Πανεπιστήμιο Θεσσαλίας

Πολυτεχνική Σχολή

Τμήμα Ηλεκτρολόγων Μηχανικών Και Μηχανικών Η/Υ

Διπλωματική Εργασία

"Πολυδιάστατες δομές ευρετηρίου σε δίσκους στερεάς κατάστασης "

"Multidimensional indexes for solid state disks"

Νιζάμης Αλέξανδρος

Επιβλέποντες:

Μποζάνης Παναγιώτης
Αναπληρωτής Καθηγητής

Κατσαρός Δημήτριος
Λέκτορας

Βόλος, 2013

Στη Γεωργία, στην οικογένεια και στους φίλους μου

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας θα ήθελα να ευχαριστήσω τους επιβλέποντες της διπλωματικής μου εργασίας κ. Μποζάνη Παναγιώτη και κ. Κατσαρό Δημήτριο για την εμπιστοσύνη που έδειξαν στο πρόσωπο μου με την ανάθεση του συγκεκριμένου θέματος. Επίσης θα ήθελα να ευχαριστήσω θερμά τον κ. Φεύγα Αθανάσιο για τη βοήθεια που μου πρόσφερε από την αρχή της εργασίας καθώς και για την άριστη συνεργασία.
Τέλος οφείλω ένα μεγάλο ευχαριστώ στην οικογένεια μου και στους φίλους μου που ήταν πάντα δίπλα μου τόσο στη δημιουργία της διπλωματικής μου εργασίας όσο και σε όλη τη διάρκεια των σπουδών μου.

Νιζάμης Αλέξανδρος

Βόλος, 2013

Περίληψη

Στη παρούσα διπλωματική εργασία γίνεται παρουσίαση της δομής και των χαρακτηριστικών των δίσκων στερεάς κατάστασης, μελέτη του R-δέντρου που αποτελεί παραδοσιακή πολυδιάστατη δομή ευρετηρίου και πραγματοποιείται μια υλοποίηση του δέντρου αυτού κατάλληλη για τους προαναφερθέντες δίσκους. Ουσιαστικά παρουσιάζεται το θεωρητικό υπόβαθρο που απαιτείται για την κατανόηση της ανάγκης δημιουργίας μιας τέτοιας υλοποίησης , η υλοποίηση και τα πειραματικά της αποτελέσματα.

Πίνακας περιεχομένων

Κεφάλαιο 1: Εισαγωγή 10

1.1 Η ανάγκη της υλοποίησης 10

1.2 Η διάθρωση της εργασίας...... 10

Κεφάλαιο 2: Χαρακτηριστικά και Δομή των Δίσκων Στερεάς Κατάστασης 12

2.1 Η αποθήκευση δεδομένων μέσα στο χρόνο 12

2.2 SSD vs. HDD 13

2.3 Χαρακτηριστικά μνήμης φλας 14

2.4 Τύποι μνήμης φλας 14

2.5 Αρχιτεκτονική των Δίσκων Στερεάς Κατάστασης 15

Κεφάλαιο 3: Πολυδιάστατες δομές δεδομένων και R-δέντρα 17

3.1Χαρακτηριστικά και δομή του R-δέντρου 17

3.2 Πράξεις στο R-δέντρο 18

Κεφάλαιο 4: Μια αποδοτική υλοποίηση του R-δέντρου πάνω από το FTL 21

4.1 Ανάγκες για μια αποδοτική υλοποίηση του R-δέντρου 21

4.2 Η υλοποίηση...... 22

4.3 Δομές Δεδομένων...... 24

4.4 Το πακετάρισμα των Index Units 25

4.5 Συμπίεση των κόμβων του R-δέντρου 26

4.6 Ανάλυση Πολυπλοκότητας...... 27

4.7 Πειραματικά Αποτελέσματα 28

4.8 Συγγενείς Μελέτες και Υλοποιήσεις 30

Κεφάλαιο 5: Η Υλοποίηση 31

5.1 Η βάση της υλοποίησης 31

5.2 Παραδοχές και συμβάσεις 31

5.3 Λεπτομέρειες της υλοποίησης 32

Κεφάλαιο 6: Πειραματικά Αποτελέσματα 37

6.1 Προδιαγραφές συστήματος και πειραμάτων 37

6.2 Πειράματα 37

Παράρτημα 43

Παραπομπές 53

Λίστα εικόνων και πινάκων

Εικόνα 2.1 Η δομή ενός SSD 15

Εικόνα 3.1 Απεικονίσεις R-δέντρου 18

Εικόνα 4.1 Ένα R-δέντρο 21

Εικόνα 4.2 Η αρχιτεκτονική του συστήματος 23

Εικόνα 4.3 κατάσταση μετά την εγγραφή του Reservation Buffer 25

Εικόνα 4.4 Η σύμπτυξη ενός κόμβου 26

Εικόνα 4.5 Αποτελέσματα της κατασκευής του R-δέντρου 28

Εικόνα 4.6 Η απόδοση του συστήματος σε μεταβολές 29

Εικόνα 4.7 Σύγκριση LCR-δέντρου με R-δέντρο 30

Εικόνα 6.1 αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 1 38

Εικόνα 6.2 αριθμός σελίδων του δίσκου που εγράφησαν στο Πείραμα 1 38

Εικόνα 6.3 Ο αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 2 39

Εικόνα 6.4 αριθμός σελίδων του δίσκου που εγράφησαν στο Πείραμα 2 39

Εικόνα 6.5 Ο αριθμός σελίδων του δίσκου που διαγράφηκαν στο Πείραμα 2 40

Εικόνα 6.6 Η Ο αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 3 40

Πίνακας 1.1 Σύγκριση SSD με HDD 13

Πίνακας 2.1 Σύγκριση μνημών Nand και Nor 15

Πίνακας 4.1 Η ενεργειακή κατανάλωση 29

Κεφάλαιο 1

Εισαγωγή

1.1 Η ανάγκη της υλοποίησης

Οι δίσκοι στερεάς κατάστασης καταλαμβάνουν ολοένα και μεγαλύτερο κομμάτι της αγοράς των μέσων αποθήκευσης. Έχουν γίνει τόσο δημοφιλείς γιατί σε σχέση με τους προκατόχους τους προσφέρουν μεγάλες ταχύτητες εγγραφής και ανάγνωσης δεδομένων καθώς και αυξημένη αξιοπιστία. Οι δίσκοι αυτοί παρουσιάζουν όμως κάποιες ιδιομορφίες. Έχουν μεγαλύτερη ταχύτητα ανάγνωσης από ότι εγγραφής και ακόμα μικρότερη ταχύτητα διαγραφής. Έχουν και συγκεκριμένο αριθμό διαγραφών για κάθε μπλοκ και αν ξεπεραστεί ο αριθμός αυτός φθείρονται και καταστρέφονται. Τα χαρακτηριστικά αυτά οφείλονται στα χαρακτηριστικά των μνημών φλας που αποτελούν το βασικό στοιχείο της δομής των δίσκων αυτών. Είναι απαραίτητο λοιπόν οι παραδοσιακές δομές που έχουν κατασκευαστεί για τους κλασσικούς σκληρούς δίσκους να τροποποιηθούν κατάλληλα ώστε να λαμβάνουν υπόψη τις προαναφερθείσες ιδιαιτερότητες. Η υλοποίηση μιας τέτοιας τροποποιημένης δομής αποτελεί και το βασικό κομμάτι της εργασίας αυτής. Πρόκειται για μια παραλλαγή της πολυδιάστατης δομής R-δέντρο[1], ώστε αυτή να συμβαδίζει με τα χαρακτηριστικά των δίσκων στερεάς κατάστασης.

1.2 Η διάθρωση της εργασίας

Η εργασία νοητά χωρίζεται σε τρείς κύριες ενότητες. Η πρώτη ενότητα περιέχει τα κεφάλαια 2 και 3 και παρέχει όλο το απαραίτητο θεωρητικό υπόβαθρο. Στο κεφάλαιο 2 γίνεται παρουσίαση των χαρακτηριστικών και της δομής των δίσκων στερεάς κατάστασης ενώ στο κεφάλαιο 3 γίνεται μια αναλυτική παρουσίαση του R-δέντρου. Η δεύτερη ενότητα που περιέχει το κεφάλαιο 4 αποτελεί την παρουσίαση της μελέτης πάνω στην οποία στηρίζεται η εν λόγω υλοποίηση του R-δέντρου και μια σύντομη αναφορά σε σχετικές υλοποιήσεις.

Τέλος η τρίτη ενότητα αποτελεί την περιγραφή της υλοποίησης που πραγματοποιήθηκε καθώς και την παρουσίαση των πειραματικών αποτελεσμάτων. Κεφάλαια 5 και 6 αντίστοιχα.

Κεφάλαιο 2

Χαρακτηριστικά και Δομή των Δίσκων Στερεάς Κατάστασης

2.1 Η αποθήκευση δεδομένων μέσα στο χρόνο

Η έννοια της αποθήκευσης δεδομένων είναι θεμελιώδης στην επιστήμη των υπολογιστών. Αφορά από την αποθήκευση του λειτουργικού συστήματος του Η/Υ μέχρι εντελώς προσωπικών δεδομένων όπως εικόνες, μουσική και έγγραφα. Τα μέσα αποθήκευσης δεδομένων[2] είναι συσκευές που χρησιμοποιούνται για την αποθήκευση δεδομένων και πληροφοριών και θεωρούνται η δευτερεύουσα μνήμη του Η/Υ. Είναι σημαντικό να προσφέρουν ασφάλεια και γρήγορη προσπέλαση. Έχουν σημειώσει τεράστια πρόοδο από την πρώτη τους εμφάνιση και είναι άρρηκτα συνδεδεμένα με τη γενικότερη εξέλιξη της επιστήμης των υπολογιστών. Η αρχαιότερη γνωστή μορφή αποθηκευτικού μέσου είναι η διάτρητη κάρτα όπου χρονολογείται στο 1725. Το 1864 εισήχθη για πρώτη φορά η διάτρητη ταινία η οποία ήταν αναδιπλούμενη και κάθε γραμμή αντιπροσώπευε ένα χαρακτήρα. Το 1946 η RCA (Radio Corporation of America) ξεκίνησε την ανάπτυξη της λυχνίας (*selectron tube*), μια προγενέστερη μορφή μνήμης υπολογιστή με μέγιστο μέγεθος 10 ίντσες και μέγιστη χωρητικότητα 4.096 bits. Παρόλα αυτά δεν βρήκε ανταπόκριση λόγω του μεγάλου κόστους των λυχνιών την εποχή εκείνη. Στις αρχές της δεκαετίας του 1950, μαγνητικές ταινίες χρησιμοποιήθηκαν από την IBM για αποθήκευση δεδομένων. Ο αποθηκευτικός χώρος των μαγνητικών ταινιών ήταν 10.000 φορές μεγαλύτερος των καρτών. Το 1956 IBM παρουσίασε τον πρώτο Η/Υ με σκληρό δίσκο. Είχε αποθηκευτικό χώρο της τάξεως των 4,4 MB (5 εκατομμύρια χαρακτήρες), μέγεθος τεράστιο για εκείνη την εποχή. Τα δεδομένα αποθηκεύονταν σε 50 μαγνητικούς δίσκους των 24 ιντσών. Οι σύγχρονοι σκληροί δίσκοι φτάνουν σε μέγεθος τουλάχιστον τα 3TB έχουν χρόνο προσπέλασης 11,5 ms - 4,16ms και ρυθμό μεταφοράς δεδομένων 1,5Gbps - 6Gbps. Τέλος από τα μέσα της δεκαετίας του 90' μια νέα μορφή δίσκων έκανε την εμφάνισή της. Οι δίσκοι στερεάς κατάστασης (SSD).Ο χρόνος προσπέλασης είναι κάτω των 100μs ενώ ο ρυθμός μεταφοράς δεδομένων μπορεί να φτάσει τα 600MB/sec.

2.2 SSD vs. HDD

Στον παρακάτω πίνακα παρουσιάζονται μερικά από τα κύρια χαρακτηριστικά ενός δίσκου αποθήκευσης δεδομένων και το πώς αυτά διαφοροποιούνται ανάλογα με τον τύπο του δίσκου.

Χαρακτηριστικό	Solid State Drive	Hard Disk Drive
Χρόνος εκκίνησης :	Ουσιαστικά μηδαμινός	Μερικά δευτερόλεπτα
Χρόνος τυχαίας προσπέλασης :	Λιγότερο των 100μs	Μεταξύ 3-12ms
Ρυθμός μεταφοράς δεδομένων :	Από 100MB/s έως 600MB//s	Μέχρι 140MB/s
Θόρυβος :	Αθόρυβος	Προκαλείται από την κίνηση των μηχανικών μερών.
Θερμοκρασία :	Ανεπηρέαστος	Πάνω από τους 55C είναι αναξιόπιστος.
Μέγεθος :	Μιας κυκλωματικής πλακέτας	2.5-3.5 ίντσες
Αξιοπιστία :	Μεγάλη	Μέτρια-Εξαρτάται από μηχανικά μέρη
Διάρκεια ζωής :	Έχει περιορισμένο αριθμό εγγραφών	Θεωρητικά 10 χρόνια
Κόστος προς χωρητικότητα:	0.59$ ανά GB	0.075 ανά GB
Χωρητικότητα αποθήκευσης :	Έως 2TB	Έως 4TB(εσωτερικός)
Κατανάλωση ενέργειας :	Το 1/6 των HDD	Έως 20 watts

Πίνακας 1.1 Σύγκριση SSD με HDD

2.3 Χαρακτηριστικά μνήμης φλας

Οι δίσκοι στερεάς κατάστασης είναι βασισμένοι στην τεχνολογία των μνημών τύπου φλας. Γι' αυτό είναι σκόπιμο να αναφερθούν και να αναλυθούν τα χαρακτηριστικά των μνημών αυτών αφού στην ουσία αποτελούν χαρακτηριστικά των SSD.

- Είναι μια ηλεκτρονικά μη ευμετάβλητη μνήμη που μπορεί να διαγραφεί και να επαναπρογραμματιστεί ηλεκτρικά.
- Δεν περιέχει μηχανικά μέρη με αποτέλεσμα να είναι ανθεκτική σε κραδασμούς καθιστώντας την έτσι ένα πιο ασφαλές αποθηκευτικό μέσο.
- Χαρακτηρίζεται από δυσαρμονία στις ταχύτητες ανάγνωσης και εγγραφής. Οι αναγνώσεις είναι σημαντικά ταχύτερες από τις εγγραφές.
- Διαθέτει όριο στον αριθμό των εγγραφών. Το όριο αυτό αφορά από 10.000 ως και 1.000.000 εγγραφές και αν ξεπεραστεί τότε η μνήμη αρχίζει να φθείρεται.
- Σώζει τα δεδομένα σε ένα πίνακα από κελιά μνήμης αποτελούμενα από τρανζίστορ[3]. Χωρίζεται σε single-level cell όπου σε κάθε κελί σώζεται μόνο ένα bit πληροφορίας και σε multi-level cell όπου σε κάθε κελί μπορεί να αποθηκευτεί πληροφορία περισσότερων bit.
- Έχει εντυπωσιακές επιδόσεις στον τομέα της ενεργειακής κατανάλωσης.

2.4 Τύποι μνήμης φλας

Η μνήμη φλας χωρίζεται σε δυο μεγάλες κατηγορίες. Στις NOR και στις NAND μνήμες φλας.

