Πανεπιστήμιο Θεσσαλίας

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

Παναγιώτης Φωτεινός

WEIGHTED-R-TREES

Επιτροπή: Παναγιώτης Μποζάνης, Επιβλέπων

Δημήτριος Κατσαρός

Περιεχόμενα

Κεφάλαιο 1

ΕΙΣΑΓΩΓΗ

[1] Τα τελευταία χρόνια, είναι εμφανής η παρουσία εφαρμογών που χειρίζονται γεωμετρικά δεδομένα (geo-data objects). Τα συστήματα γεωγραφικών πληροφοριών (GIS), τα αυτόματα εργαλεία σχεδίασης (CAD design) και τα συστήματα πρόβλεψης και ανίχνευσης κίνησης κινούμενων αντικειμένων είναι μερικές αντιπροσωπευτικές εφαρμογές. Αντίστοιχα, τα γεω-δεδομένα μπορεί να είναι σημεία, γραμμές, καλά ορισμένες περιοχές και όγκοι δύο, τριών, τεσσάρων ή και περισσοτέρων διαστάσεων. Εξαιτίας του μεγάλου μεγέθους των δεδομένων, τα συστήματα βάσεων που υποστηρίζουν τέτοιες εφαρμογές, πρέπει να είναι εφοδιασμένες με κατάλληλο σύστημα δεικτοδότησης (indexing) τέτοιο ώστε η εύρεση των δεδομένων να γίνεται με αποτελεσματικότητα και ταχύτητα.

Δύο θεμελιώδεις ερωτήσεις που πρέπει να απαντούν οι ειδικές αυτές δομές δεικτοδότησης (index structures) είναι η ερώτηση τομής (intersection query)

[1] Στο παρόν βιβλίο χρησιμοποιήθηκε η γραμματοσειρά Kerkis, τα δικαιώματα της οποίας ανήκουν στο Τμήμα Μαθηματικών του Πανεπιστημίου Αιγαίου.

και η ερώτηση των k κοντινότερων γειτόνων (k nearest neighbors). Με είσοδο ένα γεωμετρικό σχήμα, η πρώτη ερώτηση επιστρέφει όλα τα δεδομένα της βάσης που τέμνονται με το σχήμα αυτό και η δεύτερη επιστρέφει τα k πιο κοντινά δεδομένα στο σχήμα.

Για να γίνει κατανοητή η ανάγκη χρήσης των ειδικών δομών δεικτοδότησης, ας υποθέσουμε ότι γεωμετρικά δεδομένα είναι αποθηκευμένα σε ένα σύστημα βάσης χωρίς δεικτοδότηση. Τότε, εύκολα κανείς συμπεραίνει, ότι η κάθε μια από τις δύο ερωτήσεις κοστίζει σε προσβάσεις στον δίσκο (I/O), $\Theta(\frac{n}{pagecapacity})$ στην καλύτερη, την μέση και την χειρότερη περίπτωση. Αν λάβουμε υπόψιν μας ότι το κάθε δεδομένο καταλαμβάνει, ήδη, μεγάλο χώρο και ότι οι αντίστοιχες εφαρμογές χειρίζονται μεγάλο αριθμό δεδομένων, ένα τέτοιο κόστος είναι ανεπίτρεπτο.

Κεφάλαιο 2

ΑΝΑΣΚΟΠΗΣΗ ΔΥΝΑΜΙΚΩΝ ΔΟΜΩΝ ΔΕΙΚΤΟΔΟΤΗΣΗΣ

2.1 R-tree

Η πρώτη δομή δεικτοδότησης που μπορούσε να χειριστεί μεγάλα σύνολα από πολυδιάστατα γεωμετρικά δεδομένα, επινοήθηκε από τον Antonin Guttman το 1984. Πρόκειται για το διάσημο, πλέον, R-tree.

Τα φύλα του R-tree περιέχουν εγγραφές (index records) της μορφής

(I,objID)

όπου το $objID$ είναι η πραγματική διεύθυνση στην οποία κατοικεί το δεδομένο στην βάση και το I είναι το μικρότερο σε όγκο παραλληλόγραμμο που καλύπτει το δεδομένο αυτό (Minimum Bounding Rectangle). Οι διαστάσεις του MBR είναι, φυσικά, όσες είναι οι διαστάσεις των γεωμετρικων δεδομένων.

Οι εσωτερικοί κόμβοι περιέχουν εγγραφές (entries) της μορφής

(I,child-pointer)

όπου το child-pointer είναι η διεύθυνση ενός χαμηλότερου κόμβου του δένδρου και το I είναι το MBR που καλύπτει όλα τα MBRs του χαμηλότερου αυτού κόμβου.

Έστω M, ο μέγιστος αριθμός εγγραφών που μπορούν να αποθηκευτούν σε έναν κόμβο και

$$m \leqslant \frac{M}{2}$$

η παράμετρος που καθορίζει τον ελάχιστο αριθμό εγγραφών. Τότε το R-tree ορίζεται, αυστηρά, ως εξής:

1. Κάθε φύλο περιλαμβάνει από m μέχρι M index records εκτός αν είναι η ρίζα.

2. Κάθε εσωτερικός κόμβος περιλαμβάνει από m μέχρι M entries εκτός αν είναι η ρίζα.

3. Η ρίζα περιλαμβάνει τουλάχιστον δύο και το πολύ M εγγραφές εκτός αν είναι φύλο.

