώνεται όσο αυξάνουμε τους Reducers, μέχρι να φτάσουμε σε αριθμό διεργασιών Reduce τον αριθμό 10, που ισούται με το αριθμό των επιμέρους R-trees που απαρτίζουν το συνολικό. Η μείωση αυτή είναι αναμενόμενη, αφού τα «ομοιόμορφα» επερωτήματα διασπώνται σε περισσότερα μέρη (όσα και ο αριθμός των Reducers), άρα κάθε φάση Reduce έχει μικρότερο αριθμό από ομάδες queries που αντιστοιχούν σε ένα R-tree για επεξεργασία καθώς αυξάνονται οι Reducers. Η μείωση αναμέναμε να μην είναι γραμμική, καθώς οι φάσεις Map, Partitioning και με μικρή διαφορά η Shuffle παίρνουν περίπου ίδιο χρόνο ανεξάρτητα από τον αριθμό των Reducers. Επιπλέον, η αρχικοποίηση περισσότερων tasks είναι σχετικά χρονοβόρα,

πράγμα που πρέπει να αντισταθμιστεί με το κέρδος που έχουμε –ο παράγοντας αυτός επηρεάζει αρκετά την απόδοση του αλγορίθμου εκτέλεσης στην συγκεκριμένη περίπτωση, αφού η είσοδος είναι μικρή για τα δεδομένα του Hadoop. Όσο ο αριθμός των επερωτημάτων προς εκτέλεση αυξάνει, το σημείο όπου μειώνεται πολύ το κέρδος από επιστράτευση περισσότερων reduce tasks θα απομακρύνεται λαμβάνοντας μεγαλύτερες τιμές.

Από εκεί και έπειτα όσο οι Reducers αυξάνονται για τιμές πάνω από 10, η απόδοση του αλγορίθμου αρχίζει να χειροτερεύει. Αυτό συμβαίνει λόγω των επιπλέον Reducers που αρχικοποιούνται, αλλά στην ουσία μένουν αδρανείς καθώς δεν υπάρχουν πάνω από 10 ομάδες επερωτημάτων προς εκτέλεση -όσες δηλαδή και ο αριθμός των επιμέρους αρχείων με R-trees που φτιάξαμε στο HDFS και τώρα χρησιμοποιούμε. Έτσι, όσους παραπάνω Reducers και αν φτιάξουμε, μόνο μέχρι 10 από αυτούς θα έχουν λάβει ζεύγος εισόδου: το ζεύγος του οποίου η τιμή κλειδιού (το όνομα του αρχείου με το R-tree που πρέπει να σαρωθεί) έχει τιμή κατακερματισμού τον αριθμό του συγκεκριμένου Reducer.

Και πάλι, σε clusters με πολλούς κόμβους επεξεργασίας, η κατασκευή R-trees με περισσότερους Reducers, θα έφερνε καλά αποτελέσματα από άποψη χρονικής απόδοσης της εκτέλεσης «ομοιόμορφων» επερωτημάτων με την 1η τεχνική. Και αυτό γιατί σε τέτοια περίπτωση η καμπή της γραφικής παράστασης που σηματοδοτεί την έναρξη πτωτικής απόδοσης θα εμφανιζόταν για μεγάλες τιμές αριθμού Reducers, κάτι που δε θα επηρέαζε αρνητικά την εκτέλεση. Για να εκμεταλλευόμασταν κάτι τέτοιο, θα έπρεπε κάθε εργασία να τροφοδοτούταν με μεγάλο πλήθος επερωτημάτων προς εκτέλεση, ώστε να εξισσοροπεί το κόστος από την αρχικοποίηση περισσότερων tasks.

Σειρά έχει η παρουσίαση των αποτελεσμάτων της εκτέλεσης των ίδιων επερωτημάτων χρησιμοποιώντας την 2η στρατηγική. Περιληπτικά θυμίζουμε ότι κατά την 2η μέθοδο επιδιώκουμε ίση κατανομή των επερωτημάτων στους Reducers, ανεξάρτητα από το R-tree που εν δυνάμει απαντά το κάθε επερώτημα. Όπως και στην 1η μέθοδο, η φάση Map βρίσκει για κάθε επερώτημα της εισόδου το αρχείο (ή τα αρχεία) R-tree που απαντάει εν δυνάμει το επερώτημα. Τα επερωτήματα αυτή τη φορά διαμοιράζονται στην φάση Reduce με βάση την ομοιόμορφη κατανομή με μόνο κριτήριο κάθε Reducer να λάβει ίσο περίπου αριθμό επερωτημάτων. Σε κάθε διεργασία Reduce ομαδοποιούνται σε ένα ζεύγος για κάθε αρχείο R-tree τα επερωτήματα της διαμέρισης για το δέντρο αυτό. Έχουμε μετά μια κλήση της reduce για κάθε διαφορετικό όνομα αρχείου, η οποία αρχικοποιεί το δέντρο, απαντά τα επερωτήματα που «στοχεύουν» σ’ αυτό και αποθηκεύει τα αποτελέσματα στο αρχείο εξόδου.



Εικόνα 6.7: Χρόνοι εκτέλεσης «ομοιόμορφων» επερωτημάτων με ίση κατανομή των επερωτημάτων στους Reducers, ανεξάρτητα από το R-tree που εν δυνάμει απαντά το κάθε επερώτημα

Η Εικόνα 6.7 παρουσιάζει τα αποτελέσματα των πειραμάτων για εκτέλεση επερωτημάτων με την 2η τεχνική. Όπως και πριν παρουσιάζονται οι μέσοι χρόνοι εκτέλεσης που προέκυψαν από πολλές εκτελέσεις των πειραμάτων. Οι δύο γραμμές με τελείες αντιστοιχούν στο συνολικό χρόνο εκτέλεσης, ενώ οι δύο γραμμές χαμηλά δείχνουν τον χρόνο που κάνει μόνο η φάση Map. Έγιναν πειράματα και με τη μέθοδο όπου τα MBRs των ριζών φορτώνονται σε δομές της μνήμης για να βρεθούν τα R-trees που απαντάνε κάθε query, αλλά με την περίπτωση τα MBRs αυτά διαβάζονται από τα αρχεία.