Προγενέστερη ήταν από η μνήμη τύπου NOR. Η μνήμη αυτή παρέχει τυχαία προσπέλαση ενώ διευθυνσιοδοτείται σε επίπεδο byte με αποτέλεσμα να προσφέρει και δυνατότητας εγγραφής σε αυτό το επίπεδο. Η μνήμη αυτή προσφέρει υψηλούς χρόνους προσπέλασης και χρησιμοποιείται κυρίως για την αποθήκευση κώδικα.

Στη συνέχεια έκανε την εμφάνιση της η NAND μνήμη φλας. Η διευθυνσιοδότηση της μνήμης αυτής γίνεται σε επίπεδο σελίδας. Κάθε NAND φλας αποτελείται από ένα αριθμό μπλοκ καθένα από τα οποία περιέχει ένα αριθμό σελίδων. Τα δεδομένα γράφονται στις σελίδες αυτές. Χαρακτηρίζεται από υψηλή πυκνότητα και χρησιμοποιείται για την αποθήκευση δεδομένων.

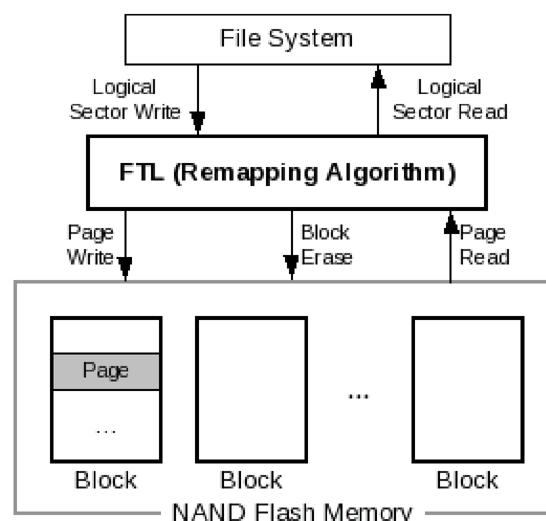
Στον παρακάτω πίνακα παρουσιάζεται μια σύγκριση μεταξύ των δύο παραπάνω τύπων της μνήμης φλας :

Χαρακτηριστικό	NOR	NAND
Προσπέλαση :	Τυχαία	Ανά σελίδα
Αξιοπιστία :	Μεγαλύτερη	Μικρότερη
Ταχύτητα διαγραφών :	Μικρότερη	Μεγαλύτερη
Ανάγνωση :	Ταχύτερη τυχαία	Ταχύτερη σε ροή
Πυκνότητα :	Χαμηλότερη	Υψηλότερη
Αποθήκευση :	Κώδικα	Δεδομένα
Διάρκεια ζωής :	Μεγαλύτερη	Μικρότερη

Πίνακας 2.1 Σύγκριση μνημών Nand και Nor

2.5 Αρχιτεκτονική των Δίσκων Στερεάς Κατάστασης

Όπως έχει αναφερθεί παραπάνω οι δίσκοι στερεάς κατάστασης ανήκουν στην οικογένεια της μνήμης φλάς η οποία αποτελεί στην ουσία το βασικό κομμάτι της δομής τους. Η ακριβής δομή ενός τέτοιου δίσκου εμφανίζεται στο σχήμα που ακολουθεί :



Εικόνα 2.1 Η δομή ενός SSD

Από το σχήμα προκύπτει πως υπάρχουν τρία διακριτά μέρη στη δομή αυτή.Μια NAND μνήμη όμοια με αυτή που έχει περιγραφεί, το FTL και το File System.

Το **File System[4]** μπορεί ναι είναι οποιοδήποτε. Τα Log-structured συστήματα αρχείων περιέχουν όλα τα επιθυμητά χαρακτηριστικά για τα φλας συστήματα. Τα σημαντικότερα είναι τα JFFS (Journaling Flash File System) και YAFFS (Yet Another Flash File System).

Το **FTL** ή πιο ολοκληρωμένα Flash Translation Layer αποτελεί ουσιαστικά ένα στρώμα που προσφέρει ένα interface σε επίπεδο μπλοκ στις φλας συσκευές. Αντιστοιχίζει της λογικές διευθύνσεις σε φυσικές, διαχειρίζεται τη συλλογή απορριμμάτων[5] και εξισορροπεί τα επίπεδα φθοράς της μνήμης φλας. Ουσιαστικά αυτό που επιτυγχάνει είναι να προσφέρει τη δυνατότητα σε συστήματα αρχείων σχεδιασμένα για HDD δίσκους να μπορούν να χρησιμοποιηθούν και στους SSD.

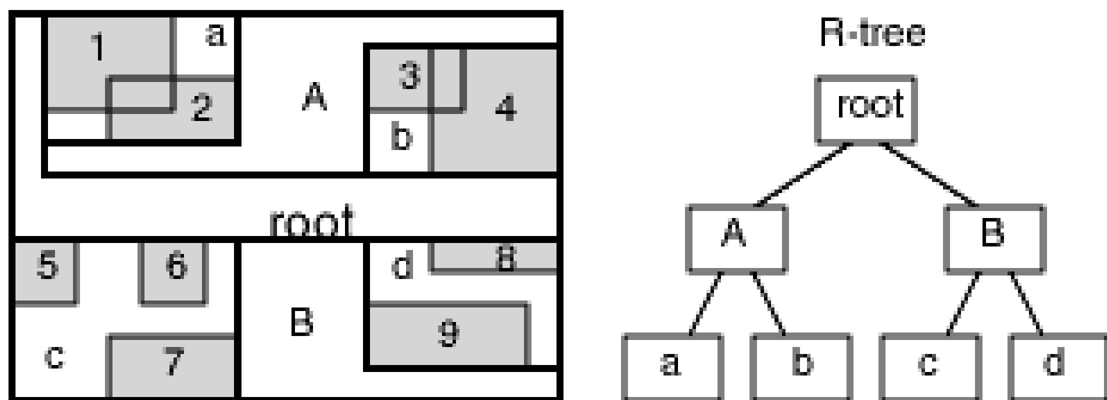
Κεφάλαιο 3

Πολυδιάστατες δομές δεδομένων και R-δέντρα

Οι δομές δεδομένων ευρετηρίου είναι πολύ δημοφιλείς στην οργάνωση δεδομένων των σκληρών δίσκων. Παρόλα αυτά απλές τέτοιες δομές δεν είναι κατάλληλες ώστε να εξυπηρετήσουν εφαρμογές που καταγράφουν γεωγραφικά δεδομένα. Για την καταγραφή τέτοιου είδους δεδομένων απαιτούνται πολυδιάστατες δομές ευρετηρίου(multidimensional indexes). Αυτού του είδους οι δομές περιέχουν τουλάχιστον δύο διαστάσεις που προσδιορίζουν τα φυσικά όρια των χαρακτηριστικών που αντιπροσωπεύονται από το ευρετήριο. Η παραδοσιακότερη πολυδιάστατη δομή ευρετηρίου είναι το R-δέντρο.

3.1Χαρακτηριστικά και δομή του R-δέντρου

Πρόκειται για ένα ισοζυγισμένο(balanced) δέντρο. Οι καταχωρήσεις ευρετηρίου αποθηκεύονται μόνο στους κόμβους φύλλα ενώ οι εσωτερικοί κόμβοι περιέχουν μόνο δείκτες ώστε να συγκροτούν συνδεδεμένη τη δομή. Το R-δέντρο είναι σχεδιασμένο έτσι ώστε η αναζήτηση ενός στοιχείου να απαιτεί την επίσκεψη μικρού αριθμών κόμβων. Είναι δυναμική δομή και μπορεί να υποστηρίξει αναμεμιγμένες πράξεις εισαγωγής ,διαγραφής και αναζήτησης χωρίς να απαιτεί αναδιάταξη. Τα δεδομένα εισάγονται στο δέντρο με τη μορφή πλειάδων. Κάθε καταχώρηση έχει μορφή : (παραλληλόγραμμο διάστασης n, αναγνωριστικό πλειάδας).Το αναγνωριστικό της κάθε πλειάδας είναι μοναδικό ενώ το παραλληλόγραμμο διάστασης n περιέχει όλα τα παραλληλόγραμμα των κόμβων παιδιών του. Στο σχήμα που ακολουθεί απεικονίζεται ένα R-δέντρο με ρίζα root,εσωτερικούς κόμβους A και B και κόμβους φύλλα τους a, b, c και d. Το δεξί μέρος της εικόνας αποτελεί τη κλασσική δενδρική απεικόνιση ενώ το αριστερό απεικόνιση με τη μορφή επικαλυπτόμενων παραλληλογράμμων.



Εικόνα 3.1 Απεικονίσεις R-δέντρου

Κάθε κόμβος μπορεί να έχει ένα μέγιστο αριθμό παιδιών ίσο με M και ελάχιστο αριθμό παιδιών ίσο με m<=M/2. Οπότε ένας εσωτερικός κόμβος θα έχει από m έως M κόμβους παιδιά ενώ ένας κόμβος φύλλο από m έως M καταχωρήσεις ευρετηρίου. Εκτός αν είναι κόμβος ρίζα.Ο κόμβος ρίζα πρέπει να έχει τουλάχιστον δύο παιδιά εκτός αν είναι κόμβος φύλλο. Τέλος, όλοι οι κόμβοι φύλλα πρέπει να βρίσκονται στο ίδιο επίπεδο.

3.2 Πράξεις στο R-δέντρο

Οι βασικές πράξεις που υποστηρίζονται είναι η αναζήτηση, η εισαγωγή, η διαγραφή και η ανανέωση ενός στοιχείου.

Ο αλγόριθμος ***αναζήτησης στοιχείου*** ξεκινά από τη ρίζα του δέντρου και κατευθύνεται προς τα φύλλα. Η είσοδος είναι ένα παραλληλόγραμμο. Ξεκινώντας από τη ρίζα και η αναζήτηση κατευθύνεται στους εσωτερικούς κόμβους που περιέχουν ένα σύνολο παραλληλογράμμων και δεικτών προς τα παιδιά τους. Για κάθε τέτοιο κόμβο πρέπει να αποφασιστεί αν επικαλύπτει το δοθέν παραλληλόγραμμο ή όχι. Αν ναι τότε η αναζήτηση συνεχίζεται με όμοιο τρόπο αναδρομικά στους κόμβους παιδιά. Όταν η αναζήτηση φτάσει σε κάποιο κόμβο φύλλο τότε εξετάζεται αν υπάρχει κάποια καταχώριση που περιέχει το παραλληλόγραμμο της εισόδου. Αν υπάρχει τότε η καταχώριση αυτή εισάγεται στο σύνολο των αποτελεσμάτων. Όταν εξεταστούν όσοι κόμβοι χρειαστεί με βάση τη παραπάνω διαδικασία τότε ο αλγόριθμος ολοκληρώνεται και το σύνολο των αποτελεσμάτων αποτελεί την έξοδο του αλγορίθμου αναζήτησης.

Ο αλγόριθμος ***εισαγωγής στοιχείου*** διασχίζει το δέντρο αναδρομικά ξεκινώντας από τη ρίζα. Η εισαγωγή του στοιχείου γίνεται πάντα σε κόμβο φύλλο και αν κόμβος υπερχειλίσει τότε γίνεται διάσπαση του κόμβου αυτού. Αν χρειαστεί η διάσπαση μεταφέρεται και στα πιο πάνω επίπεδα. Η είσοδος του αλγορίθμου είναι μια νέα καταχώριση για τη δομή του ευρετηρίου. Αρχικά ενεργοποιείται ειδικός αλγόριθμος που ξεκινά από τη ρίζα και εντοπίζει τον κόμβο φύλλο που θα χρειαστεί τη μικρότερη διεύρυνση ώστε να εισαχθεί σε αυτόν η νέα εγγραφή. Στη συνέχεια εισάγεται η καταχώριση. Αν ο κόμβος φύλλο έχει τώρα πάνω από Μ(μέγιστος αριθμός καταχωρίσεων – παιδιών) καταχωρίσεις τότε ο κόμβος αυτός θα υποστεί διάσπαση. Οι Μ+1 καταχωρίσεις πλέον θα μοιραστούν σε δύο κόμβους. Στην ουσία θα αφαιρεθούν κάποιες εγγραφές από τον παλιό υπερχειλισμένο κόμβο και θα προστεθούν στο νέο που θα δημιουργηθεί. Η διαδικασία αυτή δεν γίνεται τυχαία. Η διάσπαση πρέπει να γίνει με αποδοτικό τρόπο και να μη χρειαστεί να επισκεφθούν και οι δυο κόμβοι σε μελλοντικές αναζητήσεις. Υπάρχει πλήθος αλγορίθμων που επιτυγχάνουν αποδοτικές διασπάσεις. Η διάσπαση κόμβων φύλλων θα αυξήσει των αριθμό των παιδιών του πατρικού τους κόμβου. Αν αυτός γίνει μεγαλύτερος του Μ τότε θα χρειαστεί να διασπαστεί και ο πατρικός κόμβος. Υπάρχει περίπτωση να προκληθούν διασπάσεις σε όλα τα επίπεδα μέχρι και στην ίδια τη ρίζα. Οι διασπάσεις αυτές πραγματοποιούνται με όμοιο τρόπο με αυτόν των φύλλων. Ο αλγόριθμος ολοκληρώνεται όταν πραγματοποιηθεί η εγγραφή του νέου στοιχείου και ολοκληρωθούν οι όποιες διασπάσεις κόμβων μπορεί να προκύψουν.

Ο ***αλγόριθμος διαγραφής*** στοιχείου είναι κατά κάποιο τρόπο ο αντίθετος αυτού της εισαγωγής. Έχει κατεύθυνση από τη ρίζα προς τα φύλλα, καλεί αλγόριθμο ώστε να εντοπίσει το στοιχείο προς διαγραφή και μετά τη διαγραφή αν ο κόμβος έχει μικρό αριθμό εγγραφών(δηλαδή μικρότερο ή ίσο του Μ/2) τότε διαγράφεται ολόκληρος ο κόμβος και οι εγγραφές του διαμοιράζονται στους υπόλοιπους κόμβους.

Πιο αναλυτικά:

Η είσοδος στον αλγόριθμο είναι μια καταχώριση σαν να αυτές που έχουν εισαχθεί στο δέντρο. Καλείται αλγόριθμος που εντοπίζει τον κόμβο φύλλο που περιέχει την προς διαγραφή καταχώρηση. Αν δεν εντοπιστεί τότε τερματίζεται ο αλγόριθμος διαγραφής. Αν βρεθεί η καταχώριση τότε αυτή διαγράφεται από τον κόμβο φύλλο. Έπειτα εφαρμόζεται αλγόριθμος που εξετάζει αν ο κόμβος φύλλο που υπέστη διαγραφή στοιχείου έχει αριθμό εγγραφών m<=M. Αν ναι τότε θα πρέπει ο κόμβος αυτός να εξαλειφθεί και οι καταχωρίσεις να εισαχθούν σε άλλους κόμβους.

Η εισαγωγή των καταχωρίσεων γίνεται με βάση τον αλγόριθμο εισαγωγής στοιχείου που έχει περιγραφεί παραπάνω. Οι απαλοιφές κόμβων μπορεί να μεταφερθούν και στα πιο πάνω επίπεδα όπως συνέβαινε και με τις διασπάσεις κόμβων στον αλγόριθμο εισαγωγής. Τέλος αν ο κόμβος ρίζα έχει μόλις ένα παιδί μετά την ολοκλήρωση όλων των παραπάνω τότε θέτουμε το παιδί ως ρίζα του δέντρου. Αυτό ονομάζεται μείωση ύψους του δέντρου. Ο αλγόριθμος ολοκληρώνεται όταν εκτελεστούν όσα από τα παραπάνω βήματα κριθούν απαραίτητα κατά την εκτέλεση του.