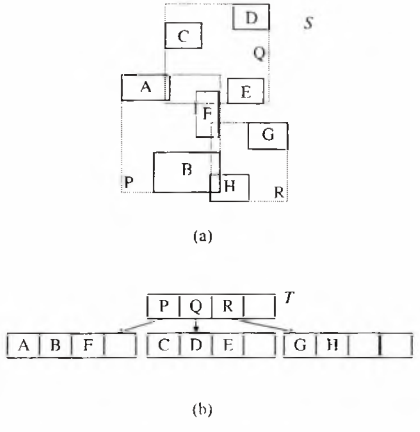
4. Όλα τα φύλα βρίσκονται στο ίδιο βάθος.

Η εικόνα A.1 δείχνει ένα στιγμιότυπο ενός R-tree με τις επικαλύψεις των MBRs.

Το R-tree είναι δυναμική δομή, υποστηρίζοντας την προσθήκη και αφαίρεση γεωμετρικών δεδομένων με αποτέλεσμα την αλλαγή του συνόλου S των δεδομένων που δεικτοδοτεί. Φυσικά, μια στατική δομή θα τακτοποιούσε τους

δείκτες στα δεδομένα καλύτερα, με αποτέλεσμα την ελαχιστοποίηση του πλήθους των προσβάσεων στον δίσκο (I/O) για την απάντηση των ερωτήσεων. Μια στατική δομή όμως θα έπρεπε κάθε φορά που θα παρουσιαζόταν παραβίαση των συνθηκών (κατά την προσθήκη ή αφαίρεση δεδομένων) να φτιαχνόταν από την αρχή για όλα τα δεδομένα που δεικτοδοτούσε μέχρι εκείνη την στιγμή (global bulk loading-rebuilding technique). Κάτι τέτοιο, όμως, αυξάνει το κόστος των ενθέσεων και των διαγραφών και σε χρόνο CPU αλλά και σε I/O. Για αυτόν τον λόγο, όταν διαπιστωθεί παραβίαση, το R-tree συμμορφώνεται τοπικά (local rebalancing). Στο local rebalancing, αντίθετα με το global rebuilding, διορθώνονται μερικά μονοπάτια από κάποια φύλα μέχρι την ρίζα.

Έστω λοιπόν, πως στην δομή προστίθεται ένα νέο δεδομένο. Η δομή θα πρέπει να αποφασίσει που θα αποθηκευτεί η εγγραφή index record που θα δεικτοδοτεί το νέο αυτό δεδομένο. Το κριτήριο που χρησιμοποιεί ο Guttman



Σχήμα 2.1: An R-tree instance.

είναι η ελαχιστοποίηση της αύξησης του όγκου των MBRs [1]. Με αυτόν τον τρόπο, ο αναγνώστης μπορεί να διαισθανθεί ότι, προσπαθεί η δομή να αποφύγει, όσο είναι δυνατόν, τις επικαλύψεις μεταξύ των MBRs του ίδιου κόμβου. Ιδανικά, δε, θα θέλαμε, η εύρεση κάθα δεδομένου να ενεργοποιεί μόνο ένα μονοπάτι, κάτι που μπορεί να συμβεί αν δεν υπάρχουν καθόλου επικαλύψεις. Αρχίζοντας, επομένως, από την ρίζα, διαλέγουμε να κατεβούμε σε εκείνο το παιδί, του οποίου το MBR χρειάζεται την μικρότερη αύξηση όγκου για να καλύψει το νέο δεδομένο. Θυμίζουμε ότι το MBR ενός κόμβου είναι αποθηκευμένο σε κάποια εγγραφή του πατέρα του. Η διαδικασία συνεχίζεται μέχρι να φτάσουμε σε κάποιο φύλο. Εκεί, υπολογίζουμε το MBR του νέου δεδομένου και έναν δείκτη που δείχνει την ακριβή θέση του νέου δεδομένου στον δίσκο, βρίσκοντας με αυτόν τον τρόπο το index record του νέου δεδομένου που αποθηκεύεται, όπως είπαμε, στο φύλο αυτό. Για την διαγραφή ενός στοιχείου, πρέπει να βρούμε σε ποιο φύλο βρίσκεται το αντίστοιχο index record. Επομένως, με depth first σειρά, κατεβαίνουμε σε εκείνους τους κόμβους των οποίων το MBR τέμνεται με το δεδομένο προς εύρεση. Όταν φτάνουμε στα φύλα, ελέγχουμε όλα τα δεδομένα που δεικτοδοτούνται απο τα index records τους για να βρουμε πιο από όλα τα index records είναι εκείνο που δεικτοδοτεί το δεδομένο. Μόλις το βρούμε, αφαιρούμε το εν λόγω index record. Μετά και από τις δύο πράξεις (αφού βρούμε το φύλο) ανεβαίνουμε το μονοπάτι από το φύλο αυτό μέχρι την ρίζα, προσαρμόζοντας τα αντίστοιχα MBRs, γιατί πλέον δεν είναι έγκυρα.