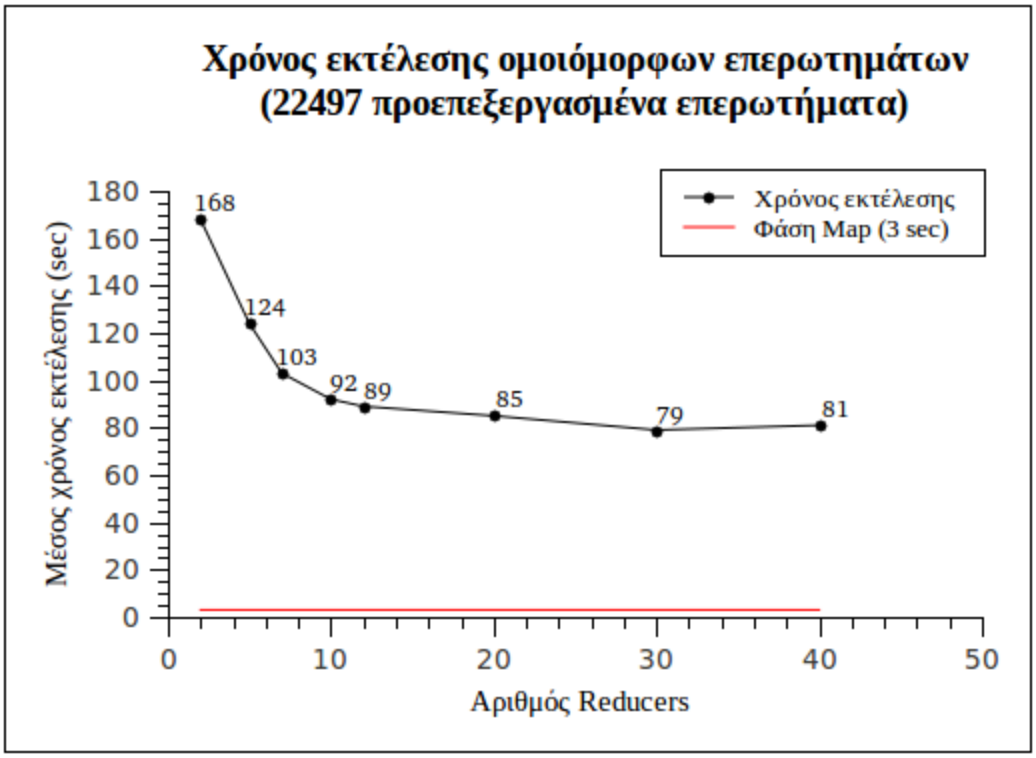
Παρατηρούμε ότι ο χρόνος εκτέλεσης της εργασίας Hadoop μειώνεται όσο αυξάνουμε τους Reducers, και πάνω από τον αριθμό 10, που ισούται με το αριθμό των επιμέρους R-trees που απαρτίζουν το συνολικό. Η μείωση αυτή είναι αναμενόμενη, αφού τα «ομοιόμορφα» επερωτήματα διασπώνται σε περισσότερα μέρη (όσα και ο αριθμός των Reducers), άρα κάθε φάση Reduce έχει μικρότερο αριθμό από ομάδες queries που αντιστοιχούν σε ένα R-tree για επεξεργασία καθώς αυξάνονται οι Reducers. Επιπλέον, μιας και δεν διαμοιράζουμε τα queries με βάση το δέντρο στο οποίο στοχεύουν, η μείωση του χρόνο εκτέλεσης συνεχίζεται και μετά τον αριθμό 10. Φαίνεται λοιπόν ότι σε αντίθεση με την πρώτη μέθοδο, η 2η μπορεί να εκμεταλλευτεί την υπολογιστική ισχύ περισσότερων Reducers. Το μόνο όριο στην αύξηση των Reduce tasks είναι και πάλι η ανάγκη για εξισορρόπηση του

κόστους αρχικοποίησης νέων Reducers σε σχέση με την ποσότητα της δουλειάς που θέλουμε να μοιράσουμε (γι' αυτό στο πείραμά μας που έχουμε σχετικά μικρή είσοδο, ο χρόνος εκτέλεσης μετά από 30 περίπου Reducers αρχίζει να σταθεροποιείται ή να χειροτερεύει ελαφρώς).

Η μείωση του χρόνου εκτέλεσης και πάλι δεν είναι γραμμική, καθώς οι φάσεις Map, Partitioning και με μικρή διαφορά η Shuffle παίρνουν περίπου ίδιο χρόνο ανεξάρτητα από τον αριθμό των Reducers. Παρατηρούμε επίσης ότι ο ρυθμός μείωσης του χρόνου εκτέλεσης δεν είναι μεγάλος, πράγμα που εξηγείται βάσει των πολλών δέντρων που χρειάζεται να ψάξει κάθε Reducer. Αυτή η διαδικασία 10 κλήσεων της συνάρτησης reduce και η αρχικοποίηση 10 δέντρων σε κάθε task, μειώνει λίγο το κέρδος που έχουμε από τη μείωση του όγκου εργασίας που έχει να κάνει ο κάθε Reducer.

Τέλος, κάναμε πειράματα που απάντησαν τα «ομοιόμορφα» queries χρησιμοποιώντας τον 3ο αλγόριθμο που αναπτύχθηκε. Ο αλγόριθμος αυτός διαμοιράζει τα επερωτήματα στους Reducers ανάλογα με το R-tree που εν δυνάμει απαντάει το εκάστοτε επερώτημα, αλλά παράλληλα φροντίζει οι διαμερίσεις αυτές να έχουν τον ίδιο περίπου αριθμό επερωτημάτων. Αυτό γίνεται με προεπεξεργασία των επερωτημάτων που αναλαμβάνει για κάθε επερώτημα στην είσοδο να βρει ποιο αρχείο με επιμέρους R-tree το απαντάει. Στη συνάχεια τα επεξεργασμένα queries, φορτώνονται από το τοπικό σύστημα αρχείων στο HDFS, δρομολογείται η δειγματοληψία τους και η εκτέλεση μιας εργασίας Hadoop με είσοδο το αρχείο αυτό. Η εργασία Hadoop εκτελεί τα επερωτήματα και αποθηκεύει τα αποτελέσματα στο HDFS.