Η ***ανανέωση στοιχείου*** πρόκειται για συνδυασμό διαγραφής και εισαγωγής. Αν ένα στοιχείο ανανεωθεί δηλαδή διαφοροποιηθεί το παραλληλόγραμμο του τότε η αντίστοιχη καταχώριση του δέντρου πρέπει να διαγραφεί, να ανανεωθεί και ανανεωμένη πλέον να εισαχθεί εκ νέου στο δέντρο. Η διαγραφή και η εισαγωγή γίνονται όπως έχουν ήδη περιγραφεί.

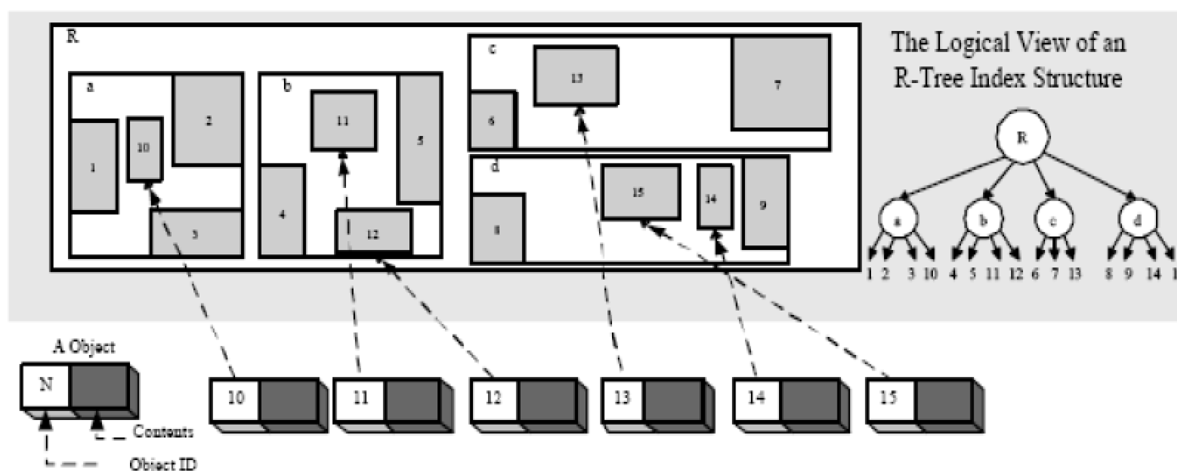
Κεφάλαιο 4

Μια αποδοτική υλοποίηση του R-δέντρου πάνω από το FTL

Στο κεφάλαιο αυτό παρουσιάζεται η μελέτη και υλοποίηση[6] ενός R-δέντρου τέτοιο ώστε να αποτελεί κατάλληλη μορφή για μνήμες τύπου φλας και κατ'επέκταση για δίσκους στερεάς κατάστασης.

4.1 Ανάγκες για μια αποδοτική υλοποίηση του R-δέντρου

Έστω ότι πρόκειται να εισαχθούν έξι νέα στοιχεία σε ένα R-δέντρο όπως φαίνεται στο παρακάτω σχήμα :



Εικόνα 4.1 Ένα R-δέντρο

Επίσης έστω πως κάθε κόμβος γράφεται ακριβώς σε μια σελίδα στο δίσκο. Έτσι, για να εισαχθούν τα έξι νέα στοιχεία στους κόμβους φύλλα a, b, c, d θα χρειαστούν έξι ανανεώσει.

Στις ανανεώσεις αυτές ακόμα και αν μεταβληθεί ένα μικρό μέρος του κόμβου θα πρέπει ο κόμβος αυτός να διαγραφεί και να εισαχθεί ξανά. Δηλαδή η σελίδα που ήταν γραμμένος θα γίνει μη έγκυρη και θα αναζητηθεί μια νέα σελίδα για τον κόμβο αυτό. Αυτό θα έχει ως αποτέλεσμα να καταναλώνονται γρήγορα οι ελεύθερες σελίδες και να ενεργοποιείται συχνά ο μηχανισμός συλλογής απορριμμάτων. Ο μηχανισμός αυτός προκαλεί αύξηση της απαιτούμενης ενέργειας αφού οι εισαγωγές και διαγραφές είναι σημαντικά πιο ενεργοβόρες από τις αναγνώσεις.

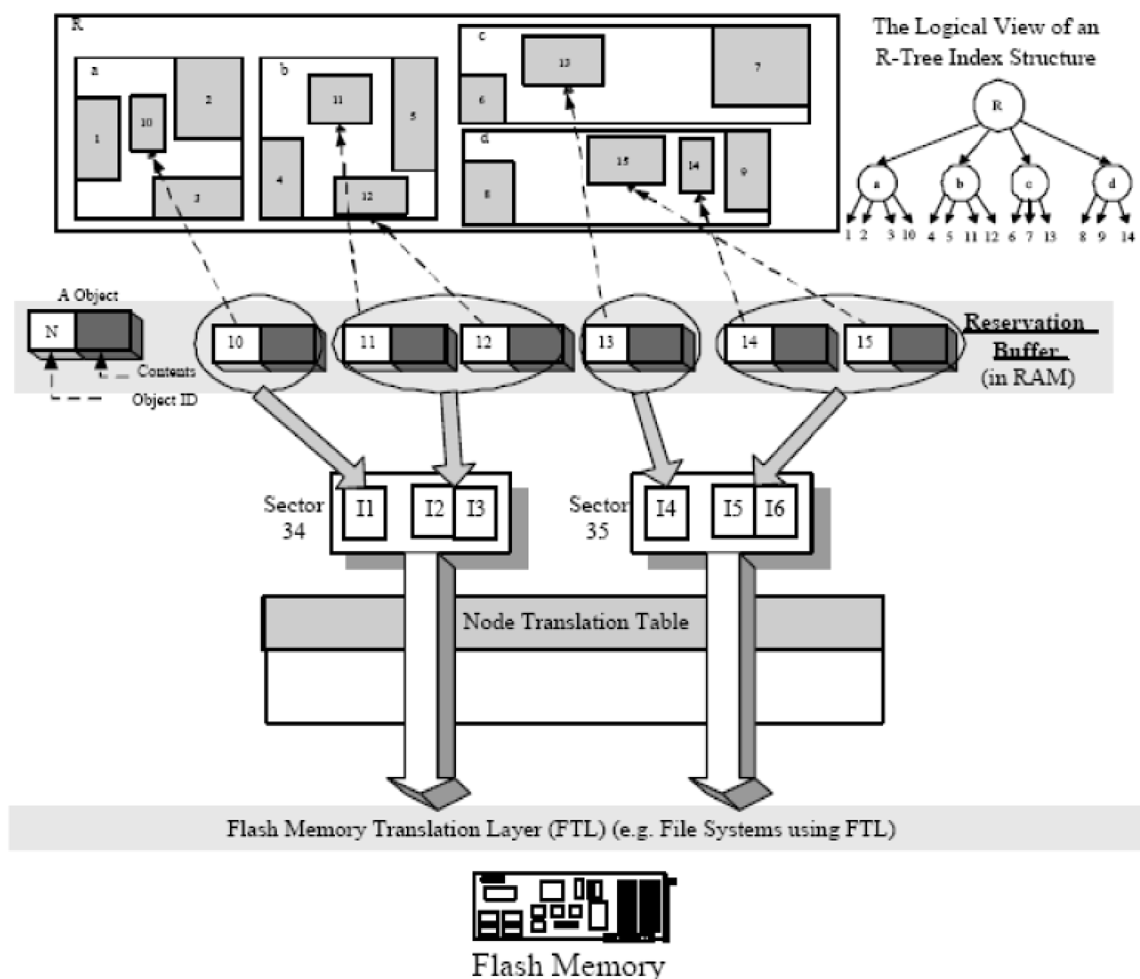
Επίσης οι πολλές διαγραφές φθείρουν τη μνήμη φλας και μειώνουν την αξιοπιστία της.

Η υλοποίηση που παρουσιάζεται παρακάτω έχει ως στόχο να βελτιώσει την απόδοση, να προφυλάξει τη μνήμη φλας και να μειώσει την κατανάλωση ενέργειας.

4.2 Η Υλοποίηση

Η υλοποίηση αυτή είναι ανεξάρτητη των εφαρμογών και του FTL. Το βασικό γνώρισμά της είναι πως αυξάνει τον αριθμό αναγνώσεων οι οποίες όμως είναι γρήγορες και με χαμηλή κατανάλωση ενέργειας ενώ αντίθετα μειώνει δραστικά τον αριθμό των εγγραφών οι οποίες έχουν μεγάλο κόστος τόσο σε χρόνο όσο και σε ενέργεια.
Όταν καλείται μια πράξη εισαγωγής ή διαγραφής στο δέντρο αυτή δεν εφαρμόζεται αμέσως αλλά διατηρείται στη κύρια μνήμη σε ένα πίνακα που λέγεται Reservation Buffer. Οι πράξεις σώζονται στο Reservation Buffer με την μορφή αντικειμένων. Κάθε αντικείμενο περιέχει το είδος της πράξης δηλαδή αν είναι εισαγωγή ή διαγραφή καθώς και τα δεδομένα που είναι προς εισαγωγή ή αφαίρεση. Το ίδιο συμβαίνει και για τις επόμενες πράξεις μέχρι να γεμίσει ο Reservation Buffer. Τότε μόνο γίνεται η εγγραφή των δεδομένων στο δίσκο. Όταν γίνεται η εγγραφή στο δίσκο τότε δημιουργείται μια νέα δομή που ονομάζεται index unit. Στην ουσία αυτή είναι που εγγράφεται. Η δομή αυτή τοποθετείται σε ένα τομέα στο δίσκο όπου τομέας είναι μια λογική σελίδα. Σε ένα τομέα πακετάρονται index units που ανήκουν σε διαφορετικούς κόμβους του R-δέντρου. Έτσι για να κατασκευαστεί κάποιος κόμβος του δέντρου απαιτείται και μια τρίτη δομή που θα υποδεικνύει σε ποιους τομείς βρίσκονται τα index units αυτού του κόμβου. Αυτή η δομή ονομάζεται Node Translation Table.

Η παραπάνω διαδικασία και οι τρείς κύριες δομές δεδομένων της απεικονίζονται στην ακόλουθη εικόνα:



Εικόνα 4.2 Η αρχιτεκτονική του συστήματος

Τα αντικείμενα 10, 11, 12, 13, 14, 15 που πρόκειται να εισαχθούν στο R-δέντρο αρχικά συγκεντρώνονται στον Reservation Buffer, έπειτα δημιουργούνται τα index units I1, I2, I3, I4, I5 και I6 τα οποία πακετάρονται σε δύο τομείς τους Sector34 και Sector35. Μετά ο Node Translation Table κρατά τις απαραίτητες αντιστοιχίσεις κόμβου και τομέα και τα δεδομένα γράφονται τελικά στη μνήμη φλας.

Όπως φαίνεται και από το σχήμα όλη αυτή η διαδικασία πραγματοποιείται πάνω από το FTL οπότε είναι ανεξάρτητη αυτού και μπορεί να χρησιμοποιηθεί αμέσως σε μεγάλο αριθμό συστημάτων.

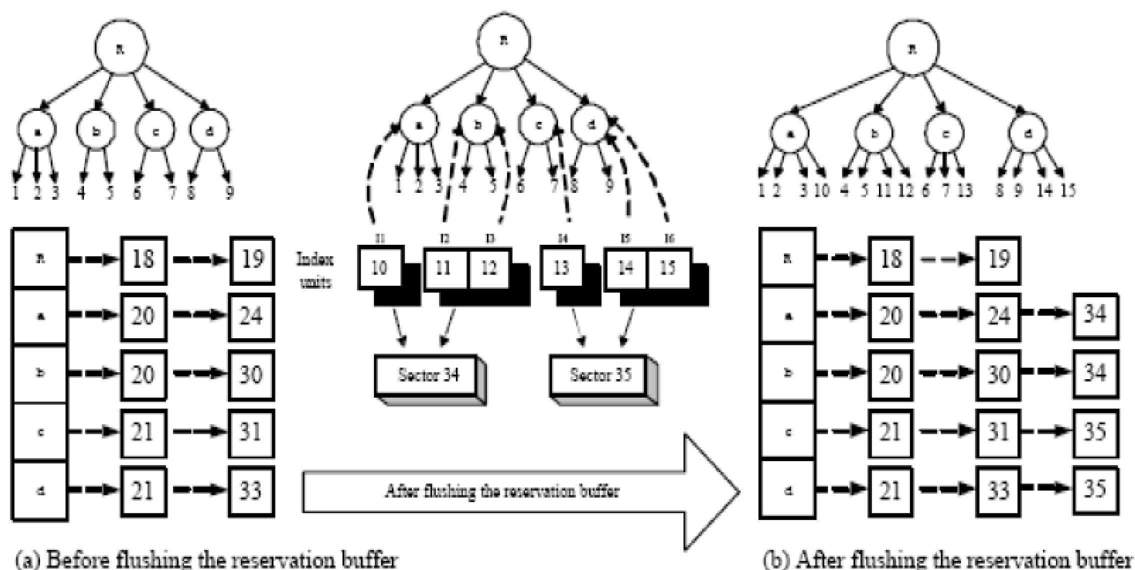
4.3 Δομές Δεδομένων

- ***Reservation Buffer:*** Πρόκειται για μια δομή που βρίσκεται στην κύρια μνήμη. Στην ουσία μπορεί να χαρακτηριστεί σαν ένα είδος αποθήκης που διατηρεί όλες τις πράξεις που καλούνται για εφαρμογή στο R-δέντρο. Διαθέτει προκαθορισμένο μέγεθος και όταν ο αριθμός των εισαχθέντων σ'αυτόν στοιχείων γίνει ίσος με το μέγεθος αυτό τότε αδειάζει και μεταφέρει τα στοιχεία για εγγραφή στο δίσκο. Όταν καλείται μια πράξη για το R-δέντρο τότε δημιουργείται ένα αντικείμενο-object που είναι αυτό που σώζεται στον Reservation Buffer. Έτσι στην ουσία τα αντικείμενα στο Buffer αναπαριστούν πράξεις που δεν έχουν ακόμα εφαρμοστεί στο δέντρο.

- ***Index Units:*** Στην υλοποίηση αυτή του R-δέντρου, η φυσική αναπαράσταση κάθε κόμβου αποτελείται από ένα σύνολο από index units. Κάθε index unit κατασκευάζεται όταν ένα αντικείμενο από το Reservation Buffer περνά μέσα στη δομή του R-δέντρου. Το Index unit περιέχει ένα σύνολο μετά-δεδομένων που δηλώνουν τις μεταβολές που έχουν γίνει σε κάποιο κόμβο του δέντρου. Πιο συγκεκριμένα περιέχει ένα δείκτη προς τον κόμβο πατέρα, ένα προς τον κόμβο παιδί, ένα δείκτη προς τα δεδομένα, μια μεταβλητή που συγκρατεί την πράξη(πρόσθεση, διαγραφή κτλ), το ελάχιστο επικαλύπτον παραλληλόγραμμο και το id του κόμβου. Όσον αφορά τη μεταβλητή op_flag δηλαδή αυτή που διατηρεί την πράξη παίρνει τιμή 'i' για εισαγωγή, 'd' για διαγραφή και 'u' για ανανέωση. Ένας προκαθορισμένος αριθμός από index units πακετάρεται κάθε φορά σε κάποιο τομέα και αποθηκεύεται στο δίσκο. Έτσι σε μια σελίδα τώρα δεν αποθηκεύεται μόνο μια πράξη άλλα ένα σύνολο από index units που στην ουσία ισοδυναμούν με πράξεις. Επίσης δεν διαγράφονται σελίδες απλά οι πράξεις διαγραφής γράφονται και αυτές ως index units στο δίσκο.