Κατά την διάρκεια των ενθέσεων και των διαγραφών, όπως ήδη έχουμε αναφέρει, ίσως παραβιαστούν οι συνθήκες. Αυτό συμβαίνει γιατί κάποιο φύ-

[1] Στις δυο διαστάσεις ο όγκος αναφέρεται στο εμβαδόν των MBRs

λο, ίσως έχει πάνω από M εγγραφές ή λιγότερες από m ή ακόμα η ρίζα να έχει μόνο μία εγγραφή και ύψος μη μηδενικό. Στην πρώτη περίπτωση κάνουμε split. Δηλαδή , διασπούμε το φύλο σε δύο καινούρια με αποτέλεσμα ο πατέρας του να αποκτά μια παραπάνω εγγραφή. Στην δεύτερη περίπτωση, αφαιρούμε τελείως το φύλο (οπότε ο πατέρας έχει μια εγγραφή λιγότερη) και ξαναβάζουμε στην δομή τις εγγραφές που κατοικούσαν στο φύλο αυτό σαν να έμπαιναν για πρώτη φορά (reinsertion). Και στις δυο περιπτώσεις δεν παραλείπουμε να ανεβούμε τα μονοπάτια τα οποία κατεβήκαμε, όχι μόνο για να προσαρμόσουμε τα MBRs αλλά και για να ξαναεκτελέσουμε, πιθανότατα, το split ή το reinsertion, καθώς μεταβάλεται το πλήθος των εγγραφών και στους προγόνους. Στην τρίτη περίπτωση, διαγράφεται τελείως η ρίζα και τον ρόλο της νέας ρίζας τον παίρνει το μοναδικό παιδί της παλιάς.

Πολλά μπορεί να ειπωθούν για το τρόπο με τον οποίο γίνεται το split. Γενικά, ο Guttman προτείνει τρεις τρόπους, ο καθένας από τους οποίους παρουσιάζει διαφορετική ταχύτητα περάτωσης και αποτελεσματικότητα. Το κοινό τους στοιχείο, πάντως, είναι ότι προσπαθούν να ελαχιστοποιήσουν τον όγκο που καταλαμβάνουν τα MBRs των δύο νέων κόμβων.

2.2 $R^* - tree$

Το 1990, τέσσερις Γερμανοί επινόησαν μια παραλλαγή του R-tree, την οποία και ονόμασαν $R^* - tree$. Το διαφορετικό ήταν ότι έπαιρναν μέρος πιο πολλά κριτήρια, κριτήρια που δεν ελαχιστοποιούσαν μόνο τον όγκο των MBRs αλλά και την επικάλυψή τους καθώς και το storage utilization. Ως storage utilization ορίζεται: $\frac{average-number-of-entries-per-node}{M}$. Ακόμα, για τον

περιορισμό του πλήθους των splits, επινόησαν την τεχνική "forced reinsertion". Κατά την τεχνική αυτή, αντί να γίνει split του κόμβου, αφαιρούσαν το 30% των εγγραφών του, τις οποίες ξαναπρόσθεταν στην δομή. Με αυτόν τον τρόπο μείωναν το πλήθος των splits και ταυτόχρονα βελτίωναν την συμπεριφορά της δομής καθώς τέτοιες δυναμικές δομές, πάσχουν από δεδομένα που έχουν μπει παλιά.

Τελικά αυτό που κατάφεραν οι Γερμανοί με το $R^* - tree$, ήταν μια βελτίωση στο μέσο αριθμό προσβάσεων στον δίσκο ανά ερώτηση, μέχρι και 138%. Πραγματικά αλλά και συνθετικά (ομοιόμορφης και γκαουσιανής κατανομής) σύνολα γεωγραφικών δεδομένων αποδεικνύουν την υπεροχή της νέας δομής.

2.3 Λοιπές δυναμικές/ημιδυναμικές δομές δεικτοδότησης

Μετά το 1983 και το 1990, εμφανίστηκαν πλήθος δομών που σαν σκοπό είχαν την βελτίωση του κλασικού R-tree και του $R^* - tree$. Ενδεικτικά αναφέρουμε την Small-Tree-Large-Tree τεχνική των Li Chen, Rupesh Choubey και Elke A. Rundensteiner ([26]), το Merging R-tree των Βασίλη Βασαΐτη, Αλέξανδρου Νανόπουλου και Παναγιώτη Μποζάνη ([25]), το Hilbert R-tree των Ibrahim Kamel και Χρήστου Φαλούτσου ([16]), το LR-tree των Παναγιώτη Μποζάνη, Αλέξανδρου Νανόπουλου και Γιάννη Μανωλόπουλου ([12]) και την bulk loading τεχνική των Stefan Berchtold, Christian Böhm και Hans-Peter Kriegel ([7, 8]).

Κεφάλαιο 3

WeR-Tree

3.1 Θεωρητική τεκμηρίωση

Τελικός σκοπός της διπλωματικής μου εργασίας ήταν η θεωρητική κατανόηση δομών δεικτοδότησης πολυδιάστατων δεδομένων, η υλοποίηση των WeR-trees και η σύγκρισή τους με το $R^* - tree$.

Ορισμός 1. *Ένα WeR-tree είναι ένα R-tree του οποίου όμως οι κόμβοι είναι σε ισορροπία. Από την άλλη, ένας κόμβος v είναι σε ισορροπία, όταν το αντίστοιχο υπόδενδρο T_v είναι ισοδύναμο με ένα bulk-loaded R-tree, στο ίδιο σύνολο δεδομένων S_v.*

Αναφέραμε στο 2.1, νωρίτερα, ότι τα global rebuildings είναι πολύ ακριβά. Ωστόσο, μια μέση λύση που συνδυάζει την καλή συμπεριφορά του rebuilding και την ταχύτητα του local rebalancing είναι η partial rebuilding μέθοδος. Σε αυτήν την περίπτωση, ένα υπόδενδρο της δομής φτιάχνεται από την αρχή, όταν συμβεί παράβαση. Για να μπορέσουμε να εισάγουμε την τεχνική αυτή

στα $R^* - trees$, θα πρέπει να ορίσουμε πότε συμβαίνει παράβαση καθώς πλέον πιο ψηλοί κόμβοι πρέπει να έχουν γνώση για το πλήθος των entries που είναι αποθηκευμένα στα φύλα. Μεγάλο, λοιπόν, ρόλο θα παίξει το βάρος των κόμβων.