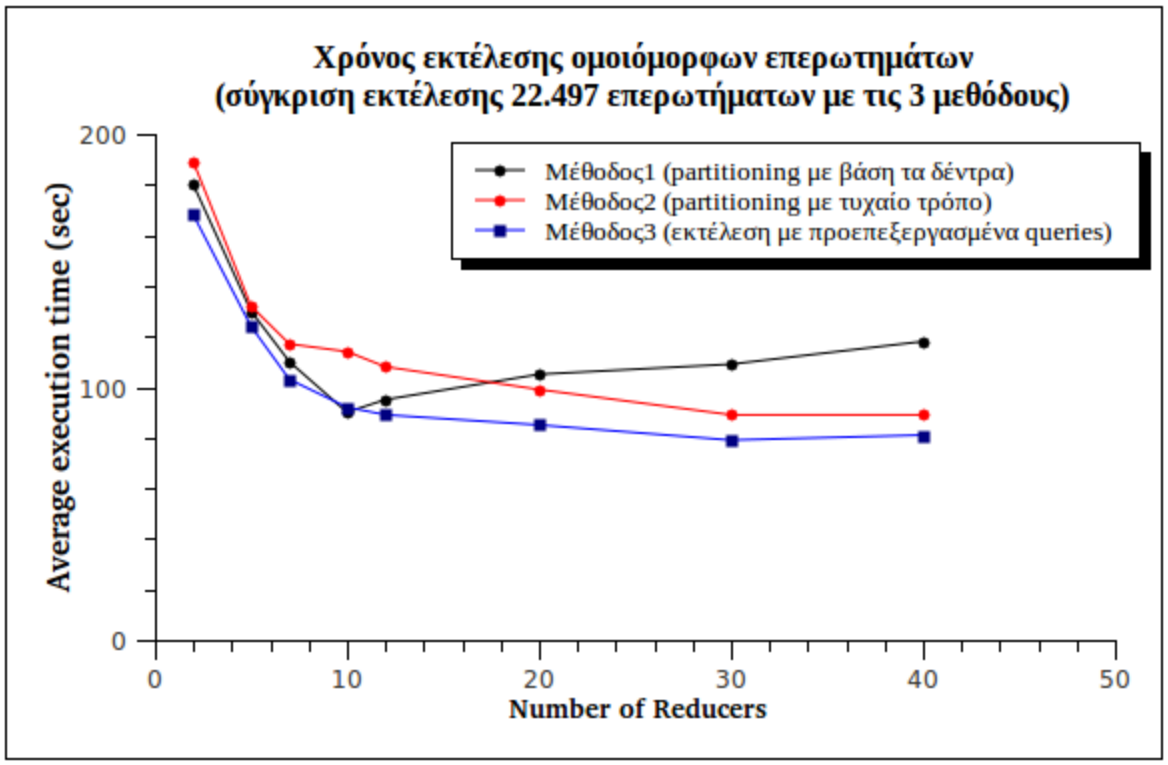
Η Εικόνα 6.8 που ακολουθεί παρουσιάζει τον μέσο χρόνο εκτέλεσης της εργασίας Hadoop μόνο (και το χρόνο εκτέλεσης μόνο της φάσης Map επίσης). Οι χρόνοι για την προεπεξεργασία και την δειγματοληψία των επερωτημάτων ήταν συγκριτικά αμελητέοι, καθώς δεν ξεπερνούσαν το 1 sec. Επιπλέον, ο χρόνος για την αντιγραφή του αρχείου των επεξεργασμένων επερωτημάτων στο HDFS ήταν εξίσου αμελητέος (κάτω από 1 sec), δεδομένου του μικρού μεγέθους του.



Εικόνα 6.8: Μέσος χρόνος εκτέλεσης της εργασίας Hadoop για την εκτέλεση "ομοιόμορφων" επερωτημάτων με την μέθοδο 3.

Τα πειράματα δείχνουν ότι η απόδοση αυξάνεται όσο αυξάνουμε τους Reducers, και πάνω από τον αριθμό 10, που ισούται με το αριθμό των επιμέρους R-trees που απαρτίζουν το συνολικό. Η αύξηση αυτή είναι αναμενόμενη, αφού τα «ομοιόμορφα» επερωτήματα διασπώνται σε περισσότερα μέρη (όσα και ο αριθμός των Reducers), άρα κάθε φάση Reduce έχει μικρότερο αριθμό από ομάδες queries που αντιστοιχούν σε ένα R-tree για επεξεργασία καθώς αυξάνονται οι Reducers. Επιπλέον, και η 2η στρατηγική μπορεί να εκμεταλλευτεί την υπολογιστική ισχύ περισσότερων Reducers από τον αριθμό των επιμέρους R-trees. Και πάλι, στο πείραμά μας λόγω της μικρής εισόδου, ο χρόνος εκτέλεσης μετά από 30 περίπου Reducers αρχίζει να σταθεροποιείται καθώς το κόστος αρχικοποίησης νέων Reducers δεν αντισταθμίζει την μείωση της ποσότητας επερωτημάτων που αναλαμβάνει ο κάθε Reducer.

Η Εικόνα 6.9 παρακάτω εμφανίζει τα αποτελέσματα και των 3 μεθόδων μαζί στην εκτέλεση «ομοιόμορφων» επερωτημάτων, για να διευκολυνθεί η σύγκριση.



Εικόνα 6.9: Σύγκριση της απόδοσης των 3 μεθόδων μαζί στην εκτέλεση «ομοιόμορφων» επερωτημάτων.

Παρατηρούμε ότι για αριθμό Reducers μέχρι 10, οι 3 μέθοδοι έχουν ελάχιστες αποκλίσεις στους χρόνους εκτέλεσης. Αντίθετα, για πάνω από 10 Reducers η κατάσταση διαφοροποιείται, καθώς η πρώτη τεχνική αρχίζει να μειώνεται σε απόδοση, ενώ οι άλλες 2 συνεχίζουν να αυξάνουν.

Παρατηρούμε ότι η 2η μέθοδος έχει λίγο χειρότερη απόδοση για αριθμό Reducers στο διάστημα 2-10, λόγω των 10 R-trees που πρέπει να αρχικοποιήσει ο κάθε Reducer, σε αντίθεση με τις άλλες 2 μεθόδους που σε κάθε Reducer αρχικοποιούνται από 1 εως 5 δέντρα. Η διαφορά στην απόδοση είναι μικρή στην αρχή αλλά προοδευτικά αυξάνεται λίγο, γιατί ενώ η 2η μέθοδος αρχικοποιεί περισσότερα δέντρα, η 1η θα πρέπει και πάλι να συμπτίξει 10 ομάδες δεδομένων σε 2-7 reducers και να τις εκτελέσει σειριακά. Π.χ για 2 Reducers και η 1η και η 3η μέθοδος αρχικοποιούν από 5 δέντρα σε κάθε Reducer. Αντίστοιχα, για 5 Reducers οι μέθοδοι 1 και 3 αρχικοποιούν 2 δέντρα σε καθέναν ενώ για 10 Reducers αρχικοποιούν και οι 2 μόλις 1 δέντρο ανά διεργασία.