- ***Node Translation Table:*** Ο "πίνακας" αυτός δημιουργήθηκε για να διατηρεί μια αντιστοίχηση μεταξύ ενός κόμβου και τον index units που αναφέρονται στον κόμβο αυτό. Το πακετάρισμα index units διαφορετικών κόμβων μέσα στο ίδιο sector και ο διαμερισμός των index units σε πολλά sectors έκανε επιτακτική την ανάγκη δημιουργίας της δομής αυτής. Όταν ένας κόμβος θέλει να ανακατασκευαστεί διατρέχει στο Node Translation Table και βρίσκει σε ποια sectors στο δίσκο υπάρχουν index units δικά τους καθώς επίσης και σε ποιο σημείο μέσα στο sector βρίσκονται αυτά. Η υλοποίηση του Node Translation Table γίνεται από ένα πίνακα αποτελούμενο απο λίστες. Κάθε στοιχείου του πίνακα αντιστοιχεί σε ένα κόμβο του δέντρου και περιέχει μια λίστα που κάθε στοιχείο της είναι ένας δείκτης σε κάποιο sector.

Στο παρακάτω σχήμα αναπαρίσταται ένα R-δέντρο και ακριβώς από κάτω του ο Node Translation Table. Έχουμε από αριστερά προς τα δεξιά την περιγραφή της αρχικής κατάστασης, την εισαγωγή 6 index units σε δύο sectors και τέλος την τελική κατάσταση τόσο του δέντρου όσο και του Node Translation Table που αντιστοιχεί σ'αυτό :



Εικόνα 4.3 Η κατάσταση μετά την εγγραφή του Reservation Buffer

4.4 Το πακετάρισμα των Index Units

Όπως έχει αναφερθεί τα index units πακετάρονται σε τομείς όπου κάθε τομέας αντιστοιχεί σε μια σελίδα. Στο παραπάνω παράδειγμα φαίνεται πως σε κάθε τομέα τοποθετούνται τρία index units. Ο στόχος της υλοποίησης αυτής είναι να ελαχιστοποιήσει τον αριθμό των τομέων που θα γραφούν τελικά στη μνήμη φλας.

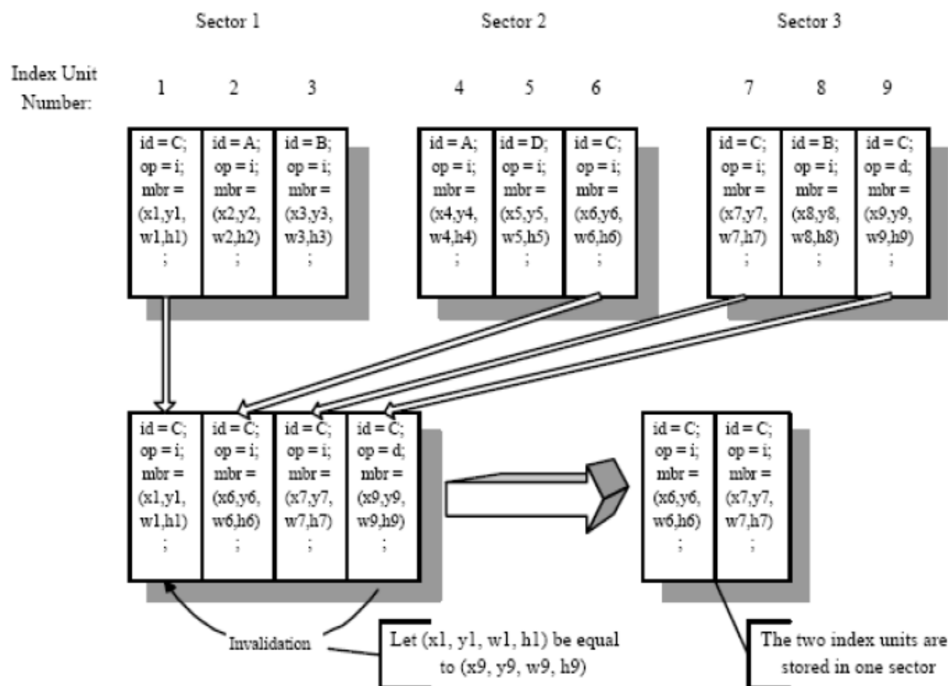
Έτσι το πρόβλημα στο πακετάρισμα των Index units διατυπώνεται ως εξής: "Δοθέντος ενός συνόλου T από ξένα index units, μια χωρητικότητα C κάθε τομέα και ενός θετικού ακεραίου I θα πρέπει να βρεθεί μια διαμέριση του T σε ομάδες τέτοια ώστε ο αριθμός των index units στις ομάδες αυτές να μην ξεπερνά το C ενώ ο αριθμός των ομάδων να μην ξεπερνά το I."

Το πρόβλημα αυτό μπορεί να αναχθεί στο Bin Packing Problem[7]. Έτσι αν υπάρχει λύση για το πακετάρισμα των index units αυτή θα μπορεί να δοθεί από το Bin Packing Problem. Το πρόβλημα αυτό είναι NP-Hard[8].

4.5 Συμπίεση των κόμβων του R-δέντρου

Κάθε κόμβος του R-δέντρου απαρτίζεται από ένα σύνολο index units διασκορπισμένα σε διάφορους τομείς. Ο Node Translation Table κρατά την αντιστοίχηση κόμβων και τομέων διατηρώντας μια λίστα για κάθε κόμβο. Όμως αν σε ένα κόμβο γίνουν πολλές εισαγωγές και διαγραφές τότε η λίστα αυτή θα γίνει υπερβολικά μεγάλη. Δημιουργείται έτσι η ανάγκη για συμπίεση της λίστας αν αυτή ξεπεράσει ένα συγκεκριμένο αριθμό w. Η συμπίεση πραγματοποιείται όταν εντοπισθούν στη λίστα δυο πράξεις που η μία να αναιρεί την άλλη. Δηλαδή αν υπάρχει μια πράξη εισαγωγής σε ένα κόμβο και υπάρχει και μια πράξη διαγραφής στον ίδιο κόμβο με όμοιο id και ελάχιστο επικαλύπτον παραλληλόγραμμο τότε αυτές οι δυο πράξεις μπορούν να σβηστούν από το Node Translation Table. Έτσι μειώνεται το μήκος των λιστών του και οι αντίστοιχοι κόμβοι συμπιέζονται. Το αποτέλεσμα της συμπίεσης αυτής είναι διπλό. Από τη μία μειώνει το μέγεθος του Node Translation Table και από την άλλη μειώνει σημαντικά τον αριθμό των πράξεων που εφαρμόζονται στο δέντρο άσκοπα σε κάθε ανακατασκευή ενός κόμβου.

Στο παρακάτω σχήμα απεικονίζεται μια τέτοια συμπίεση:



Εικόνα 4.4 Η σύμπτυξη ενός κόμβου

4.6 Ανάλυση Πολυπλοκότητας

Έστω ένα R-δέντρο με ύψος H, στο οποίο θα εισαχθούν n στοιχεία με διαφορετικά ελάχιστα παραλληλόγραμμα. Κάθε κόμβος αποθηκεύεται σε μια σελίδα στο δίσκο ενώ m είναι ο αριθμός των στοιχείων που προϋπάρχουν στο δέντρο(δηλαδή πριν την εισαγωγή των επόμενων n στοιχείων). Το ύψος H οριοθετείται από τη σχέση $O(log_{fan_out}(m+n))$. Ο αριθμός των αναγνώσεων για τις n εισαγωγές είναι $R_R = O(n*H)$. Ενώ ο αριθμός των αντίστοιχων εγγραφών είναι $W_R = O(n+3*N_{Splits})$ όπου N_{Splits} είναι ο αριθμός των διασπάσεων που προκαλούνται κατά την εισαγωγή των n στοιχείων. Όλα τα παραπάνω αφορούν την υλοποίηση του κλασσικού R-δέντρου. Στην προτεινόμενη υλοποίηση του R-δέντρου ο **αριθμός των αναγνώσεων** είναι $R_{PR} = O(n*H*w)$, όπου w ο μέγιστος αριθμός στοιχείων της λίστας στο Node Translation Table. Ο αριθμός αυτός προκύπτει φυσιολογικά αφού κάθε φορά που χτίζεται ένας κόμβος γίνεται επίσκεψη στη λίστα με μέγιστο μέγεθος w. Συγκρίνοντας τους R_R , R_{PR} φαίνεται πως ο αριθμός των αναγνώσεων στην προτεινόμενη υλοποίηση είναι μεγαλύτερος από τον αντίστοιχο της κλασσικής.

Ο υπολογισμός του **αριθμού των εγγραφών** της προτεινόμενης υλοποίησης είναι δυσκολότερος και τα βασικά σημεία του υπολογισμού είναι τα ακόλουθα:

- Αν b το μέγεθος του Reservation Buffer τότε αυτός θα εγγραφεί το λιγότερο $\lceil n/b \rceil$ φορές.
- Σε κάθε εγγραφή του Reservation Buffer προκύπτουν (b +N_{Splits}*(fanout-1)+ N_{Splits}*2) Index Units. Τα προηγούμενα προκύπτουν από το ότι κάθε διάσπαση θα επιφέρει αλλαγές σε δύο κόμβους ενώ ο αριθμός των Index Units στους δύο αυτούς κόμβους θα είναι fanout-1. Το 2*fanout είναι η χειρότερη περίπτωση.
- Λ = fanout-1

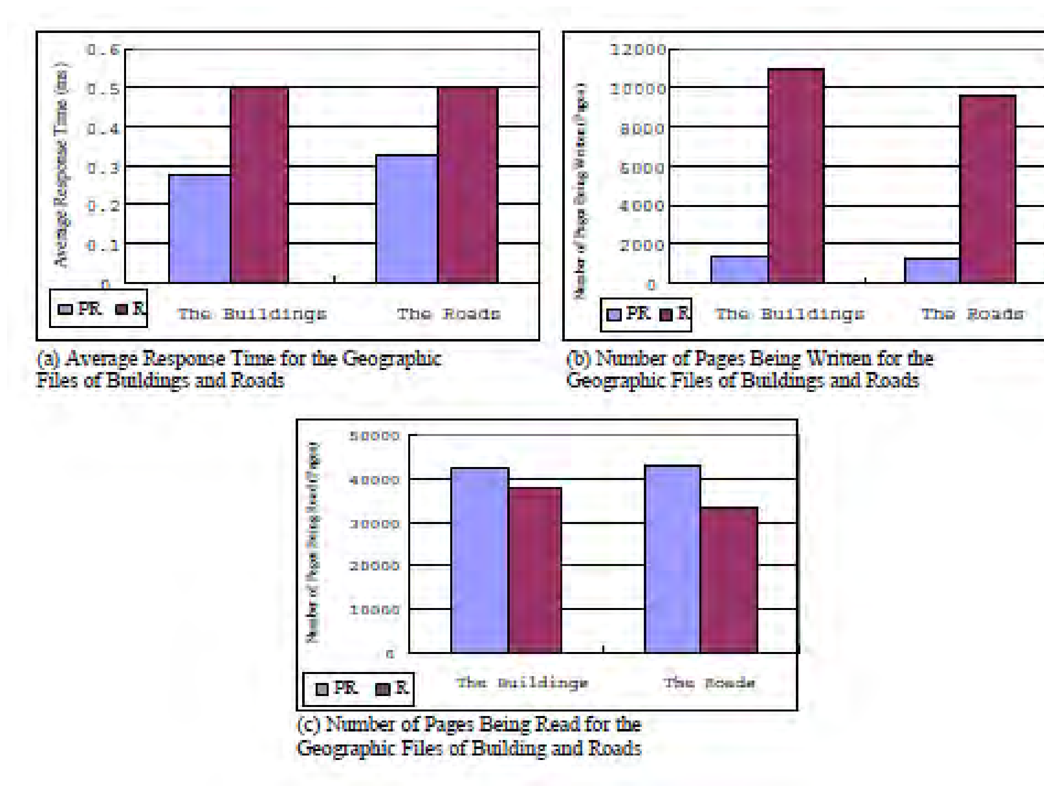
Τελικά προκύπτει πως ο $W_{PR} = O(2*(^{\lceil x/b \rceil} b/Λ)+ N_{Splits}) = O((2*n)/Λ + N_{Splits})$ Από τα παραπάνω προκύπτει πως ο αριθμός εγγραφών W_{PR} της νέας υλοποίησης είναι σημαντικά μικρότερος από τον W_R της κλασσικής.

Τέλος η προσπάθεια για συμπίεση στο Node Translation Table μπορεί να προσφέρει μια αύξηση του χρόνου τρεξίματος καθώς θα εισάγει επιπλέον καθυστερήσεις που εξαρτώνται από τα μεγέθη των λιστών που θα πρέπει να διαβαστούν καθώς και από τον αριθμό των τομέων.

4.7 Πειραματικά Αποτελέσματα

Τα πειράματα εκτελέστηκαν σε ένα σύστημα εφοδιασμένο με 4MB NAND μνήμη φλας και αφορούσαν δρόμους και κτίρια στην Ταϊπέι . Ο Reservation Buffer είχε μέγεθος 80 στοιχεία και το fan-out ήταν 16.

Τα σημαντικότερα αποτελέσματα απεικονίζονται στα διαγράμματα που ακολουθούν(PR η παρούσα υλοποίηση και R η κλασσική):



(a) Average Response Time for the Geographic Files of Buildings and Roads

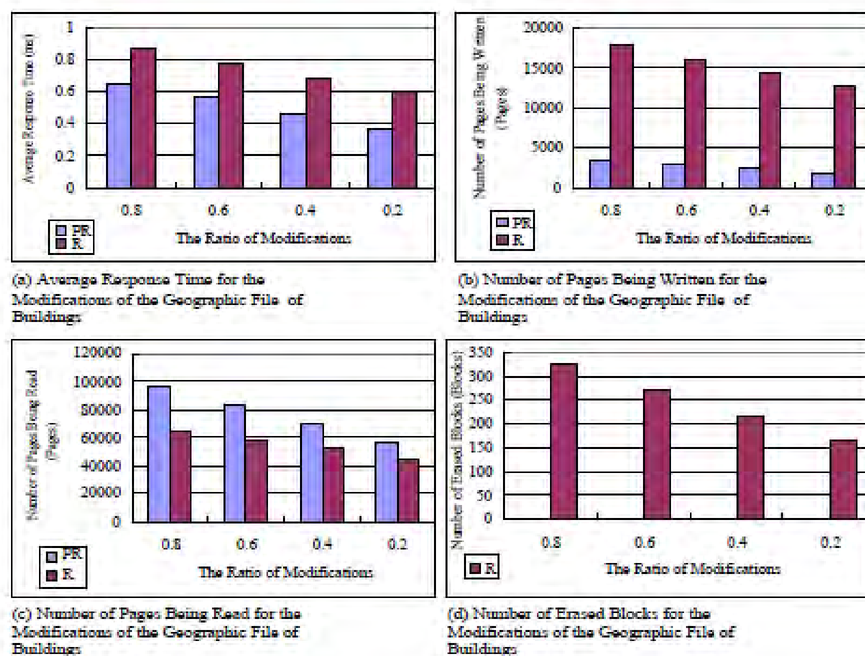
(b) Number of Pages Being Written for the Geographic Files of Buildings and Roads

(c) Number of Pages Being Read for the Geographic Files of Building and Roads

Εικόνα 4.5 Αποτελέσματα της κατασκευής του R-δέντρου

Στα παραπάνω σχήματα φαίνεται πως η προτεινόμενη υλοποίηση μείωσε σημαντικά το χρόνο απόκρισης κατά την κατασκευή του δέντρου ενώ σχεδόν εξαφάνισε τον αριθμό σελίδων που εγγράφονται. Σε αντίθεση όμως με τα παραπάνω αύξησε τον αριθμό των αναγνώσεων. Παρόλα αυτά η αύξηση αυτή είναι πολύ μικρή σε σχέση με τον αριθμό αναγνώσεων του παραδοσιακού R-δέντρου.