Ορισμός 2. *Έστω ένα WeR-tree. Ο κόμβος του δένδρου υ, με ύψος h_v και βάρος n_v θα βρίσκεται σε ισορροπία όταν και μόνο όταν*

$$m^{h_v} \leqslant n_v \leqslant M^{h_v}$$

Επομένως, οι πράξεις της ένθεσης και της αφαίρεσης απλά βρίσκουν τον πιο χαμηλό κόμβο (ξεκινώντας από κάποιο φύλο στο οποίο βρέθηκαν με τους τρόπους που εξηγήσαμε στα 2.1 και 2.2) ο οποίος δεν ικανοποιεί την συνθήκη της ισορροπίας και φτιάχνουν από την αρχή το υπόδενδρο του πατέρα του. Αν ο χαμηλοτερος κόμβος που βρέθηκε είναι η ρίζα τότε γίνεται bulk-loading σε όλη την δομή.

Θεώρημα 1. *Το επιμερισμένο κόστος της πρόσθεσης και της αφαίρεσης είναι $O(log^2 n)$.*

Απόδειξη. Θα αποδείξουμε το άνω φράγμα για την πράξη της προσθήκης. Ομοίως προκύπτει το αποτέλεσμα και για την αφαίρεση. Έστω, λοιπόν, ότι βρήκαμε τον χαμηλότερο κόμβο w που παραβιάζει την συνθήκη και είμαστε έτοιμοι να χτίσουμε από την αρχή το υπόδενδρο του πατέρα του, v, που βρίσκεται σε ύψος h_v. Επειδή το bulk-loading που εφαρμόζουμε δεν αλλάζει το ύψος του πατέρα, για το βάρος του n_v μετά το rebuilding ισχύει ότι:

$$m^{h_v} \leqslant n_v \leqslant M^{h_v}. \tag{3.1}$$

Δεν ισχύει, όμως το ίδιο για τον κόμβο w. Αν συμβολίσουμε με n_w και $n_w^{'}$, τα βάρη του αμέσως μετά το προηγούμενο rebuilding και αμέσως πριν αυτό το rebuidling τότε ισχύει ότι:

$$m^{h_v-1} \leqslant n_w \leqslant M^{h_v-1}$$

$$m^{h^{'}} \leqslant n_w^{'} \leqslant M^{h^{'}}$$

με $h^{'} > h_v - 1$. Επομένως

$$n_w^{'} - n_w \geqslant m^{h^{'}} - M^{h_v-1} \geqslant (cM)^{h^{'}} - M^{h_v-1} \geqslant kM^{h_v}$$

με $\frac{1}{M} \leqslant c \leqslant \frac{1}{2}$ και k μια κατάλληλη μη αρνητική παράμετρος με σωστή επιλογή της παραμέτρου c.[1] Από την 3.1 προκύπτει ότι:

$$n_w^{'} - n_w \geqslant kn_v$$

Αυτό σημαίνει, ότι μεταξύ δυο διαδοχικών rebuildings μεσολαβούν $\Omega(n_v)$ ενθέσεις. Έστω, ότι μεταξύ δύο διαδοχικών rebuildings συμβαίνουν X ενθέσεις. Τότε το επιμερισμένο κόστος στην χειρότερη περίπτωση είναι:

$$O(\frac{Xlogn + BL(n_v)}{X+1}) = \frac{O(Xlogn + BL(n_v))}{\Omega(X)} = O(logn) + \frac{O(BL(n_v))}{\Omega(X)} =$$

$$= O(logn) + \frac{O(BL(n_v))}{\Omega(n_v)} = O(logn) + O(\frac{BL(n_v)}{n_v}) = O(logn) + O(\frac{BL(n)}{n})$$

Λαμβάνοντας υπόψιν τους κόμβους σε όλο το μονοπάτι, τελικά το επιμερισμένο κόστος ισούται με $O(log^2 n) + O(logn\frac{BL(n)}{n})$. Το κόστος του rebuilding που χρησιμοποιούμε ,όμως, είναι $O(n)$,οπότε προκύπτει ότι το ζητούμενο επιμερισμένο κόστος είναι $O(log^2 n)$. □

[1] Για $c = 0.4$ και $M = 50$ που είναι συνήθεις ποσότητες, το k είναι μη αρνητικός αριθμός μέχρι ύψος $h_v = 4$. Επειδή, κιόλας, με αυτό το ύψος, μπορούν να δεικτοδοτηθούν πάνω από 312.000.000 δεδομένα, μας ενδιαφέρει, ακόμα, η ασυμπτωτική θεωρητική ανάλυση.