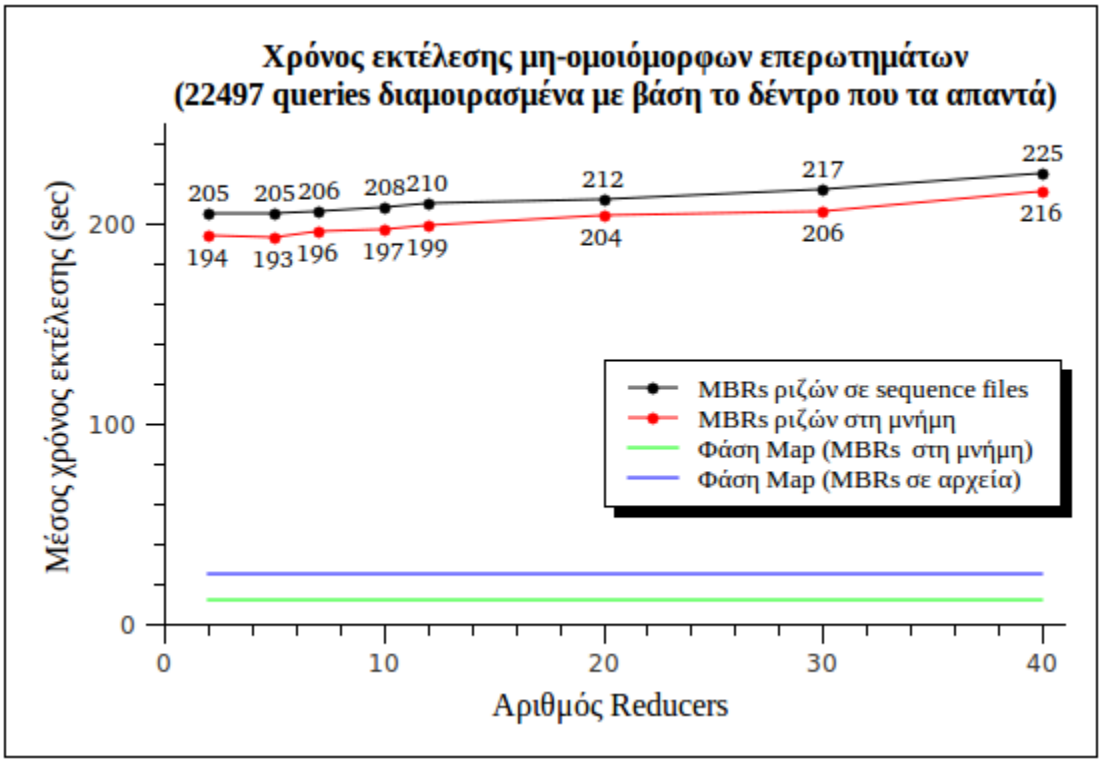
Καθώς ο αριθμός των Reducers ξεπερνάει τους 10, οι μέθοδοι 2 και 3 έχουν καλύτερη απόδοση, με την 3η να υπερτερεί λόγω της μικρότερης σε διάρκεια φάσης Map και της αρχικοποίησης μόλις 1 δέντρου περίπου ανά Reducer στη θέση 10 που αρχικοποιεί η 2η μέθοδος. Αρχικά περιμέναμε μεγαλύτερη διαφορά στην απόδοση. Ωστόσο η διαφορά

μεταξύ τους είναι μικρή, λόγω της πιο ακριβής φάσης Shuffle και sort που χρειάζεται να κάνει η 3η μέθοδος για να μεταφέρει και να ταξινομήσει τα λίγο μεγαλύτερα σύνθετα κλειδιά της.

6.3.2 Εκτέλεση μιας δέσμης μη-ομοιόμορφων, ως προς τα R-trees, επερωτημάτων

Στην ενότητα αυτή θα παρουσιαστούν τα πειράματα από την εκτέλεση ίδιου αριθμού επερωτημάτων με πριν (22.497), αλλά αυτή τη φορά μη ομοιόμορφα μοιρασμένα σε όλα τα 10 επιμέρους R-trees. Συγκεκριμένα, εξετάσαμε τη συμπεριφορά κάθε μιας από τις 3 μεθόδους εκτέλεσης επερωτημάτων σε queries που αφορούσαν δεδομένα *μόνο ενός από τα 10 δέντρα.* Να σημειωθεί, ότι ένα ποσοστό της τάξης του 20% δρομολογήθηκε επίσης σε ένα άλλο δέντρο λόγω επικάλυψης αλλά δεν βγήκαν αποτελέσματα από το 2ο R-tree.

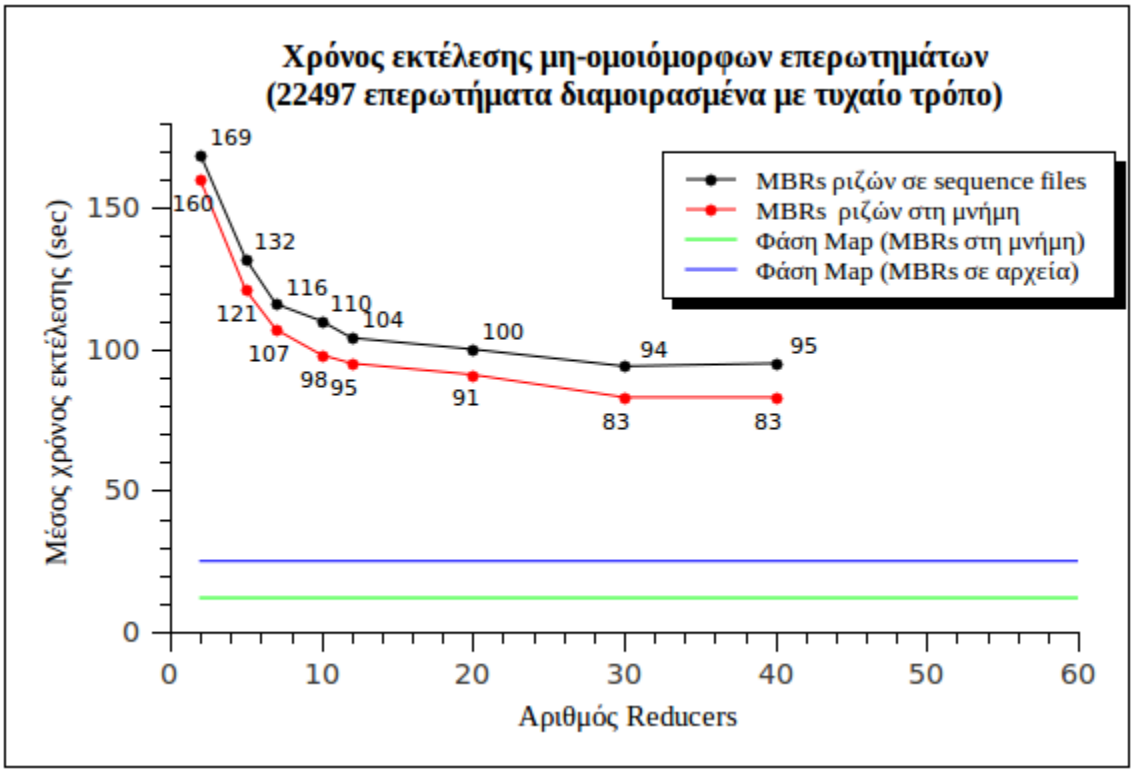
Σε πρώτη φάση εκτελέσαμε την δέσμη μη-ομοιόμορφων επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 1η μέθοδο.



Εικόνα 6.10: Εκτέλεση δέσμης μη-ομοιόμορφων επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 1η μέθοδο.