Πέρα αυτών των πειραμάτων που αφορούσαν τη κατασκευή του αρχείου των κτιρίων και των δρόμων πραγματοποιήθηκαν και τα παρακάτω που αφορούν μεταβολές στο αρχείο αυτό:



Εικόνα 4.6 Η απόδοση του συστήματος σε μεταβολές

Τα αποτελέσματα είναι εντελώς όμοια με αυτά της περίπτωσης του “χτισίματος” του αρχείου. Μείωση του χρόνου απόκρισης και του αριθμού εγγραφών και μικρή αύξηση του αριθμού των αναγνώσεων.

Τέλος όσον αφορά την ενεργειακή κατανάλωση ήταν απαραίτητη η εύρεση ενός κατάλληλου Reservation Buffer. Καθώς αν το μέγεθος του ήταν αρκετά μεγάλο αυτό θα αύξανε τις ενεργειακές απαιτήσεις. Τελικά η κατανάλωση στο PR ήταν μικρότερη συγκριτικά με το R όπως φαίνεται και στον ακόλουθο πίνακα. Το σύστημα που χρησιμοποιήθηκε είναι όμοιο με των υπόλοιπων πειραμάτων.

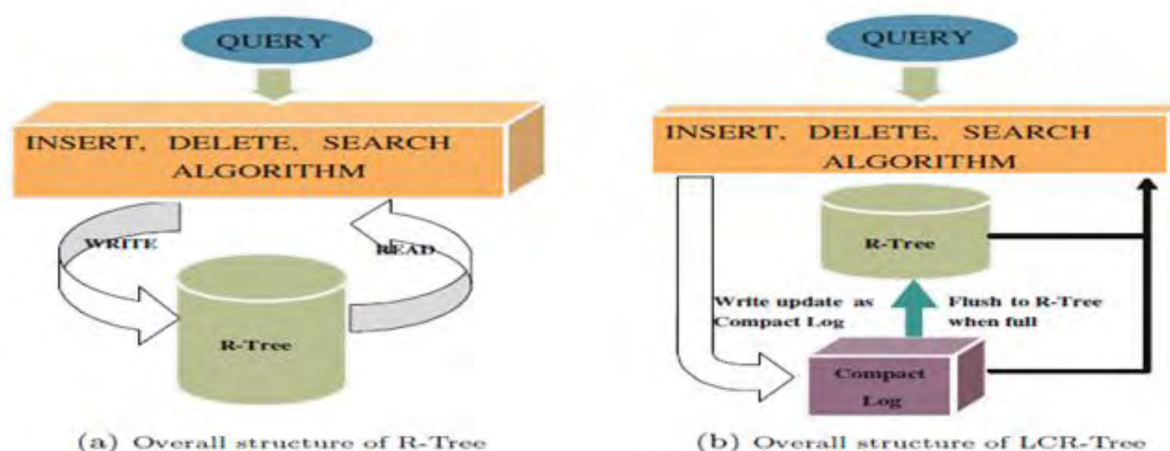
Creation		
	The Proposed R-Tree	The Original R-Tree
The Buildings	4.52	6.43
The Roads	4.55	5.63

Πίνακας 4.1 Η ενεργειακή κατανάλωση

4.8 Συγγενείς Μελέτες και Υλοποιήσεις

LCR-Tree(Log-Compact R-Tree)[9]

Βασική ιδέα αποτελεί το γεγονός ότι συγκεντρώνει όλα τα logs που αφορούν έναν κόμβο και τα τοποθετεί σε μια σελίδα που ονομάζεται compact log. Η σελίδα αυτή βρίσκεται σε μια συνεχόμενη περιοχή στο δίσκο. Επιπροσθέτως, όπου είναι εφικτό, συγχωνεύει τα compact logs που αφορούν διαφορετικούς κόμβους σε μία σελίδα.



Εικόνα 4.7 Σύγκριση LCR-δέντρου με R-δέντρο

Το LCR-Tree αποτελείται από δύο μέρη: το tree και το log. Το tree μέρος είναι ένα τυπικό R-Tree στο οποίο βρίσκονται όλα τα δεδομένα του R-Tree. Στο log τμήμα καταγράφονται οι ενημερώσεις των κόμβων του δένδρου προκειμένου να μειωθούν οι τυχαίες εγγραφές στη δομή. Όταν το log τμήμα γεμίσει, συγχωνεύεται στο R-Tree. Στην κύρια μνήμη διατηρείται ένας πίνακας αντιστοιχίσεων προς διευκόλυνση της εύρεσης των log ενός κόμβου. Όταν γίνεται μία ενημέρωση κόμβου, διαβάζεται το ήδη υπάρχον log και συγχωνεύεται με την ενημέρωση αυτή(compact log). Επισημαίνεται πως για να λάβει κανείς ενημέρωση για κάποιον κόμβο θα πρέπει να κοιτάξει τόσο το compact log όσο και το R-Tree.

Κεφάλαιο 5

Η Υλοποίηση

5.1 Η βάση της υλοποίησης

Η υλοποίηση της παρούσας εργασίας στηρίζεται στον κώδικα του R-δέντρου που πρώτος ανέπτυξε ο κ.Greg Douglas το 2004 και έπειτα επεξεργάστηκε το 2011 ο κ. Yariv Barkan. Στην ουσία έχει χρησιμοποιηθεί η επεξεργασμένη υλοποίηση του δεύτερου. Πρόκειται για μια υλοποίηση σε γλώσσα C++ και πιο συγκεκριμένα είναι templated [10]. Υποστηρίζει δισδιάστατες και τρισδιάστατες δομές ευρετηρίου και ο τύπος των δεδομένων προς εγγραφή μπορεί να καθοριστεί από το χρήστη κατά τη δημιουργία του δέντρου. Οι πράξεις που μπορεί να εκτελέσει ο χρήστης είναι η εισαγωγή και η διαγραφή μιας εγγραφής καθώς και η αναζήτηση των εγγραφών που περιέχονται σε ένα συγκεκριμένο επικαλύπτον παραλληλόγραμμο.

5.2 Παραδοχές και συμβάσεις

- Η παραλλαγή του R-δέντρου θα καλείται ως RFTL.
- Το RFTL όπως και το αρχικό R-δέντρο δεν διαθέτουν πρόβλεψη για ακατάλληλη είσοδο από το χρήστη και τερματίζουν αιφνιδίως.
- Το μέγεθος του εγγράψιμου μπλοκ στο δίσκο έχει οριστεί στο 1K.
- Η υλοποίηση του RFTL στηρίζεται στη μελέτη του προηγούμενου κεφαλαίου.
- Βασική διαφορά είναι πως το RFTL υιοθετεί τα χαρακτηριστικά της μελέτης μόνο για τους κόμβους φύλλα στους οποίους γίνονται ουσιαστικά όλες οι πράξεις που αφορούν δεδομένα.
- Για τους εσωτερικούς κόμβους διατηρείται η δομή του R-δέντρου.
- Το RFTL μπορεί να αποθηκευτεί, να φορτωθεί και να επεξεργαστεί εκ νέου.

5.3 Λεπτομέρειες της υλοποίησης

Στην ενότητα αυτή παρουσιάζονται τα βασικότερα στοιχεία της υλοποίησης που αφορούν δομές δεδομένων και συναρτήσεις.

Δομές Δεδομένων:

1. Ο ***ResBuffer*** πρόκειται για τη δομή που αποθηκεύει τις πράξεις που επρόκειτο να εφαρμοστούν στο RFTL. Έχεις υλοποιηθεί ως μια δομή-struct με πεδία τα ορθογώνια παραλληλόγραμμα(min&max), το id του στοιχείου που αφορά η πράξη καθώς και μια μεταβλητή(op) που περιέχει το είδος της πράξης. Αν είναι δηλαδή εισαγωγή ή διαγραφή. Τέλος περιέχει και μια συνάρτηση μέλος(FillResBuffer) που ουσιαστικά συμπληρώνει το ResBuffer.

2. Η δομή ***IndexUnit*** δημιουργείται μόλις μια πράξη που είναι αποθηκευμένη στο ResBuffer εφαρμοστεί στο RFTL. Περιέχει αντίστοιχα πεδία με τον ResBuffer συν ένα πεδίο που περιέχει τον κόμβο στον οποίο αναφέρεται η πράξη στο IndexUnit. Μόλις συμπληρωθούν όλα τα πεδία του εισάγεται σε μια λίστα. Η εισαγωγή στη λίστα γίνεται με first fit αλγόριθμο. Όταν το μήκος της λίστας φτάσει σε ένα καθορισμένο αριθμό(IUNITPACKAGE) τότε αυτή εγγράφεται στο δίσκο.

3. Όταν ένα σύνολο από IndexUnits εγγράφεται στο δίσκο τότε το σύνολο αυτό αποτελεί μια νέα εισαγωγή στο ***NodeTransTable***. Ο πίνακας αυτός έχει υλοποιηθεί ως μια δομή multimap[11]. Το κλειδί(key) της δομής αποτελεί ένα ακέραιος που είναι το αναγνωριστικό του κόμβου που αφορούν οι πράξεις στα IndexUnits ενώ ως τιμές περιέχει ένα ζευγάρι ακεραίων. Ο πρώτος ακέραιος είναι η απόσταση του συνόλου των IndexUnits από την αρχή του αρχείου ενώ ο δεύτερος θέση ενός IndexUnit μέσα στο σύνολο.

Συναρτήσεις:

1. Η συνάρτηση ***Save*** εγγράφει στο δίσκο τη λίστα με τα IndexUnits που δέχεται ως όρισμα και δημιουργεί αντίστοιχη εγγραφή στο NodeTranslTable.

Algorithm 1 Save (list IndexUnits)

```
1.  Open File
2.  Buffer[PAGESIZE]
3.  For each IndexUnit in list
4.      Write Node_name to Buffer
5.      Write id to Buffer
6.      Write Rectangle to Buffer
7.      Write operation to Buffer
8.      First Table value = page_id
9.      Second Table value = iterator
10.     Insert (key=Node_name ,T value =( First Table value, Second Table value))
        to NodeTransTable
11.     Iterator++
12. End for
13. Seek = page_id * PAGESIZE
14. Page_id++
15. Move for page_id from File beginning
16. Write buffer to File
17. Close File
```

2. Όταν ο χρήστης καλεί τις Insert και Remove τότε αυτές οι πράξεις δεν εκτελούνται άμεσα αλλά αποθηκεύονται στο ResBuffer. Όταν αυτός γεμίσει τότε θα εκτελεστούν οι πράξεις με τη σειρά καλώντας αντίστοιχα τις ***InsertRect*** και RemoveRect. Επειδή οι διαγραφές αντιμετωπίζονται στο RFTL ως εισαγωγές αρκεί να παρουσιαστεί η InsertRect και πιο συγκεκριμένα η InsertRectRec που καλείται αναδρομικά από τη πρώτη και εισάγει μια εγγραφή στο δέντρο. Αν είναι εισαγωγή μιας νέας εγγραφής τότε απλά αρκεί να βρεθεί αναδρομικά ο κόμβος φύλλο που θα εισαχθεί ώστε ο κόμβος αυτός να εγγραφή στο τρέχον IndexUnit. Αν είναι όμως εισαγωγή εγγραφής από κάποιο κόμβο που έχει διαγραφή εξαιτίας λίγων εγγραφών τότε πρέπει να αλλαχθούν οι εγγραφές στο NodeTransTable για το κόμβο αυτό.

Algorithm 2 InsertRectRec(rectangle, id, node, newnode, level, iucount, interval)

```
1.  If node->level > level
2.      Call InsertRectRec recursively because node isn't leaf
3.  End If
4.  Else If node->level == level
5.      If is a user insert
6.          ++counter for IndexUnit inside list
7.          Read Node_name
8.      Else If is a reinsertion
9.          Open File
10.         Read NodeTransTable elements for key == interval
```

```
11.         For Each of these elements
12.           If element.rectangle == currentbranch.rectangle
               && element.id ==   currentbranch.id
13.             Erase this element from NodeTransTable
14.             Insert this element in NodeTransTable with key = node
15.             Find element in NodeTransTable is true
16.           End If
17.         End for
18.         If don't find element search it in IndexUnits list
19.           Listcounter = 1
20.           For Each IndexUnit in list
21.             If Listcounter ==  counter for currentbranch
22.               Make IndexUnit id = newnode id
23.               Find element in list is true
24.             End if
25.             Listcounter++
26.           End for
27.         End if
28.         If didn't find element && is a user insertion
29.           Node_name = newnode name
30.         End if
31.     End else if
32.     Close File
33.   End else if
34.  Add node
```

3. Όταν ένας κόμβος αποκτήσει παραπάνω από MAXNODES εγγραφές τότε θα πρέπει να διασπαστεί και κάποιες από τις εγγραφές του να εισαχθούν στο νέο κόμβο που δημιουργείται. Έτσι θα πρέπει να αλλαχθούν οι αντίστοιχες εγγραφές στο NodeTransTable και στα μη αποθηκευμένα IndexUnits στη λίστα.

Algorithm 3 SplitNode(node, branch, newnode)

```
1.  Get branches of node
2.  Choose Partition of node branches
3.  Load branches
4.  If node is Leaf
5.    Open File
6.    For each branch
7.      If branch belongs to newnode
8.        Read NodeTransTable elements for key == node id
9.        For Each of these elements
10.         If element.rectangle == currentbranch.rectangle
             && element.id ==   currentbranch.id
```

```
11.                 Erase this element from NodeTransTable
12.                 Insert this element in NodeTransTable with key = newnode
13.                 Find element in NodeTransTable is true
14.               End If
15.             End for
16.             If don't find element search it in IndexUnits list
17.             Listcounter = 1
18.             For Each IndexUnit in list
19.                 If Listcounter ==  counter for currentbranch
20.                   Make IndexUnit id = newnode id
21.                   Find element in list is true
22.                End if
23.               Listcounter++
24.             End for
25.           End if
26.           If didn't find element
27.             Node_name = newnode name
28.           End if
29.         End if
30.       End for
31.     Close File
32.   End if
```

4. Όταν ζητηθούν οι εγγραφές ενός κόμβου φύλλο τότε καλείται η συνάρτηση ***ConstactNode*** η οποία διαβάζει τις εγγραφές που υπάρχουν για το κόμβο αυτό στο NodeTransTable καθώς και στη λίστα των IndexUnits που δεν έχουν ακόμη αποθηκευτεί και δημιουργεί τον κόμβο.