3.2 Bulk Loading

Σκοπός όλων των bulk loading τεχνικών είναι η πλήρης αξιοποίηση της πληροφορίας που μας δίνουν τα δεδομένα. Στις δομές που αναφέρονται στο 2.3, υπάρχουν αρκετές τέτοιες τεχνικές. Εμείς δανειζόμαστε την λογική των Hilbert R trees ([16]) και την λογική που προτείνουν οι Stefan Berchtold, Christian Böhm και Hans-Peter Kriegel ([7, 8]). Υλοποιούμε, δηλαδή, δυο εκδοχές των WeR-trees, κάθε μια από τις οποίες χρησιμοποιεί μια από τις παραπάνω Bulk Loading τεχνικές. Η δομή μας είναι ευέλικτη με την έννοια ότι μπορεί να δεχτεί οποιαδήποτε τέτοια μέθοδο. Η κατασκευή όμως του νέου υπόδενδρου, από την στιγμή που οι τεχνικές αυτές έχουν ταξινομήσει τα δεδομένα, είναι καθαρά θέμα και λειτουργία των WeR-Trees. Αυτό, δηλαδή, που κάνουν στην πραγματικότητα οι bulk loading μέθοδοι είναι ένα είδος ταξινόμησης.

1. Η πρώτη εκδοχή ταξινομεί τα δεδομένα με βάση την Hilbert τιμή τους. Τα δεδομένα είναι εκείνα που κατοικούν στο υπόδενδρο που πρόκειται να ανακατασκευαστεί. Αν και αυτή η ταξινόμηση κοστίζει $O(nlogn)$ στην χειρότερη περίπτωση, με κάποιες τροποποιήσεις καταφέρνουμε να κοστίζει μόλις O(1). Αν λάβουμε υπόψιν και το ότι πρέπει να μαζέψουμε τα δεδομένα πριν την ταξινόμηση, τότε το συνολικό κόστος είναι $O(n)$. Κατά κάποιο τρόπο πετυχαίνουμε τα δεδομένα να είναι ήδη ταξινομημένα.

2. Η δεύτερη είναι πιο πολύπλοκη. Τα δεδομένα ταξινομούνται διαδοχικά σε κάθε διάσταση με την βοήθεια του quicksort. Δεν μας ενδιαφέρει όμως η πλήρης ταξινόμηση. Θέλουμε να κόβουμε κάθε φορά το par-

tition στη μέση, μέχρι που το τμήμα αυτό να περιέχει το πολύ m δεδομένα καθώς δεν έχει νόημα περισσότερο σπάσιμο. Επομένως, γίνεται σαφές ότι χρειαζόμαστε και τον διάσημο αλγόριθμο k-select, ο οποίος επιστρέφει το στοιχείο που προσεγγιστικά είναι το median ενός πίνακα. Αυτό στη μέση περίπτωση κοστίζει συνολικά $O(n) + O(n) = O(n)$. Φυσικά, η χειρότερη περίπτωση είναι πιο άσχημη αλλά επειδή δουλεύουμε με μεγάλα data sets, η μέση χειρότερη περίπτωση είναι αρκετά αντιπροσωπευτική.

Έστω ότι πρόκειται να ξαναχτίσουμε το υπόδενδρο T_v που ορίζει ο κόμβος v. Αφού μαζέψουμε τα δεδομένα εφαρμόζουμε μια από τις παραπάνω τεχνικές. Το συνολικό κόστος όπως αναφέραμε καθορίζεται από το κόστος να διατρέξουμε όλα τα δεδομένα, το οποίο είναι:

$$BL(n) = \sum_{i=0}^{logn} M^i = \frac{nM - 1}{M - 1} = O(n)$$

όπως υποθέσαμε στο 3.1.

Από εκεί και πέρα το χτίσιμο του ανανεωμένου υπόδενδρου το αναλαμβάνει η δομή μας. Η ανακατασκευή του υπόδενδρου γίνεται με top-down τρόπο.

Αν αμέσως πριν το ξαναχτίσιμο, το υπόδενδρο είχε ύψος h_v, το ίδιο ύψος θα έχει και μετά το rebuilding. Αυτό που αλλάζει είναι ο αριθμός των παιδιών σε κάθε κόμβο του νέου υπόδενδρου. Σκοπός μας είναι η ομοιόμορφη κατανομή των δεδομένων στα φύλα. Προσπαθούμε, δηλαδή, όλα τα νέα φύλα να έχουν τον ίδιο αριθμό δεδομένων.

Πριν όμως αναλύσουμε την τεχνική μας, ας δούμε με ποιον τρόπο μεγιστοποιείται το storage utilization. Αυτό είναι ένα μέγεθος που δείχνει το πόσο καλά

αξιοποιούμε τους πόρους της δομής και ο τύπος του δίνεται στο 2.2. Έστω, λοιπόν, ότι, το υπόδενδρο T_v, ύψους 2 και βάρους N_v χτίζεται από την αρχή. Οι δύο ακραίες, λύσεις είναι οι εξής:

1. Ο κόμβος v έχει, πλέον, m υπόδενδρα(entries ή fanout), οπότε τα φύλα γεμίζουν με τον μέγιστο αριθμό δεδομένων ($\frac{N_v}{m}$).

2. Ο κόμβος v έχει, πλέον, M υπόδενδρα, οπότε τα φύλα γεμίζουν με τον ελάχιστο αριθμό δεδομένων ($\frac{N_v}{M}$).