Από την Εικόνα 6.10, βλέπουμε την ανεπάρκεια της 1ης μεθόδου για εκτέλεση μη-ομοιόμορφων επερωτημάτων. Συγκεκριμένα, όχι μόνο δεν μειώνεται ο χρόνος εκτέλεσης καθώς αυξάνουμε τους Reducers, αλλά αυξάνεται μόλις ο αριθμός των διεργασιών Reduce περάσει τον αριθμό των δέντρων που αφορούν τα επερωτήματα. Αυτό είναι πολύ λογικό, αφού τα επερωτήματα διαμοιράζονται σύμφωνα με το δέντρου που χρησιμοποιούν και οι επιπλέον Reducers παραμένουν αδρανείς, επιβαρύνοντας μόνο με το κόστος δημιουργίας τους.

Στην Εικόνα 6.11 παρουσιάζεται η συμπεριφορά της 2ης μεθόδου στην περίπτωση μη-ομοιόμορφων επερωτημάτων στην ακραία περίπτωση που όλα απαντώνται από 1 δέντρο.

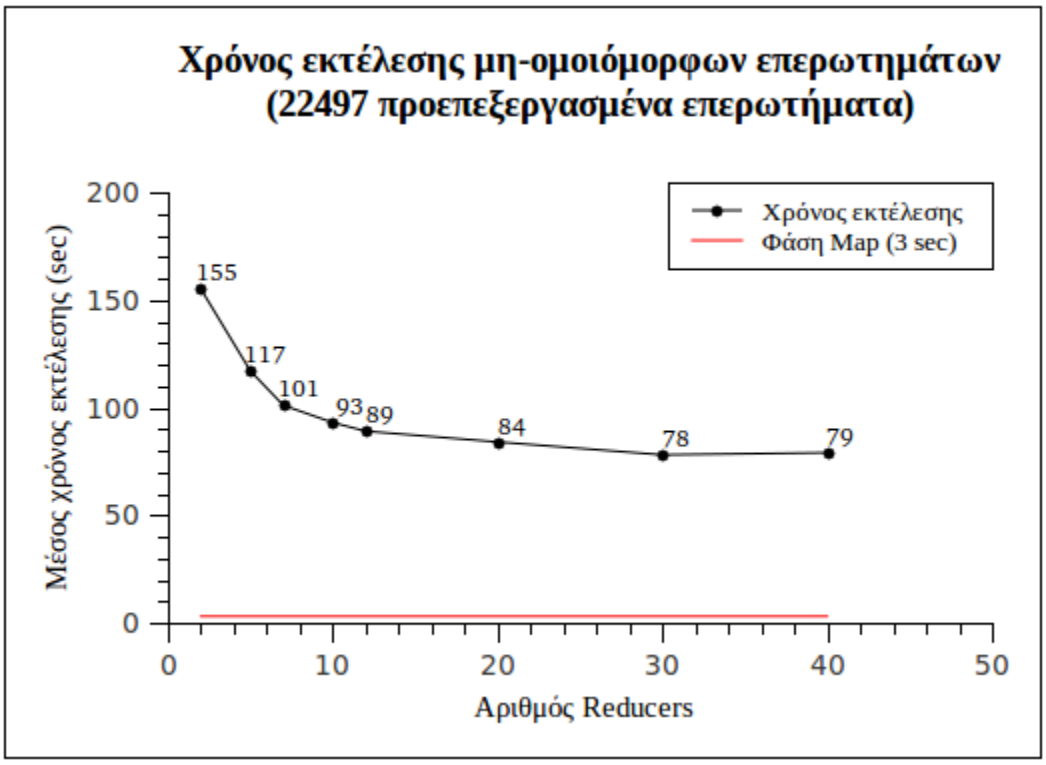


Εικόνα 6.11:Εκτέλεση δέσμης μη-ομοιόμορφων επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στη 2η μέθοδο.

Παρατηρούμε καταρχήν ότι η γραφική παράσταση έχει την ίδια μορφή περίπου με την περίπτωση εκτέλεσης της μεθόδου σε «ομοιόμορφα» επερωτήματα, πράγμα που φανερώνει μια «συνέπεια» ή ανθεκτικότητα του αλγορίθμου σε ποικίλα επερωτήματα ανεξάρτητα από τον αριθμό των δέντρων που χρειάζεται να σαρώσουν τα queries. Ωστόσο, οι χρόνοι είναι λίγο μικρότεροι από τους αντίστοιχους που είχαμε στην εκτέλεση

«ομοιόμορφων» επερωτημάτων, καθώς αυτή τη φορά κάθε Reducer έχει μόνο 1 δέντρο να αρχικοποιήσει (με εξαίρεση 1 Reducer που έχει αρχικοποιεί 2 δέντρα).

Στην Εικόνα 6.12 που ακολουθεί βλέπουμε την απόδοση του αλγορίθμου με χρήση της 3ης μεθόδου που προεπεξεργάζεται τα επερωτήματα πριν δρομολογήσει την εργασία Hadoop.



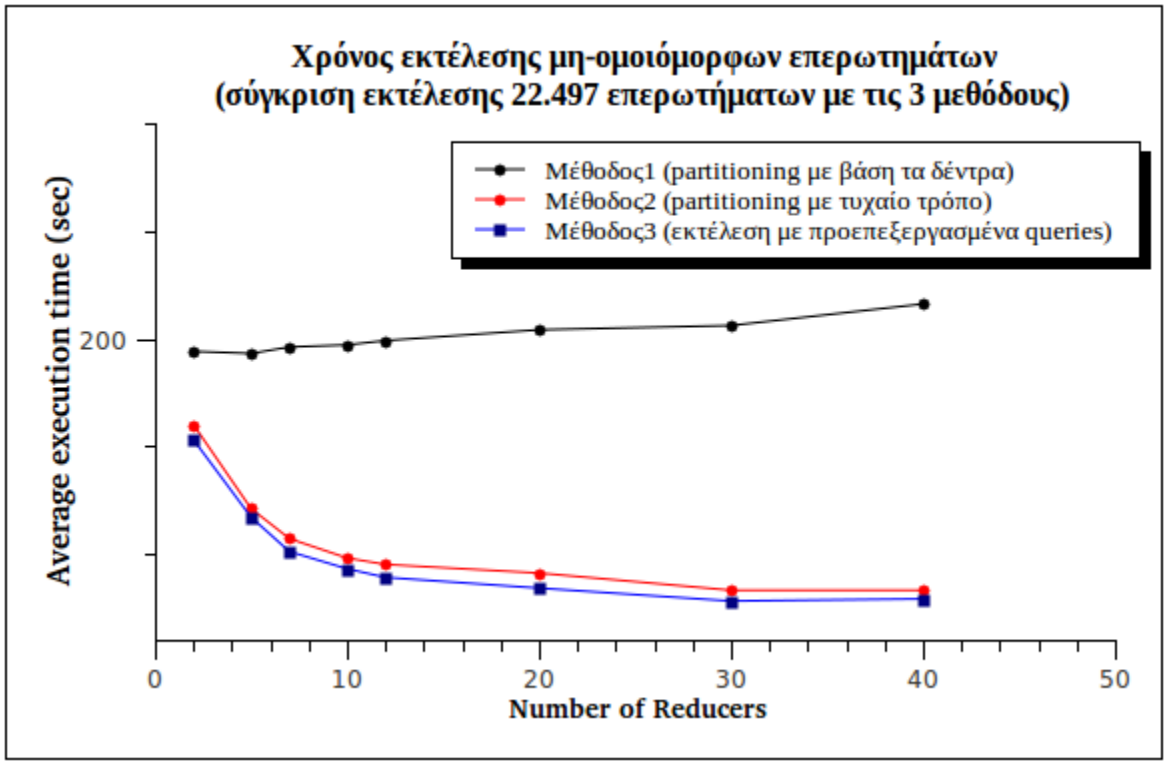
Εικόνα 6.12:Εκτέλεση δέσμης μη-ομοιόμορφων επερωτημάτων με χρήση του αλγορίθμου που αντιστοιχεί στην 3η μέθοδο.