Algorithm 4 ConstractNode(node_to_constract, node id)

```
1.  Open File
2.  Read NodeTransTable elements for key == node id
3.  For each of this elements
4.      If operation = insert
5.         Put element to Insertion list
6.      End if
7.      Else if operation = remove
8.         Put element to remove list
9.      End else if
10. End for
11. For each element in IndexUnits list
12.     If element id == node id
13.        If operation = insert
```

```
14.          Put element to insert list
15.       End if
16.      Else if operation == remove
17.               Put element to remove  list
18.      End else if
19.    End if
20.  End for
21.  For each element in remove list
22.      For each element in insert list
23.          If insert element == remove element
24.             Delete insert element
25.          End if
26.      End for
27.  End for
28.  For each element in insert list
29.      Write this element to node_to_constract
30.  End for
31.  Close File
```

Κεφάλαιο 6

Πειραματικά Αποτελέσματα

6.1 Προδιαγραφές συστήματος και πειραμάτων

Τα πειράματα εκτελέστηκαν σε σύστημα με τα ακόλουθα χαρακτηριστικά:

- Επεξεργαστής : Intel Core2 Quad Q8200 2.33GHz
- Μνήμη Ram : 8GB DDR3 συχνότητας 1333 MHz
- Λειτουργικό Σύστημα : Windows 8 32-bit
- Σκληρός Δίσκος : SSD OCZ Agility 3 120GB με ταχύτητα ανάγνωσης έως 525MB/s και ταχύτητας εγγραφής έως 500MB/s

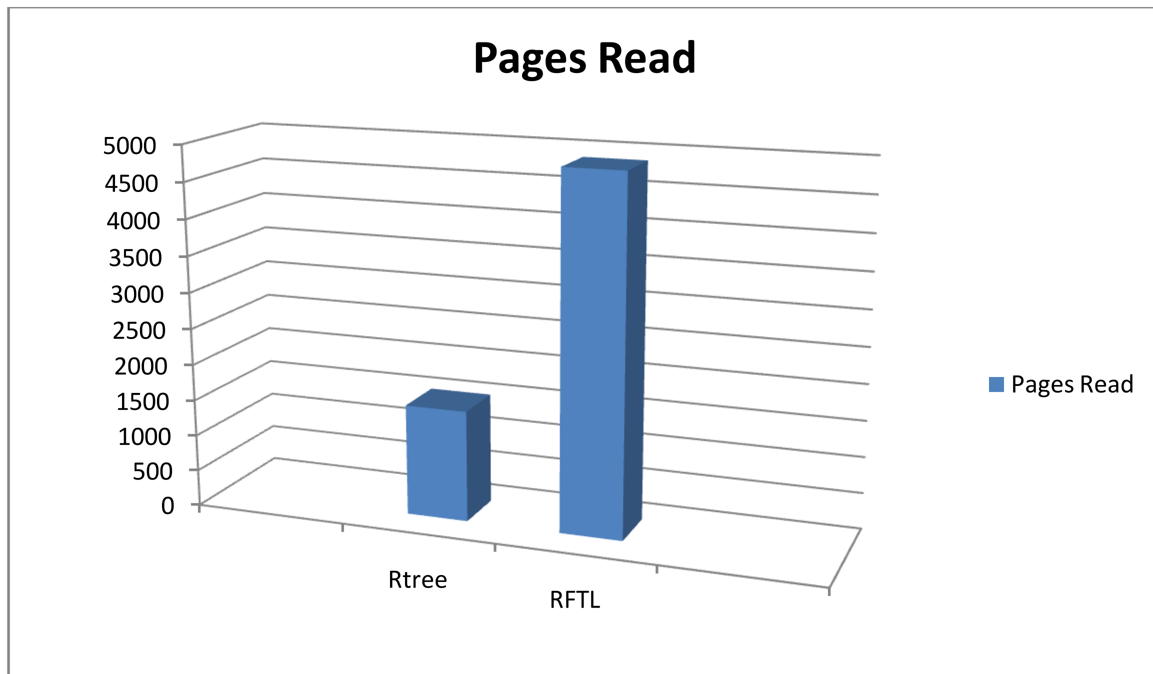
Οι παράμετροι του RFTL αφορούσαν το μέγεθος του ResBuffer να είναι 10 , τον αριθμό των IndexUnits που θα αποθηκεύεται σε ένα μπλοκ στο δίσκο να είναι 5, το μέγιστο αριθμό παιδιών ή fan-out ενός κόμβου να είναι 8 ενώ δεν έχει ληφθεί κάποιος περιορισμός για το μήκος της λίστας των στοιχείων στο NodeTransTable.

Τα δεδομένα εισάγονται ή αφαιρούνται με τη μορφή πλειάδων. Κάθε πλειάδα είναι της μορφής : ((xmin , ymin, xmax, ymax), id) όπου τα x,y δίνουν τις διαστάσεις του ορθογωνίου παραλληλογράμμου και το id αναφέρεται στο αναγνωριστικό της εγγραφής. Οι διαστάσεις έχουν δημιουργηθεί τυχαία τηρώντας μόνο τις συνθήκες $xmin < xmax$ και $ymin < ymax$. Το id δίνεται σειριακά καθώς καλούνται πράξεις εισαγωγής και διαγραφής μέσα από σε κάποια δομή επανάληψης. Το αρχείο με τις εγγραφές που χρησιμοποιήθηκε στα πειράματα προέρχεται από το chorochronos.org[12].

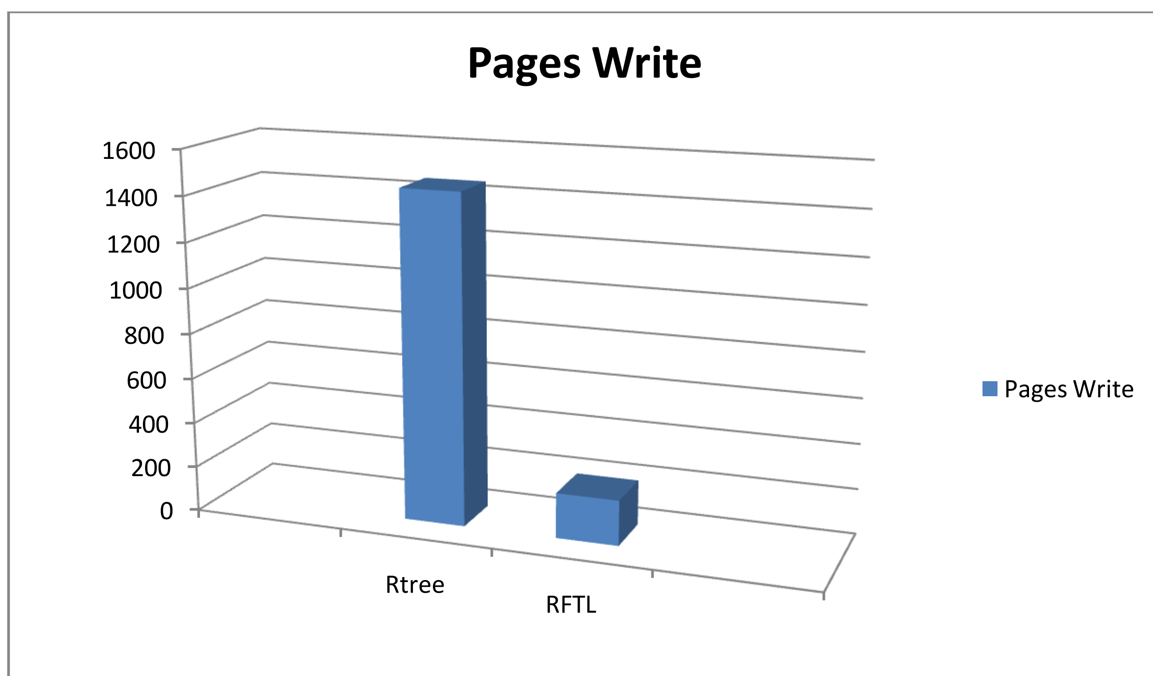
6.2 Πειράματα

Πείραμα $1^{ο}$:

Στο πείραμα αυτό πραγματοποιείται η κατασκευή του δέντρου με την εισαγωγή 1000 εγγραφών. Τα αποτελέσματα απεικονίζονται στο επόμενο διάγραμμα:



Εικόνα 6.1 Ο αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 1



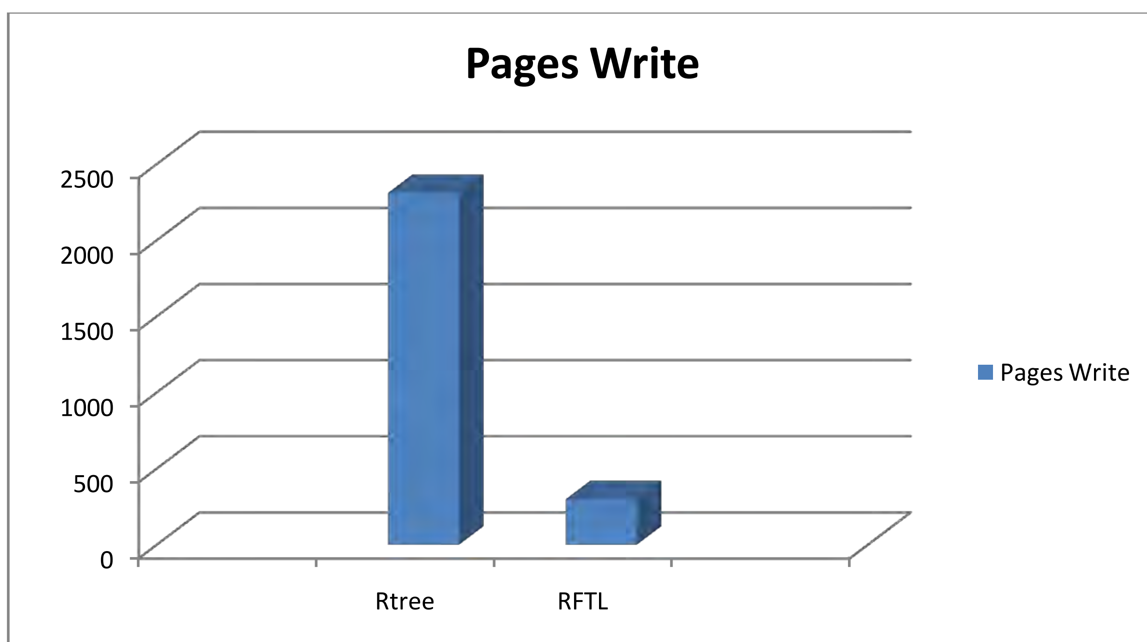
Εικόνα 6.2 Ο αριθμός σελίδων του δίσκου που εγράφησαν στο Πείραμα 1

Πείραμα 2°:

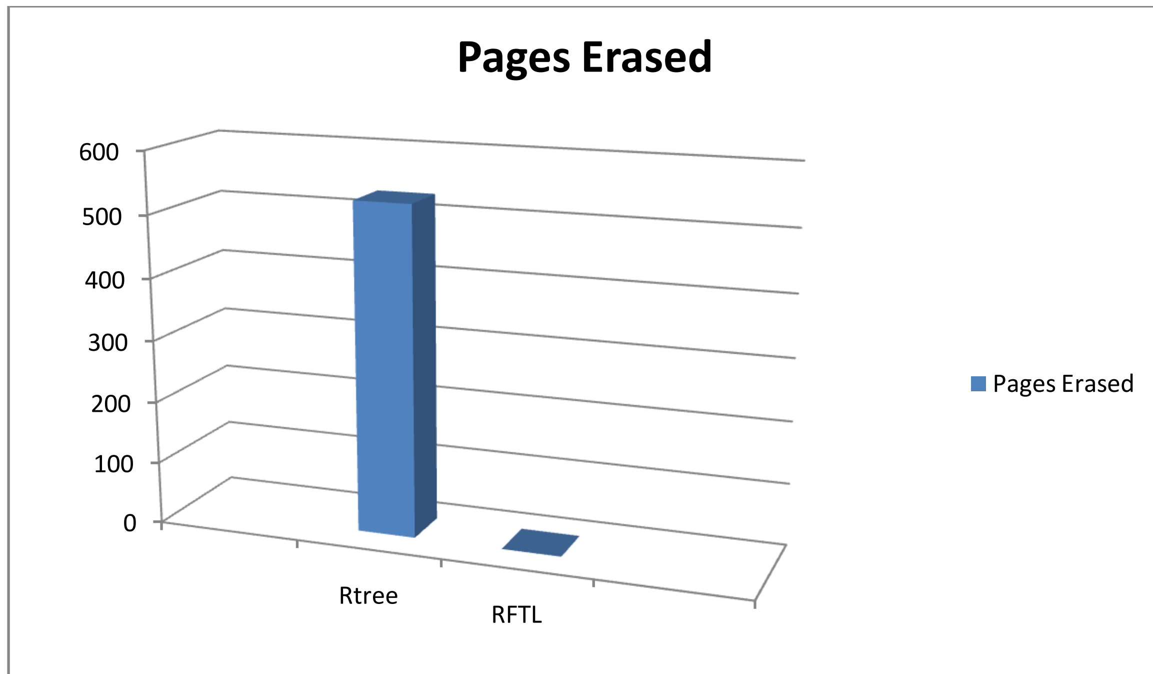
Στο πείραμα αυτό πραγματοποιήθηκαν αρχικά οι 1000 εγγραφές του πειράματος 1 και προστέθηκαν αναμεμειγμένες 400 διαγραφές και 90 εισαγωγές στο δέντρο που είχε κατασκευαστεί στο πείραμα 1.

Τα αποτελέσματα ήταν:

Εικόνα 6.3 Ο αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 2



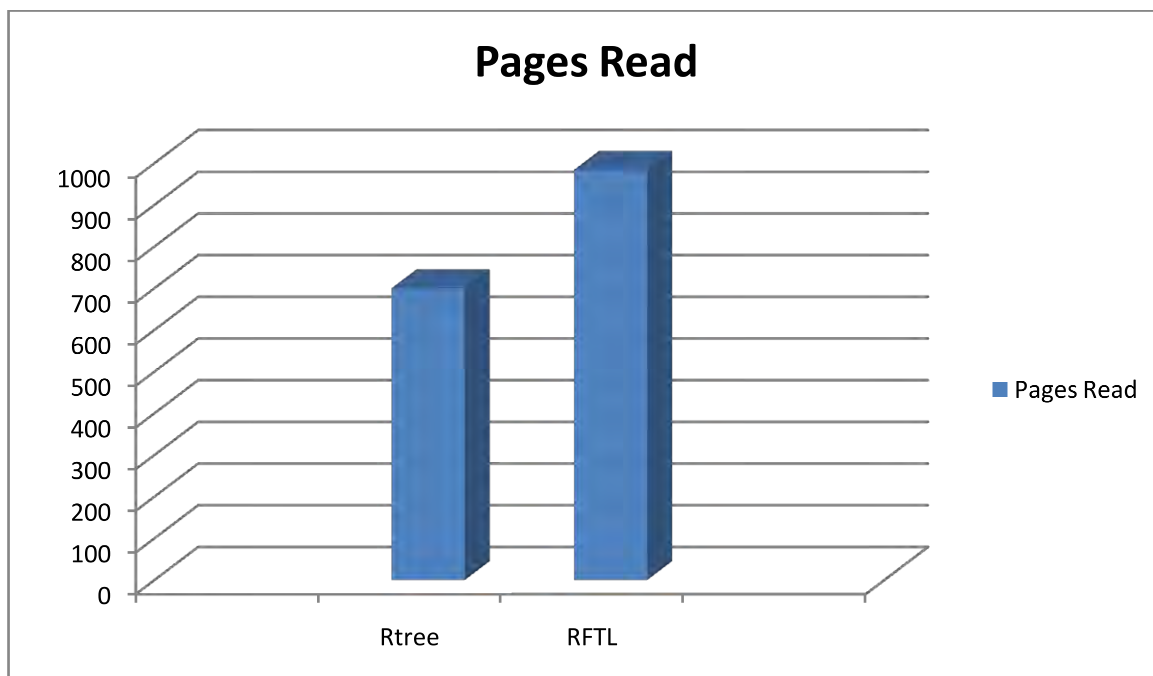
Εικόνα 6.4 Ο αριθμός σελίδων του δίσκου που εγράφησαν στο Πείραμα 2



Εικόνα 6.5 Ο αριθμός σελίδων του δίσκου που διαγράφηκαν στο Πείραμα 2

Πείραμα 3^{ο} :

Στο πείραμα αυτό απλά φορτώθηκε ολόκληρο το δέντρο που δημιουργήθηκε στο δεύτερο πείραμα.