Στην πρώτη περίπτωση, το storage utilization του υποδένδρου T_v είναι $\frac{m+N_v}{M+mM}$ ενώ στη δεύτερη είναι $\frac{M+N_v}{M+M^2}$. Αποδεικνύεται, εύκολα, ότι το storage utilization της πρώτης περίπτωσης είναι το καλύτερο. Γενικά, επειδή στα πιο χαμηλά επίπεδα έχουμε πολύ περισσότερους κόμβους, προτιμούμε να γεμίζουμε αυτούς πρώτα γιατί διαφορετικά, θα είχαμε κακή αξιοποίηση των πόρων. Η βελτίωση, πάντως, είναι γύρω από έναν παράγοντα 2. Ακόμα, το γεγονός ότι γεμίζουμε το φύλα με το μέγιστο αριθμό δεδομένων, οδηγεί την δομή σε πιο συχνές παραβάσεις άρα και rebuildings. Για αυτόν τον λόγο επιλέγουμε σε κάθε κόμβο να έχουμε τον μέσο αριρθμό υποδένδρων. Αυτό έχει σαν αποτέλεσμα, τα φύλα να έχουν περίπου $\frac{m+M}{2}$ δεδομένα, πάντα.

Υποθέτουμε, τώρα, ότι ο κόμβος v βρίσκεται σε ύψος h_v και έχει βάρος n_v. Μετά το χτίσιμο πρέπει κάθε κόμβος, ύψους h και βάρους n, να ικανοποιεί την σχέση:

$$m^h \leqslant n \leqslant M^h$$

Αν x είναι το fanout του v, τότε το βάρος των παιδιών του πρέπει να ικανοποιεί την σχέση:

$$m^{h_v-1} \leqslant \frac{n_v}{x} \leqslant M^{h_v-1}$$

γιατί όλα τα παιδιά μοιράζονται το βάρος του πατέρα τους σε ίσα μέρη. Γενικά, δεν έχει νόημα, οι κόμβοι ενός επιπέδου να μην φιλοξενούν το ίδιο πλήθος δεδομένων. Θέλουμε ίση και δίκαιη μεταχείριση και στους ενδιάμεσους κόμβους. Προκύπτει, επομένως, ότι:

$$\frac{n_v}{M^{h_v-1}} \leqslant x \leqslant \frac{n_v}{m^{h_v-1}} \tag{3.2}$$

Η παραπάνω σχέση, όμως, δεν εγγυάται ότι το x θα είναι ανάμεσα από m και M, που είναι απαραβίαστη συνθήκη στα δένδρα της $R-tree$ οικογένειας. Εξάλλου, χρειάζονται τα όρια αυτά για να μπορεί να καθοριστεί η χωρητικότητα της σελίδας (page capacity). Για αυτό η σχέση 3.2 μετασχηματίζεται στην

$$\max(\frac{n_v}{M^{h_v-1}}, m) \leqslant x \leqslant \min(\frac{n_v}{m^{h_v-1}}, M)$$

Ο ζητούμενος, επομένως, αριθμός x είναι επομένως

$$x = \frac{\max(\frac{n_v}{M^{h_v-1}}, m) + \min(\frac{n_v}{m^{h_v-1}}, M)}{2} \tag{3.3}$$

που είναι, ακριβώς, ο μέσος επιτρεπτός αριθμός παιδιών. Παρατηρείστε ότι όλοι οι κόμβοι του ίδιου επιπέδου έχουν το ίδιο fanout επειδή όλα τα παιδιά μοιράζονται το βάρος του πατέρα τους σε ίσα κομάτια.

Αυτή η αριθμητική τεχνική που μόλις περιγράψαμε, παρουσιάζει μια σημαντική αδυναμία: λόγω της διακριτής στρογγυλοποίησης που υποστηρίζουν όλες οι γλώσσες προγραμματισμού, το σφάλμα συσσωρεύεται μετά από επαναλήψεις, με αποτέλεσμα, είτε να έχουμε σε κάποιο κόμβο πάνω από M entries είτε κάτω από m. Παρόλο που οι παραπάνω σχέσεις είναι σωστές, δεν λαμβάνεται υπόψιν το σφάλμα των αριθμητικών πράξεων. Ας δούμε, λοιπόν, πώς λύνεται το πρόβλημα.

Έχει υπολογιστεί, λοιπόν, ο αριθμός x. Το κάθε ένα από τα x παιδιά φιλοξενεί $\lfloor \frac{n_v}{x} \rfloor$ δεδομένα και όχι απλά $\frac{n_v}{x}$ όπως έχουμε υποθέσει μέχρι τώρα. Επειδή $\lfloor \frac{n_v}{x} \rfloor x \neq x$, το τελευταίο παιδί ίσως να πρέπει να φιλοξενήσει πιο πολλά από $\lfloor \frac{n_v}{x} \rfloor$ για να μην μείνουν δεδομένα απέξω. Ή, μπορεί το κάτω όριο αυτό να μην ικανοποιεί, άμεσα, την αριστερή ανισότητα της σχέσης 3.3. Αυτό όμως σημαίνει και πιθανή παράβαση στις προϋποθέσεις της δομής καθώς τίποτα δεν μας εγγυάται ότι το βάρος του τελευταίου παιδιού n_w θα ικανοποιεί την σχέση:

$$m^{h_v-1} \leqslant n_w \leqslant M^{h_v-1}$$

Τη σχέση αυτή θα την ικανοποιηούσε μόνο αν το βάρος n_w υπολογιζόταν όπως περιγράψαμε στην 3.3. Για να είμαστε σωστοί, λοιπόν, πρέπει:

1. Αν $\lfloor \frac{n_v}{x} \rfloor$ είναι μικρότερο από το m^{h_v-1} να μειώσουμε το x (αυτός ο έλεγχος γίνεται για το πρώτο παιδί, δηλαδή, μόνο μια φορά στην αρχή, επομένως, πάλι τα βάρη είναι περίπου ίσα για όλα τα παιδιά).