Όπως και στην μέθοδο 2, παρατηρούμε καταρχήν ότι η γραφική παράσταση έχει την ίδια μορφή περίπου με την περίπτωση εκτέλεσης της μεθόδου σε «ομοιόμορφα» επερωτήματα. Επομένως, και η 3η μέθοδος ανταπεξέρχεται καλά σε ποικίλα επερωτήματα, ανεξάρτητα από τον αριθμό των δέντρων που χρειάζεται να σαρώσουν τα queries. Οι χρόνοι είναι λίγο μικρότεροι από τους αντίστοιχους που είχαμε στην εκτέλεση «ομοιόμορφων» επερωτημάτων στο τμήμα της γραφικής παράστασης που αφορά μέχρι 10 Reducers. Αυτό η μικρή διαφορά οφείλεται στο ότι τώρα κάθε Reducer αρχικοποιεί ένα δέντρο, ενώ π.χ. στην περίπτωση «ομοιόμορφων» επερωτημάτων, για 2 Reducers αρχικοποιούνταν περίπου 5 δέντρα ανά διεργασία Reduce.

Τέλος, η Εικόνα 6.13 απεικονίζει την συμπεριφορά και των 3 αλγορίθμων συγκριτικά. Είναι φανερό ότι οι αλγόριθμοι 2 και 3 είναι κατά πολύ αποδοτικότεροι για επερωτήματα που δεν αφορούν όλα τα επιμέρους R-trees στο ίδιο ποσοστό. Ο αλγόριθμος 1 φαίνεται να μην μπορεί να ανταπεξέλθει καλά σε τέτοιες ακραίες περιπτώσεις, πράγμα αναμενόμενο.

Τέλος, αξιοσημείωτη είναι η παρόμοια απόδοση των στρατηγικών 2 και 3, ενώ ομολογουμένως αναμέναμε η 3η μέθοδος να υπερέχει περισσότερο. Σ' αυτό ρόλο παίζει η φάση Shuffle και Sort, καθώς η μέθοδος αυτή χρησιμοποιεί μεγαλύτερα ενδιάμεσα κλειδιά απ' ότι οι άλλες 2.

Ωστόσο, σε ένα υποθετικό σενάριο όπου θα έχουμε 40 αρχεία R-trees και επερωτήματα που να αφορούν τα 20 από αυτά, είναι ασφαλές να αναμένουμε την 3η μέθοδο να έχει καλύτερη απόδοση από την 2η με πιο μεγάλη διαφορά. Αυτό, γιατί θα μεγαλώνει ο αριθμός των κλήσεων της reduce() και των δέντρων που η 2η μέθοδος θα αναγκάζεται να αρχικοποιεί σε κάθε Reducer, σε αντίθεση με την 3η μέθοδο που θα έχει πάντα λιγότερες ομάδες εισόδου σε κάθε Reducer της!



Εικόνα 6.13: Η συμπεριφορά και των 3 αλγορίθμων συγκριτικά για «μη-ομοιόμορφα» επερωτήματα.

7. Συμπεράσματα και Μελλοντικές επεκτάσεις

Ποικίλα γεωγραφικά συστήματα εφαρμογών ή χωρικές βάσεις δεδομένων καλούνται να αποθηκεύσουν και να διαχειριστούν ένα μεγάλο όγκο πολυδιάστατων δεδομένων. Μια καλή τεχνική δεικτοδότησης τέτοιων δεδομένων είναι τα R-trees, που μπορούν να επιταχύνουν ένα ευρύ φάσμα λειτουργιών, όπως την πρόσβαση στα δεδομένα ή την εκτέλεση χωρικών επερωτημάτων. Καθώς αυξάνεται ο όγκος των χωρικών αντικειμένων και η πολυπλοκότητα των υπολογισμών για την ανάλυσή τους, τα παραδοσιακά μοντέλα σειριακής επεξεργασίας αποδεικνύονται ανεπαρκή. Την ίδια στιγμή, νέα παράλληλα υπολογιστικά μοντέλα, όπως το Hadoop MapReduce, φαίνεται να δίνουν μια προοπτική για ανάπτυξη αποδοτικών και επεκτάσιμων αλγορίθμων για διαχείριση χωρικών δεδομένων.

Αντικείμενο της διπλωματικής εργασίας αυτής ήταν η διερεύνηση της δυνατότητας για κατανεμημένη κατασκευή ενός στατικού R-tree χρησιμοποιώντας το περιβάλλον Hadoop, και η χρήση του μετέπειτα για παράλληλη εκτέλεση μιας δέσμης χωρικών επερωτημάτων. Το Hadoop Map/Reduce είναι ένα framework λογισμικού γραμμένο σε Java για την ανάπτυξη εφαρμογών παράλληλης επεξεργασίας μεγάλου όγκου δεδομένων (της τάξης δεκάδων Gigabytes/Terabytes) σε υπολογιστές συνδεδεμένους μεταξύ τους μέσω δικτύου (cluster υπολογιστών), με τρόπο αξιόπιστο και ανεκτικό στις αστοχίες.

Στο πλαίσιο αυτό, σχεδιάστηκε και υλοποιήθηκε ένας αλγόριθμος για την κατασκευή ενός Packed Hilbert R-tree χρησιμοποιώντας το framework Hadoop. Το Packed Hilbert R-tree που σχεδιάσαμε δεικτοδοτεί πολυδιάστατα χωρικά δεδομένα εφαρμόζοντας κατανεμημένα μαζική φόρτωσή τους (bulk-loading) και αποθηκεύεται στο HDFS ενός cluster υπολογιστών. Ο αλγόριθμος έχει σαν αποτέλεσμα μια δομή Packed Hilbert R-tree που αποτελείται από επιμέρους δέντρα αποθηκευμένα σε ξεχωριστά αρχεία. Τα επιμέρους δέντρα εκμεταλλεύονται καλύτερα τον χώρο αποθήκευσης με 100% κατάληψη του χώρου των κόμβων. Παράλληλα, η εκτέλεση στο περιβάλλον Hadoop και η αποθήκευση στο HDFS δίνει τη δυνατότητα για δεικτοδότηση τεράστιου όγκου δεδομένων που δε θα μπορούσε να γίνει σειριακά, ούτε να αποθηκευτεί σε ένα συμβατικό μέσο.