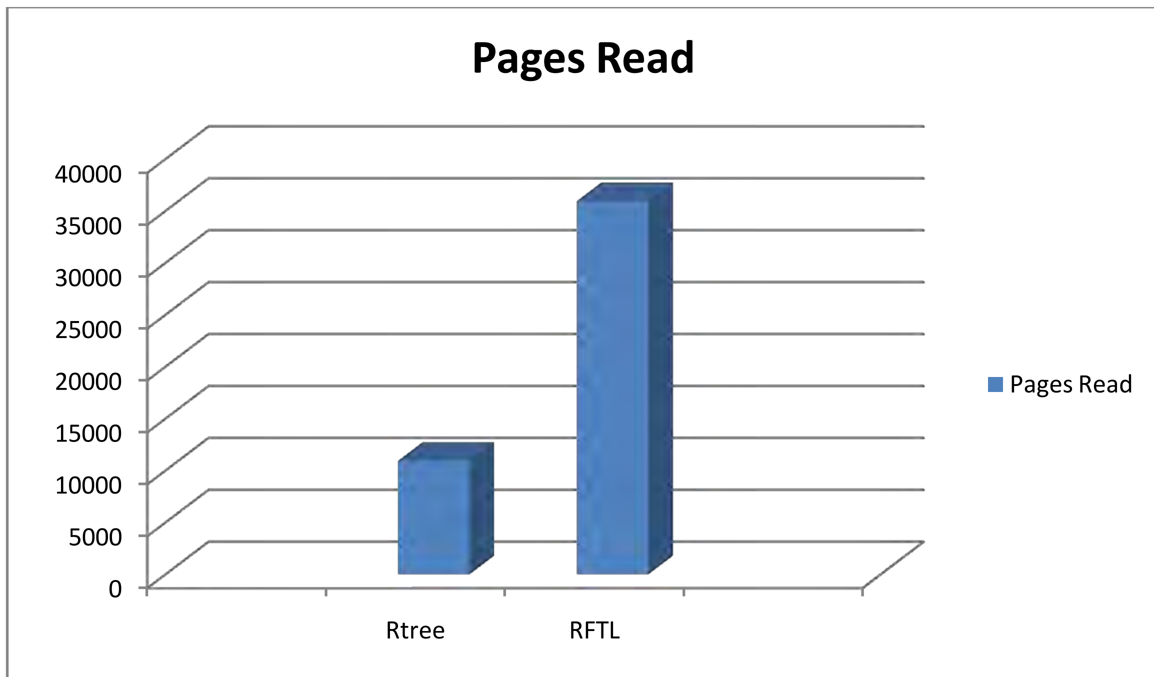
Εικόνα 6.6 Ο αριθμός σελίδων του δίσκου που διαβάστηκαν στο Πείραμα 3

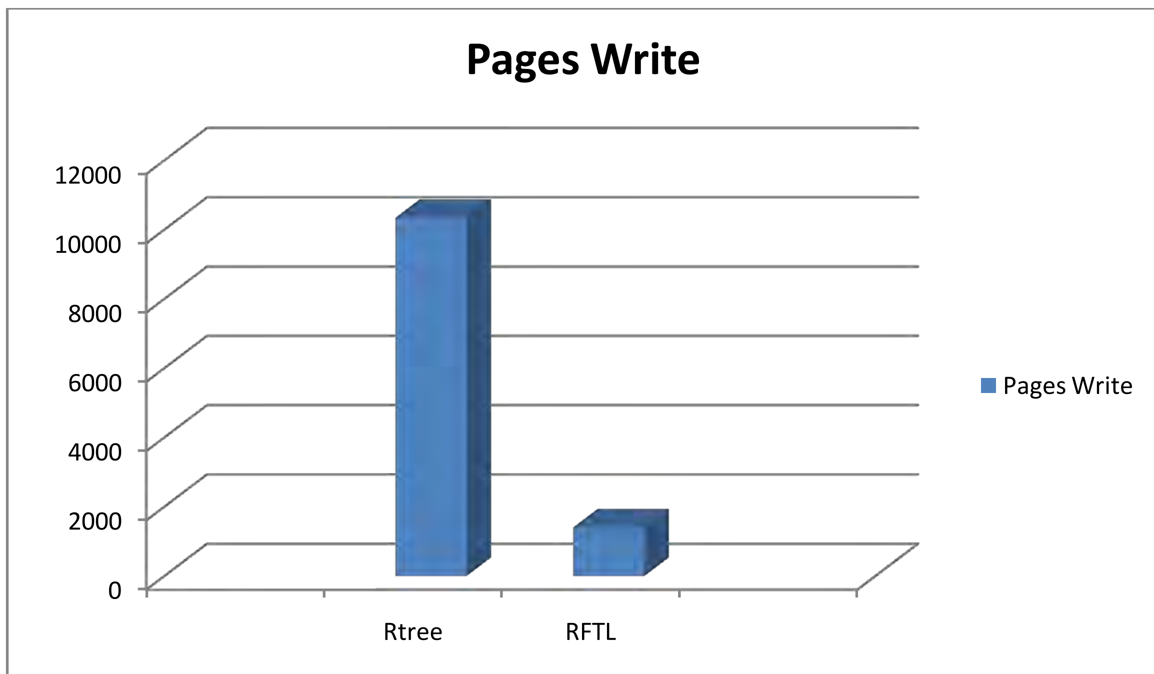
Από τα παραπάνω διαγράμματα φαίνεται πως ο αριθμός των εγγραφών στο RFTL είναι σημαντικά μικρότερος από τον αντίστοιχο του Rtree όπως ήταν επιθυμητό. Μάλιστα είναι 7 φορές μικρότερος ενώ στην υλοποίηση του κεφαλαίου 4 είναι 5 με 6 φορές. Επίσης ο αριθμός των διαγραφών είναι μηδενικός στο RFTL όπως αναμενόταν. Παρόλα αυτά όμως παρατηρείται μεγάλη αύξηση στον αριθμό των αναγνώσεων σε σχέση με το αρχικό Rtree. Αυτό οφείλεται κυρίως στο ότι στην υλοποίηση του RFTL δεν έχει ληφθεί περίπτωση για σύμπτυξη της λίστας στις εγγραφές του Node Translation Table όπως έχει γίνει στην προτεινόμενη υλοποίηση του κεφαλαίου 4. Επίσης σε σχέση με την υλοποίηση του κεφαλαίου 4 υπάρχουν αρκετές διαφορές στην διεξαγωγή των πειραμάτων που αφορούν διαφορετικό αρχικό R-δέντρο, διαφορετικό fan-out, διαφορετικό μέγεθος του ResBuffer και διαφορετική βάση εγγραφών που εισάγονται στο δέντρο.

Πιο αναλυτικά όσον αφορά τον αριθμό σελίδων που διαβάζονται από το δίσκο στην προτεινόμενη υλοποίηση του κεφαλαίου 4 αυτός μπορεί να είναι έως και 1.5 φορές μεγαλύτερος από ότι στο R-δέντρο. Στο RFTL ο αριθμός αυτός κυμαίνεται μεταξύ 3 και 3.5. Δηλαδή το RFTL πραγματοποιεί λίγο παραπάνω από 2 φορές περισσότερες αναγνώσεις από ότι η αντίστοιχη υλοποίηση του κεφαλαίου 4. Όπως έχει αναφερθεί και παραπάνω αυτή η διαφοροποίηση οφείλεται στην απουσία λειτουργίας σύμπτυξης στο RFTL. Η απουσία της λειτουργίας αυτής οδηγεί το RFTL στην ανάγνωση 4 με 5 σελίδων στο δίσκο κατά μέσο όρο για την κατασκευή ενός κόμβου ενώ αντίστοιχα η υλοποίηση στο κεφάλαιο 4 πραγματοποιεί 2 με 3 αναγνώσεις σελίδων. Δηλαδή κατά μέσο όρο κάτι λιγότερο από 2 φορές σε σύγκριση με το RFTL. Φαίνεται λοιπόν πως η διαφορά μεταξύ των δυο υλοποιήσεων έγκειται κυρίως στην απουσία ή μη της λειτουργίας σύμπτυξης. Για να τεκμηριωθεί το συμπέρασμα αυτό πραγματοποιήθηκε ένα ακόμα πείραμα(Πείραμα 4^{ο}) στο οποίο αυξήθηκε σημαντικά ο αριθμός των εγγραφών που εισάγονται στο δέντρο και παρατηρήθηκε πως τα ποσοστά και οι αναλογίες που προαναφέρθηκαν παρέμειναν σχεδόν σταθερά.

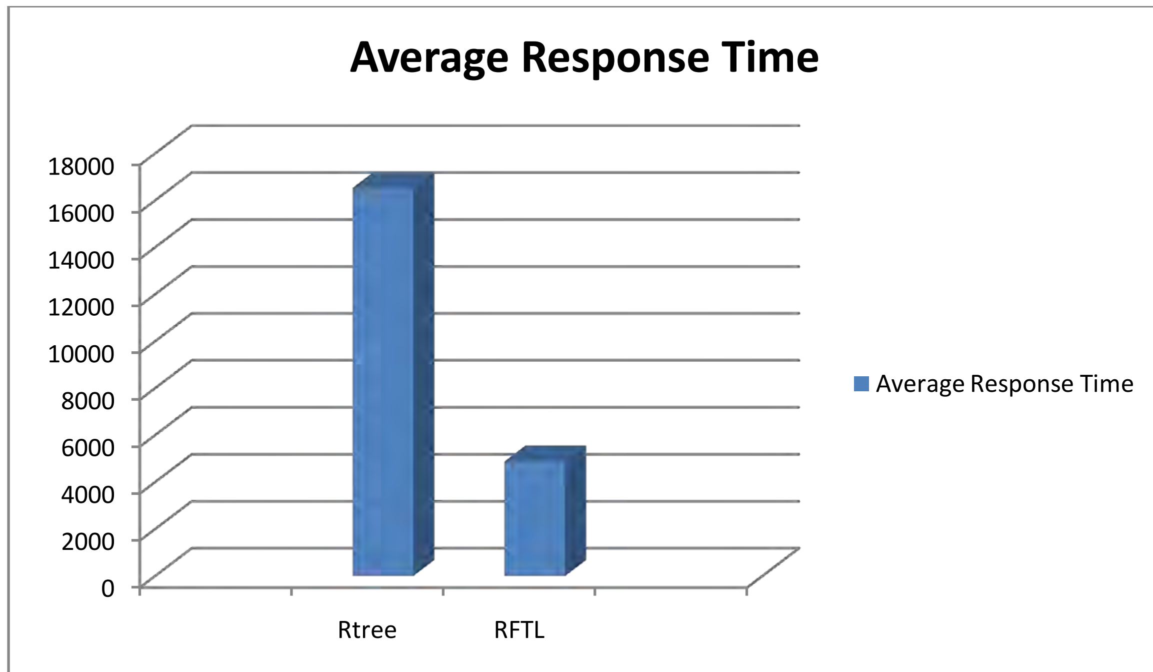
Πείραμα 4^{ο}

Στο πείραμα αυτό πραγματοποιείται εισαγωγή 7000 στοιχείων στο δέντρο. Τα γραφήματα που προκύπτουν είναι ανάλογα με αυτά του πειράματος 1 και καταδεικνύουν μια σταθερή απόδοση από το RFTL.


Pages Read
40000
35000
30000
25000
20000
15000
10000
5000
0
Rtree
RFTL
Pages Read


Pages Write
12000
10000
8000
6000
4000
2000
0
Rtree
RFTL
Pages Write

Τέλος στο πείραμα 4 μετρήθηκαν και οι χρόνοι των δυο υλοποιήσεων για την κατασκευή του δέντρου και ο χρόνος του RFTL ήταν σημαντικά μικρότερος από αυτόν του κανονικού R-δέντρου.



Παράρτημα

Βασικές Συναρτήσεις σε C++

Save:

```
//save an index unit structure to disk and make the insertion to NodeTransTable
  void  Save(std::list<IndexUnit> &mylist)
  {

       mfile.open("savefile",ios::out|ios::in|std::ios::binary);

       char buffer[PAGESIZE];
       int IUtotalsize = 6*sizeof(ELEMTYPE)+sizeof(char);
       memset(buffer,'0',PAGESIZE);
       int i = 0;
       int_2 table ;
       int flag = 0;
       int count =0;
       int list_count = 1;
       int *temp;


       for(std::list<IndexUnit>::iterator                        list_it=mylist.begin();
list_it!=mylist.end(); ++list_it)
       {

          temp = (int*)list_it->iu_id;

          memcpy(buffer+(i*IUtotalsize),&list_it->id,sizeof(int));
          memcpy(buffer+(i*IUtotalsize)+sizeof(ELEMTYPE),&list_it-
>minimal_bounding_box.m_min[0],sizeof(ELEMTYPE));
          memcpy(buffer+(i*IUtotalsize)+2*sizeof(ELEMTYPE),&list_it-
>minimal_bounding_box.m_min[1],sizeof(ELEMTYPE));
          memcpy(buffer+(i*IUtotalsize)+3*sizeof(ELEMTYPE),&list_it-
>minimal_bounding_box.m_max[0],sizeof(ELEMTYPE));
          memcpy(buffer+(i*IUtotalsize)+4*sizeof(ELEMTYPE),&list_it-
>minimal_bounding_box.m_max[1],sizeof(ELEMTYPE));
          memcpy(buffer+(i*IUtotalsize)+5*sizeof(ELEMTYPE),&list_it-
>op_flag,sizeof(char));

          list_count++;


memcpy(buffer+(i*IUtotalsize)+5*sizeof(ELEMTYPE)+sizeof(char),&temp,sizeof(int));

          table.i_0 = page_id;
          table.i_1 = i;
          NodeTransTable.insert(std::pair<int,int_2>((const int) temp,table));

              ++i;
        }

        update_flag = 0;
        int seek = page_id*PAGESIZE;

        ++page_id;
```

```
    mfile.seekp(seek,std::ios_base::beg);
    mfile.write (buffer, PAGESIZE);

      if (mfile.fail()){

            mfile.clear();
            mfile.write (buffer, PAGESIZE);
      }

    mfile.flush();

    if(mfile)
    {
        mfile.close();
    }

  }
```