2. Διαφορετικά, αν i είναι το εξεταζόμενο παιδί ($1 \leqslant i \leqslant x$) θα πρέπει να ισχύει $m^{h_v-1} \leqslant remaining - (x-i)\lfloor \frac{n_v}{x} \rfloor \leqslant M^{h_v-1}$ και αν όχι είτε να μειώνουμε κατά ένα το βάρος του i-στού παιδιού είτε να το αυξάνουμε. Το $remaining$ αναφέρεται στο πλήθος των δεδομένων του φιλοξενεί ο πατέρας v αλλά δεν έχουν ακόμα ανατεθεί σε κάποιο παιδί του.

Με την διαδικασία που έχουμε περιγράψει ως εδώ, υπολογίζονται τα fanouts και τα βάρη των κόμβων. Το νέο υπόδενδρο φτιάχνεται από πάνω προς το κάτω (top down approach), όπως θα έκανε ένας depth first αλγόριθμος. Και αυτό είναι λογικό γιατί το βάρος ενός κόμβου εξαρτάται από το βάρος του πατέρα του σύμφωνα με την τεχνική μας. Έτσι, όταν επισκεπτόμαστε έναν νέο κόμβο, εφαρμόζουμε την διαδικασία αυτή, μέχρι να φτά-

σουμε σε κάποιο φύλο, οπότε δημιουργούμε τα index records εκείνα που δεικτοδοτούν τα συγκεκριμένα δεδομένα. Όταν επιστρέψουμε από κάποιο κόμβο, προσαρμόζουμε τα MBRs του γιατί, πλέον, έχει τελειώσει πλήρως η κατασκευή των παιδιών του.

3.3 Σύγκριση με το $R^* - tree$

Εκτελέσαμε, ένα μεγάλο πλήθος πειραμάτων που αποδεικνύουν την ανωτερότητα του $WeR - tree$. Αυτά βασίστηκαν σε 4 σύνολα δυσδιάστατων δεδομένων. Τα πρώτα 2 είναι πραγματικά σύνολα, δηλαδή, δεδομένα που χρησιμοποιούνται από πραγματικές εφαρμογές. Το 3ο αποτελείται από ομοιόμορφα κατανεμημένα δεδομένα και το 4ο από δεδομένα που ακολουθούν zipfian κατανομή για να μπορέσουμε να ελέγξουμε το εμβαδό και την πυκνότητα των δεδομένων.

Η δομή μας γλιτώνει μέχρι και 50% προσβάσεις στον δίσκο και στους δυο τύπους ερωτήσεων για πραγματικά δεδομένα ενώ η αντίστοιχη βελτίωση για τα συνθετικά δεδομένα φτάνει μέχρι και το 90%. Εκτελέσαμε πειράματα με αναμειγμένες ακολουθίες από προσθέσεις, αφαιρέσεις και ερωτήσεις και διαπιστώσαμε ότι η δομή μας κάνει μέχρι και 31% λιγότερες προσβάσεις στον δίσκο. Πολύ καλά αποτελέσματα δίνει και η scaleup analysis. Εδώ μετρήσαμε την κλιμάκωση του πλήθους των προσβάσεων ανά ερώτηση σε σχέση με το πλήθος των δεδομένων. Η βελτίωση είναι γύρω στο 43%. Τέλος το storage utilization που επιτυγχάνεται αγγίζει την μονάδα (98.1%) ενώ στα $R^* - trees$ αυτό δεν ξεπερνά το 70%.

Για μια λεπτομερέστερη παρουσίαση τόσο των πειραματικών αποτελεσμάτων

οσο και της τεκμηρίωσης των WeR-trees ,ο αναγνώστης μπορεί να ανατρέξει στο Παράρτημα Α.

Appendix A

WeR-Trees

A.1 Introduction

Numerous applications, like Geographical Information Systems, CAD and VLSI design have emerged during the last years that demand the efficient manipulation of massive sets of geometric objects like points, lines, areas or volumes in one or more dimensions. The databases that accommodate this specific kind of objects are called spatial and they employ indexes that must be able to answer a very diverse set of geometric queries, like range queries, that ask for all objects lying within a given region, and nearest-neighbor queries, that seek for the object closest to a given object.

The diversity of the query repertoire combined with the massive nature of the involved datasets explains why practical, general purpose indices like R-trees attract so much research interest [19]. An R-tree is a height-balanced tree which can be considered as an extension of B+-tree for multi-dimensional data. The minimum bounding rectangle (MBR) of each

geometric object, along with a pointer to the address where the object actually resides, are stored into the leaves. Each internal node entry consists of a pair (pointer to a subtree $\mathcal{T}$, MBR of $\mathcal{T}$), with the MBR of a tree $\mathcal{T}$ defined as the MBR enclosing all the MBRs stored in $\mathcal{T}$. Like in B+-trees, each node contains at least m and at most M entries, where $m \leqslant M/2$. On the other hand, unlike B+-trees, a search query may activate several search paths from the root to the R-tree leaves, resulting, in the worst case, in a linear to the size of dataset performance just to retrieve a few objects. Figure A.1 illustrates an R-tree instance on a set S of rectangles.