Σε δεύτερη φάση σχεδιάστηκαν και υλοποιήθηκαν 3 διαφορετικοί αλγόριθμοι που εκτελούν μια δέσμη επερωτημάτων κατανεμημένα σε περιβάλλον Hadoop MapReduce, χρησιμοποιώντας ως βοηθητική δομή τα Packed Hilbert R-trees που βρίσκονται αποθηκευμένα στο HDFS. Οι αλγόριθμοι υποστηρίζουν την εκτέλεση τοπολογικών επερωτημάτων, και συγκεκριμένα ερωτημάτων εύρους ή Range Queries που στοχεύουν σε ένα μικρό μέρος ενός συνόλου χωρικών δεδομένων. Ο λόγος για την ανάπτυξη 3 αλγορίθμων ήταν η προσπάθειά μας να προτείνουμε αλγορίθμους που να μπορούν να

ανταπεξέλθουν ικανοποιητικά σε διαφορετικές συνθήκες, ανάλογα με την φύση των επερωτημάτων, το μέγεθος του cluster, και τον βαθμό ομοιομορφίας των χωρικών δεδομένων στα οποία στοχεύει το κάθε query. Σε κάθε βήμα ανάπτυξης του εκάστοτε αλγορίθμου υπήρξε κριτήριο η όσο το δυνατόν καλύτερη δρομολόγηση των επερωτημάτων στους κόμβους επεξεργασίας, και η προσαρμογή της ιεραρχικής δομής των R-tree στις ιδιαιτερότητες του Hadoop και του HDFS.

Τα πειράματά μας έγιναν με τρόπο τέτοιο, ώστε να μετρηθεί η απόδοση της κάθε υλοποίησης ως προς το χρόνο εκτέλεσης, την εγκυρότητα των αποτελεσμάτων εξόδου και την επεκτασιμότητα (παράγοντες scale-up και size-up).

Η πειραματική αξιολόγηση των αλγορίθμων μας οδήγησε στο συμπέρασμα ότι η παραλληλοποίηση της κατασκευής ενός Packed Hilbert R-tree για στατικά χωρικά πολυδιάστατα δεδομένα μπορεί να γίνει με πολύ αποδοτικό τρόπο. Το σημαντικό κέρδος από έναν τέτοιο αλγόριθμο εντοπίζεται ιδιαίτερα στα πλεονεκτήματα του Hadoop που εκμεταλλευόμαστε κατά την δημιουργία του δέντρου:

- ✓ Μείωση του χρόνου εκτέλεσης γενικότερα λόγω παράλληλης επεξεργασίας.
- ✓ Μηδενική αστοχία, καθώς το Hadoop αναλαμβάνει εξ' ολοκλήρου την επίβλεψη και την ομαλή εκτέλεση παρέχοντας μηχανισμούς επαναδρομολόγησης αποτυχημένων διεργασιών.
- ✓ Δυνατότητα να δεικτοδοτήσουμε πολύ μεγάλο όγκο δεδομένων, ενδεχομένως εκατοντάδων GigaBytes, που χωρίς την κατανεμημένη εκτέλεση θα ήταν ουτοπικός.
- ✓ Δυνατότητα να κατασκευάζουμε και να αποθηκεύουμε μεγάλα R-trees σε cluster υπολογιστών μέσω του HDFS, που δεν θα χωρούσαν σε έναν συμβατικό δίσκο.
- ✓ Μείωση του χρόνου κατασκευής του R-tree όσο αυξάνουμε τον αριθμό διεργασιών Reduce, στα όρια των διαθέσιμων κόμβων του cluster (scale-up).
- ✓ Καλή επεκτασιμότητα ως προς το μέγεθος εισόδου για σταθερό αριθμό Reducers (size-up).

Τέλος, ως προς τις 3 μεθόδους εκτέλεσης δέσμης επερωτημάτων που υλοποιήθηκαν, οδηγούμαστε στα παρακάτω συμπεράσματα:

- ✓ Οι 3 διαφορετικές τεχνικές παρέχουν εναλλακτικές ώστε ο αλγόριθμος που θα επιλεγεί κάθε φορά να ανταπεξέλθει ικανοποιητικά στις συνθήκες, ανάλογα με την φύση των επερωτημάτων, το μέγεθος του cluster, και τον βαθμό ομοιομορφίας των χωρικών δεδομένων στα οποία στοχεύει το κάθε query.
- ✓ Οι 3 αλγόριθμοι έχουν παρόμοια απόδοση για «ομοιόμορφα» επερωτήματα που στοχεύουν σε όλα τα δέντρα, όσο ο αριθμός των Reducers δεν ξεπερνά το πλήθος των επιμέρους R-trees που απαντάνε συνολικά τα επερωτήματα. Όταν ο αριθμός

των διεργασιών Reduce αυξηθεί κι άλλο, οι μέθοδοι 2 και 3 παρουσιάζουν καλύτερη απόδοση, σε αντίθεση με την 1η μέθοδο που η απόδοσή της πέφτει.

- ✓ Για δέσμη μη «ομοιόμορφων» επερωτημάτων που στοχεύουν σε μικρό αριθμό R-trees σε σχέση με τα υπάρχοντα, η 1η μέθοδος δεν είναι επεκτάσιμη όταν οι Reducers αυξάνονται πάνω από τον αριθμό των δέντρων που στοχεύουν τα επερωτήματα. Αντίθετα, οι μέθοδοι 2 και 3 παρουσιάζουν πολύ καλή απόδοση και σ' αυτή την περίπτωση, με την 3η μέθοδο να υπερτερεί.
- ✓ Οι 3 συγκεκριμένοι αλγόριθμοι αφορούν επερωτήματα που απαντώνται από το ίδιο περίπου κατά μέσο όρο αριθμό απαντήσεων (αυτό εξαρτάται και από το μέγεθος παραθύρου). Σε άλλη περίπτωση, οι στρατηγικές αυτές θα εμφάνιζαν καθυστέρηση από την πιο αργή διεργασία Reduce, που θα έπρεπε να δώσει σημαντικά μεγαλύτερο αποτέλεσμα.

Μελλοντική εργασία πάνω στο θέμα, θα μπορούσε να επικεντρωθεί στην σύγκριση της απόδοσης των αλγορίθμων εκτέλεσης επερωτημάτων με περιβάλλοντα όπως το HBase, που φέρονται να ενδείκνυνται για low-latency εργασίες.