Insert:

```
// Inserts a new data rectangle into the index structure.
// Recursively descends tree, propagates splits back up.
// Returns 0 if node was not split.  Old node updated.
// If node was split, returns 1 and sets the pointer pointed to by
// new_node to point to the new node.  Old node updated to become one of two.
// The level argument specifies the number of steps up from the leaf
// level to insert; e.g. a data rectangle goes in at level = 0.
// If is an internal insertion then make the changes in NodeTransTable and IndexUnit
list
RTREE_TEMPLATE
bool  RTREE_QUAL::InsertRectRec(Rect*  a_rect,  const  DATATYPE&  a_id,  Node*  a_node,
Node** a_newNode, int a_level, int iucount, int interval)
{
  ASSERT(a_rect && a_node && a_newNode);
  ASSERT(a_level >= 0 && a_level <= a_node->m_level);

  int index = 0;
  Branch branch;
  Node* otherNode;

  // Still above level for insertion, go down tree recursively
  if(a_node->m_level > a_level)
  {
    index = PickBranch(a_rect, a_node);

    if  (!InsertRectRec(a_rect,  a_id,  a_node->m_branch[index].m_child,  &otherNode,
a_level, iucount, interval))
    {
      // Child was not split
      a_node->m_branch[index].m_rect       =       CombineRect(a_rect,       &(a_node-
>m_branch[index].m_rect));

      return false;
    }

    else // Child was split
    {
```

```
      a_node->m_branch[index].m_rect = NodeCover(a_node->m_branch[index].m_child);
      branch.m_child = otherNode;
      branch.m_rect = NodeCover(otherNode);

         bool res = AddBranch(&branch, a_node, a_newNode);

      return res;

    }
  }
  else if(a_node->m_level == a_level) // Have reached level for insertion. Add rect, 
split if necessary
  {
    branch.m_rect = *a_rect;
    branch.m_child = (Node*) a_id;

       if(a_node->unique_id < 0)
         {
              a_node->unique_id = ++UniqueId;
         }

       if(interval==0)
       {
              branch.iu_count = ++IUcount;
              nodebuffer[1] = a_node;
       }

       else if(interval > 0)
       {

              branch.iu_count = iucount;

              char buffer[PAGESIZE];
              mfile.open("savefile", ios::in|ios:: out|std::ios::binary);
              int IUtotalsize = 5*sizeof(ELEMTYPE)+sizeof(int)+sizeof(char);
              int flag = 0;
              memset(buffer,'0',PAGESIZE);
              int_2 table ;
              pair<multimap  <int,int_2>::iterator,  multimap  <int,int_2>::iterator  > 
map_it ;
              map_it = NodeTransTable.equal_range(interval);

              for  (std::multimap  <int,  int_2>::iterator  it2  =  map_it.first;  it2  != 
map_it.second;)
              {
                     int seek = (*it2).second.i_0*PAGESIZE;
                     mfile.seekg(seek,std::ios_base::beg);
                     mfile.read ((char*)buffer, sizeof(buffer));

                     ELEMTYPE load_temp[4];
                     int    load_id;
                     int newid=-1;

                     memcpy(&load_id,buffer+((*it2).second.i_1*
IUtotalsize),sizeof(int));
                     memcpy(&load_temp[0],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(int),sizeof(ELEMTYPE));
                     memcpy(&load_temp[1],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                     memcpy(&load_temp[2],buffer+((*it2).second.i_1*
IUtotalsize)+2*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
```

```
				memcpy(&load_temp[3],buffer+((*it2).second.i_1*
IUtotalsize)+3*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));

				if((load_temp[0]   ==   branch.m_rect.m_min[0])&&(load_temp[1]   ==
branch.m_rect.m_min[1])
					&&(load_temp[2]  ==   branch.m_rect.m_max[0])&&(load_temp[3]
==  branch.m_rect.m_max[1])&&(load_id == branch.m_data))
				{
					table.i_0 = (*it2).second.i_0;
					table.i_1 = (*it2).second.i_1;
					std::multimap <int, int_2>::iterator itcopy = it2;
					itcopy++;

					NodeTransTable.erase(it2);
					it2=itcopy;
					NodeTransTable.insert(std::pair<int,int_2>((a_node)-
>unique_id,table));
					flag=1;

					break;
				}

				else ++it2;
			}


			if(flag==0)
			{

					int list_count = 1;
					std::list<int_2>::iterator s_it;

					for(std::list<IndexUnit>::iterator  list_it=mylist.begin();
list_it!=mylist.end(); ++list_it)
					{
						if(list_count == branch.iu_count)
						{
							list_it->iu_id      =      (int)(*a_newNode)-
>unique_id;
							flag=2;
							break;

						}
						list_count++;
					}

			}

			if((flag == 0)&&(interval==0))
			{
				nodebuffer[1] = (*a_newNode);
			}


			if(mfile)
			{
			mfile.close();
			}

		}

		bool res_else = AddBranch(&branch, a_node, a_newNode);
```

```
    return res_else;

 }

  else
  {
    // Should never occur
    ASSERT(0);
    return false;
  }
}
```

SplitNode:

```
// Split a node.
// Divides the nodes branches and the extra one between two nodes.
// Old node is one of the new ones, and one really new one is created.
// Tries more than one method for choosing a partition, uses best result.
// Make changes to NodeTransTable and Index units list
RTREE_TEMPLATE
void RTREE_QUAL::SplitNode(Node* a_node, Branch* a_branch, Node** a_newNode)
{
  ASSERT(a_node);
  ASSERT(a_branch);

  // Could just use local here, but member or external is faster since it is reused
  PartitionVars localVars;
  PartitionVars* parVars = &localVars;
  int level;
  // Load all the branches into a buffer, initialize old node
  level = a_node->m_level;
  GetBranches(a_node, a_branch, parVars);

  // Find partition
  ChoosePartition(parVars, MINNODES);

  // Put branches from buffer into 2 nodes according to chosen partition
  *a_newNode = AllocNode();
  (*a_newNode)->m_level = a_node->m_level = level;
  (*a_newNode)->unique_id = ++UniqueId;

  LoadNodes(a_node, *a_newNode, parVars);

  if(a_node->IsLeaf())
  {
        mfile.open("savefile", ios::in|ios:: out|std::ios::binary);
        int IUtotalsize = 5*sizeof(ELEMTYPE)+sizeof(int)+sizeof(char);
        char buffer[PAGESIZE];
        int flag = 0;
        char op_flag;
        memset(buffer,'0',PAGESIZE);
        int_2 table ;
        std::list<int> splitNodes;
        splitNodes.push_back(a_node->unique_id);
        splitNodes.push_back((*a_newNode)->unique_id);
     std::list<int> split_it;
        update_flag = 1;
```

```
         int j_count=0;
         ELEMTYPE load_temp[4];
         int load_id;

        pair<multimap <int,int_2>::iterator, multimap <int,int_2>::iterator > map_it
;//= map_it.second;

        for(int index=0; index < parVars->m_total; ++index)
        {

        map_it = NodeTransTable.equal_range(a_node->unique_id);

        if(parVars->m_partition[index] == 1)
        {

                flag=0;
                for (std::multimap <int, int_2>::iterator it2 = map_it.first; it2 !=
map_it.second;)
                {

                        int seek = (*it2).second.i_0*PAGESIZE;
                        mfile.seekg(seek,std::ios_base::beg);
                        mfile.read ((char*)buffer, sizeof(buffer));

                        memcpy(&load_id,buffer+((*it2).second.i_1*
IUtotalsize),sizeof(int));
                        memcpy(&load_temp[0],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(int),sizeof(ELEMTYPE));
                        memcpy(&load_temp[1],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                        memcpy(&load_temp[2],buffer+((*it2).second.i_1*
IUtotalsize)+2*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                        memcpy(&load_temp[3],buffer+((*it2).second.i_1*
IUtotalsize)+3*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                        memcpy(&op_flag,buffer+((*it2).second.i_1*
IUtotalsize)+4*sizeof(ELEMTYPE)+sizeof(int),sizeof(char));

                        if((load_temp[0]                    ==                  parVars-
>m_branchBuf[index].m_rect.m_min[0])&&(load_temp[1]          ==             parVars-
>m_branchBuf[index].m_rect.m_min[1])
                                &&(load_temp[2]              ==                 parVars-
>m_branchBuf[index].m_rect.m_max[0])&&(load_temp[3]          ==             parVars-
>m_branchBuf[index].m_rect.m_max[1])&&(load_id==parVars->m_branchBuf[index].m_data))
                        {

                                table.i_0 = (*it2).second.i_0;
                                table.i_1 = (*it2).second.i_1;

                                split_it.remove(index);

                                std::multimap <int, int_2>::iterator itcopy = it2;
                                itcopy++;

                                NodeTransTable.erase(it2);
                                it2=itcopy;

                                NodeTransTable.insert(std::pair<int,int_2>((*a_newNode)-
>unique_id,table));

                                flag=1;
                                break;
```

```
                        }

                        else ++it2;
                }


                if(flag==0)
                {

                                int list_count = 1;
                                std::list<int_2>::iterator s_it;

                                for(std::list<IndexUnit>::iterator  list_it=mylist.begin();
list_it!=mylist.end(); ++list_it)
                                {

                                        if(list_count              ==              parVars-
>m_branchBuf[index].iu_count)
                                        {

                                                list_it->iu_id      =      (int)(*a_newNode)-
>unique_id;

                                                flag=2;
                                                break;

                                        }
                                        list_count++;

                                }



                }
                if(flag == 0)
                {

                        nodebuffer[1] = (*a_newNode);
                }
        }


                j_count = 0;

  }



   if(mfile)
   {
                mfile.close();
    }

  }

  ASSERT((a_node->m_count + (*a_newNode)->m_count) == parVars->m_total);
}
```

ConstractNode:

```
RTREE_TEMPLATE
void RTREE_QUAL::ConstractNode(Node **a_node, int node_id)
{
        mfile.open("savefile",ios::out|ios::in|std::ios::binary);

        if(!mfile)
          {
          mfile.clear();
          }

          *a_node = AllocNode();
          (*a_node)->m_level = 0;
          (*a_node)->m_count = 0;
          (*a_node)->unique_id = node_id;
          char buffer[PAGESIZE];

          struct tempbranch
          {
                 int t_id;
                 ELEMTYPE t_min[2];
                 ELEMTYPE t_max[2];
                 char t_flag;
                 int t_iucount;
          };

          std::list< tempbranch> mylist_i;   //list for inserts
          std::list< tempbranch> mylist_r;   //list for removes

          tempbranch ins;
          tempbranch rem;
          int i=1;
          int IUtotalsize = 5*sizeof(ELEMTYPE)+sizeof(int)+sizeof(char);
          int flag = 0;
          memset(buffer,'0',PAGESIZE);

          pair<multimap <int,int_2>::iterator, multimap <int,int_2>::iterator > map_it
;
          map_it = NodeTransTable.equal_range(node_id);

      int index = 0;
      char op;

      for (multimap <int, int_2>::iterator it2 = map_it.first; it2 != map_it.second;
++it2)
         {
             int seek = (*it2).second.i_0*PAGESIZE;

             mfile.seekg(seek,std::ios_base::beg);
             mfile.read ((char*)buffer, sizeof(buffer));

             if (mfile.fail())
             {
           mfile.clear();
                    mfile.seekg(seek,std::ios_base::beg);
           mfile.read ((char*)buffer, sizeof(buffer));
         }

             flag=1;
```

```
            memcpy(&op,buffer+((*it2).second.i_1*
IUtotalsize)+5*sizeof(int),sizeof(char));

            if(op=='i')
            {
                    memcpy(&ins.t_id,buffer+((*it2).second.i_1*
IUtotalsize),sizeof(int));
                    memcpy(&ins.t_min[0],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&ins.t_min[1],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&ins.t_max[0],buffer+((*it2).second.i_1*
IUtotalsize)+2*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&ins.t_max[1],buffer+((*it2).second.i_1*
IUtotalsize)+3*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    ins.t_flag = op;
                    mylist_i.push_back(ins);
            }

            if(op=='r')
            {
                    memcpy(&rem.t_id,buffer+((*it2).second.i_1*
IUtotalsize),sizeof(int));
                    memcpy(&rem.t_min[0],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&rem.t_min[1],buffer+((*it2).second.i_1*
IUtotalsize)+sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&rem.t_max[0],buffer+((*it2).second.i_1*
IUtotalsize)+2*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    memcpy(&rem.t_max[1],buffer+((*it2).second.i_1*
IUtotalsize)+3*sizeof(ELEMTYPE)+sizeof(int),sizeof(ELEMTYPE));
                    rem.t_flag = op;
                    mylist_r.push_back(rem);

            }

            index++;

        }

        for(std::list<IndexUnit>::iterator                    list_it=mylist.begin();
list_it!=mylist.end(); ++list_it)
        {

              if(list_it->IUNode->unique_id ==  (*a_node)->unique_id)
              {
                     if((char)list_it->op_flag == 'i')
                     {
                            ins.t_id = (int)list_it->id;
                            ins.t_min[0] = list_it->minimal_bounding_box.m_min[0];
                            ins.t_min[1] = list_it->minimal_bounding_box.m_min[1];
                            ins.t_max[0] = list_it->minimal_bounding_box.m_max[0];
                            ins.t_max[1] = list_it->minimal_bounding_box.m_max[1];
                            ins.t_flag = 'i';
                            ins.t_iucount = i;
                            mylist_i.push_back(ins);

                     }

                     else if((char)list_it->op_flag == 'r')
                     {
```

```
                                        rem.t_id = (int)list_it->id;
                                        rem.t_min[0] = list_it->minimal_bounding_box.m_min[0];
                                        rem.t_min[1] = list_it->minimal_bounding_box.m_min[1];
                                        rem.t_max[0] = list_it->minimal_bounding_box.m_max[0];
                                        rem.t_max[1] = list_it->minimal_bounding_box.m_max[1];
                                        rem.t_flag = 'r';
                                        rem.t_iucount = i;
                                        mylist_r.push_back(rem);

                                }
                        }

                ++i;
                }

               std::list<tempbranch>::iterator i_it;
               std::list<tempbranch>::iterator r_it;

              for(std::list<tempbranch>::iterator                      r_it=mylist_r.begin();
r_it!=mylist_r.end(); r_it++){


                     for(std::list<tempbranch>::iterator               i_it=mylist_i.begin();
i_it!=mylist_i.end(); ++i_it)
                     {
                            if((r_it->t_id        ==        i_it->t_id)&&(r_it->t_min[0]==i_it-
>t_min[0])&&(r_it->t_min[1]==i_it->t_min[1])
                                   &&(r_it->t_max[0]==i_it->t_max[0])&&(r_it->t_max[1]==i_it-
>t_max[1]))
                            {
                                   std::list<tempbranch>::iterator itcopy = i_it;

                                   itcopy++;
                                   mylist_i.erase(i_it);
                                   i_it=itcopy;
                                   break;
                            }
                     }
               }

              index = 0;

              for(std::list<tempbranch>::iterator                      i_it=mylist_i.begin();
i_it!=mylist_i.end(); ++i_it)
              {

                     (*a_node)->m_branch[index].iu_count = i_it->t_iucount;
                     (*a_node)->m_branch[index].m_data = i_it->t_id;
                     (*a_node)->m_branch[index].m_rect.m_min[0] = i_it->t_min[0];
                     (*a_node)->m_branch[index].m_rect.m_min[1] = i_it->t_min[1];
                     (*a_node)->m_branch[index].m_rect.m_max[0] = i_it->t_max[0];
                     (*a_node)->m_branch[index].m_rect.m_max[1] = i_it->t_max[1];
                     (*a_node)->m_count++;
                     index++;
              }
              if(mfile)
           {
                     mfile.close();
           }

}
```

Παραπομπές

[1] A. Guttman R-Tree: A dynamic index structure for spatial searching.

[2] http://en.wikipedia.org/wiki/Computer_data_storage

[3] http://en.wikipedia.org/wiki/Transistor

[4] http://en.wikipedia.org/wiki/File_system

[5] http://en.wikipedia.org/wiki/Garbage_collection_(SSD)#Garbage_collection

[6] Wu CH, Chang LP, Kuo TW, An Efficient R-Tree Implementation over Flash-Memory Storage Systems

[7] http://mathworld.wolfram.com/Bin-PackingProblem.html

[8] http://en.wikipedia.org/wiki/NP-hard

[9] Yanfei Lv, Bin Cui, Xuexuan Chen Log-Compact R-Tree: An Efficient Spatial Index for SSD

[10] http://www.cplusplus.com/doc/tutorial/templates/

[11] http://www.cplusplus.com/reference/map/multimap/

[12] http://www.chorochronos.org/