Επιπλέον, θα μπορούσε να μελετήσει την απόδοση του παραγόμενου R-tree για εκτέλεση επερωτημάτων που γίνονται μεμονωμένα, από ένα περιβάλλον όπως το Hadoop Fuse. Αυτό θα μπορεί να γίνει εύκολα, χρησιμοποιώντας την ήδη υπάρχουσα βιβλιοθήκη HDFSStorageManager που έχουμε αναπτύξει, ώστε να επιτυγχάνεται πρόσβαση στο R-tree του HDFS.

Τέλος, μια μελλοντική επέκταση θα μπορούσε να μελετήσει την εκτέλεση και άλλων μορφών επερωτημάτων, όπως κοντινότερου γείτονα (k-nearest neighbor queries). Αναμένουμε ωστόσο, ότι πιθανή χρήση του R-tree σε επερωτήματα που απαιτούν Join εξετάζοντας μεγάλο μέρος του συνόλου δεδομένων δεν θα είναι αποδοτική. Αυτό, λόγω του αυξημένου κόστους που έχουν οι λειτουργίες Random Read στο HDFS, που κάνουν άλλες τεχνικές εκτέλεσης Join καταλληλότερες από τη χρήση ευρετηρίων.

Βιβλιογραφία

[1] A. Guttman. "R-trees: a dynamic index structure for spatial searching", Proceedings of the SIGMOD Conference, Boston, MA, June 1984, pages 47-57

[2] Yannis Manolopoulos , Alexandros Nanopoulos , Apostolos N. Papadopoulos , Yannis Theodoridis. "R-Trees Have Grown Everywhere", 2007

[3] Y. Manolopoulos, Y. Theodoridis and V. Tsotras. "Advanced Database Indexing", Kluwer Academic Publishers, 1999.

[4] H. Samet. "The Design and Analysis of Spatial Data Structures", Addison-Wesley, Reading MA, 1990.

[5] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*". In* Proceedings of the 13th International Conference on Very Large Data Bases *(VLDB '87*), Peter M. Stocker, William Kent, and Peter Hammersley (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pages 507-518.

[6] N. Roussopoulos and D. Leifker: "Direct Spatial Search on Pictorial Databases Using Packed R-trees", Proceedings ACM SIGMOD Conference, Austin, TX, 1985, pages 17-31

[7] Ibrahim Kamel and Christos Faloutsos. 1993. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management* (CIKM '93), Bharat Bhargava, Tim Finin, and Yelena Yesha (Eds.). ACM, New York, NY, USA, pages 490-499

[8] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. 1997. Space-filling curves and their use in the design of geometric data structures. *Theor. Comput. Sci.* vol 181, issue 1 (July 1997), pages 3-15

[9] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB '94), Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pages 500-509

[10] C. Faloutsos and S. Roseman. 1989. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (PODS '89). ACM, New York, NY, USA, pages 247-252

[11] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. 1997. Space-filling curves and their use in the design of geometric data structures. *Theor. Comput. Sci.* vol 181, issue 1 (July 1997), pages 3-15

[12] J.K.Lawder and P.J.H.King. Using Space-filling Curves for Multi-dimensional Indexing. In Brian Lings & Keith Jeffery, editors, *Advances in Databases: proceedings of the 17th British National Conference on Databases (BNCOD 17)*, volume 1832 of *Lecture Notes in Computer Science*, pages 20 - 35. Springer Verlag, July 2000

[13] T. Brinkhoff, H. Horn, H.-P. Kriegel and R. Schneider. "A Storage and Access Architecture for Efficient Query Processing in Spatial Database Systems", Proceedings 3rd SSD Symposium, Singapore, 1993, pages 357-376

[14] D. Papadias, Y. Theodoridis, T. Sellis and M. Egenhofer: "Topological Relations in the World of Minimum Bounding Rectangles: a Study with R-trees", Proceedings ACM SIGMOD Conference, San Jose, CA, 1995, pages 92-103

[15] Nick Koudas, Christos Faloutsos, and Ibrahim Kamel. 1996. Declustering Spatial Databases on a Multi-Computer Architecture. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology* (EDBT '96), Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin (Eds.). Springer-Verlag, London, UK, pages 592-614

[16] Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, USENIX Association, Volume 6, December 2004, pages 10-10

[17] Schnitzer B., Leutenegger S.T.: Master-client R-trees: a new parallel R-tree architecture, In Proceedings of the 11th International Conference on Scientific and Statistical Database Management, , August 1999, pages 68-77.

[18] Apostolos Papadopoulos and Yannis Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Comput.* 29, 10 (October 2003), pages 1419-1444.

[19] I. Kamel and C. Faloutsos: "Parallel R-trees", Proceedings ACM SIGMOD Conference, San Diego, CA, 1992, pages 195-204

[20] Apache Hadoop project: http://hadoop.apache.org

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, "The Hadoop Distributed File System", pages 1-10, 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010

[22] Google's MapReduce programming model -Revisited. Sci. Comput. Program. 68, 3 (October 2007), pages 208-237

[23] Yahoo! Hadoop Tutorial: http://developer.yahoo.com/hadoop/tutorial/index.html

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. *SIGOPS Oper. Syst. Rev.* vol.37, issue 5 (October 2003), pages 29-43

[25] Shafer, J.; Rixner, S.; Cox, A.L , The Hadoop distributed filesystem: Balancing portability and performance , Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium, pages 122 – 133.

[26] Hadoop: The Definitive Guide, Second Edition, Tom White, published by O'Reilly Media 2010.

[27] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, Alexander Rasin, MapReduce and Parallel DBMSs: Friends or Foes?, Communications of the ACM Vol. 53 No. 1, pages 64-71.

[27] Spatial Index Library, Marios Hadjieleftheriou: http://libspatialindex.github.com/

[28] Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, , Addison Wesley. October 1994.

[29] Owen O'Malley: The Anatomy of Hadoop I/O Pipeline, Yahoo!, August 2009

[30] Haojun Liao, Jizhong Han, and Jinyun Fang. 2010. Multi-dimensional Index on Hadoop Distributed File System. In Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage (NAS '10). IEEE Computer Society, Washington, DC, USA, pages 240-249.

[31] H. Jagadish. Linear clustering of objects with multiple attributes. SIGMOD '90 Proceedings of the 1990 ACM SIGMOD international conference on Management of data, , Atlantic City, NJ, May 1990, pages 332–342.

[32] O'Malley: TeraByte Sort on Apache Hadoop, Yahoo!, May 2008.

[33] U.S. Census Bureau, TIGER/Line® Shapefiles and TIGER/Line® Files, CAR dataset http://www.census.gov/geo/www/tiger/shp.html

[34] Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. 2009. Experiences on Processing Spatial Data with MapReduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management* (SSDBM 2009), Marianne Winslett (Ed.). Springer-Verlag, Berlin, Heidelberg, pages 302-319.