Since its introduction in 1984, several variants of the R-tree have been proposed, each one aiming at improving the performance by tuning some parameters. Among the members of the "R-tree family", the most prominent one is the R*-tree of Beckmann et al. [4]. In R*-trees a number of heuristics were applied, like forced re-insertions during insertions (as in the case of deletions), buffering and optimization criteria for split-

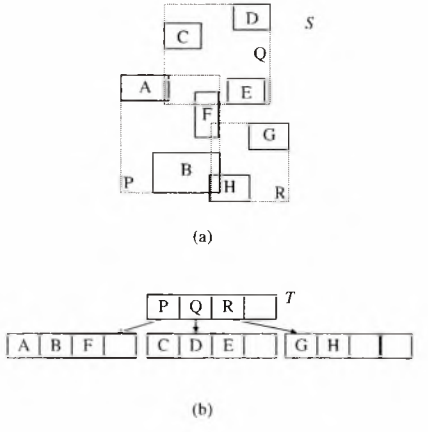


Figure A.1: An R-tree instance.

ting/merging nodes and adjusting the involved MBRs, so that R*-trees are widely accepted as achieving the best performance. However, the construction of any R-tree version by using repeated insertions does not guarantee the efficiency of the query performance; actually, the linear worst-case query time complexity cannot even be avoided.

On the other hand, bulk-loading techniques for R-trees do exist which capitalize on the priori knowledge of static datasets to build the structure from scratch. In this way, better utilization and search performance are achieved in the average case. This fact was the main motivation for the present work. Next, we briefly discuss bulk loading: [16] use the Hilbert sorting technique to impose a total order on the data and then build the R-tree according to the resultant sequence. [18] extended this approach, applying successive sorting and division of data into slabs for each of the dimensions. [9, 3] proposed the building of indices with repeated block-wise insertion, attaching auxiliary buffers to index pages. In [7] a recursive top-down algorithm is employed, which, operating in a manner similar to quick sort, determines the tree topology (height, fan-out, etc.) and uses a split strategy to bisect the data in secondary storage and construct the index directory in a depth-first, post order way. Finally, [2] introduced the Priority R-tree which exhibit a bulk-loading algorithm utilizing priority rectangles, in a way that an $O((n/M)^{1-1/d} + T/M)$ worst case performance is achieved, where M is the node capacity and d the data dimensionality.

Since it is easier to design data structures for static sets with good query time, a great number of efforts introduced general methods for converting static or semi-dynamic data structures into dynamic ones [5, 21, 20, 22].

This kind of research has been initially conducted in the area of main memory data structures. The methods can be classified into four main categories: (i) *partial rebuilding*, which enforces tree balance with subtree rebuilding; (ii) *global rebuilding*, allowing the structure to go out of balance before rebuilding it again, using only the valid data; (iii) *block-based segmentation* for query problems satisfying certain decomposability criteria; and (iv) *local balancing*, which maintain query efficiency by local changes to the tree structure. The applicability of (i)-(iv) depends on the specific characteristics of the transformed data structures, for instance, the rate of performance degradation or the ability to absorb insertion- or deletion-only operations.

Examining disk-based indexing solutions, various mobile indexes [11] in a loose way may be classified as members of the global rebuilding family, because of the limited horizon validity restriction they exhibit. R-trees can be considered of the last category, since their maintenance is based on local node adjustments along the path leading to the insertion or deletion location. On the other hand, the LR-tree [12] is an example of a block-based segmented index; it consists of a logarithmic number of component sub-structures, called blocks, organized as semi-dynamic, deletions-only R-tree-like indexes.

Although LR-tree outperforms R*-tree, it comprises a forest of indexes which may be difficult to be handled when one wants to combine it with single R-tree-like structures, say, in a merging ([25]) or join ([10]) operation. In this paper we use for the first time the technique of partial rebuildings in order to generate Weighted R-tree (WeR-tree), a very practical and well-

behaved single-tree member of the R-tree family. The rest of the paper is organized as follows. Section A.2 presents the partial rebuilding paradigm and Section A.3 introduces the WeR-trees. In Section A.4 we give theoretical bounds while Section A.5 provides experimental evidence on the superiority of our scheme over R*-trees. Finally, Section A.6 concludes our work.

A.2 The Partial Rebuilding Paradigm

Let us assume a tree structure index T already built for a set of items. Whenever one wants to insert or delete a point into/from T, the balance of the nodes along the involved search path changes. As a matter of fact several of them may become out of balance. The criteria of balance are specific to each data structure. For example, in B-trees, one node cannot accommodate fewer than $B/2$ or more than B children, B being the node capacity. Once violations are detected, balance must be restored.

The treatment of imbalance characterizes each data structure. B-trees, for instance, remedy violations applying local adjustments; repeated splitting/merging of nodes along the search path is employed until invariants are restored. In the partial rebuilding paradigm, the actions are more "elementary": one locates the highest problematic node v, and rebuilds the subtree T_v rooted at v in a *perfectly balanced* shape. "Perfect balance" is determined exclusively by the underlying data structure definition and the aiming properties.

In the above coarse description there is a missing point; the higher

the unbalanced node v the bigger the subtree T_v and therefore the more expensive its rebuilding. However, enforcing fewer expensive rebuildings and more cheaper ones, good amortized update times can be achieved [20, 22]. Usually this involves proving: (i) a lower bound on the number of updates performed on node v before becoming unbalanced since the last rebuilding of T_v; and (ii) an upper bound on the rebuilding cost in terms of the size of T_v. When (i) and (ii) are determined, the cost of rebuilding or bulkloading is charged to the updates that caused the reconstruction so that the total accumulated cost is bounded.

A.3 WeR-trees

A.3.1 Intuition

R*-tree is considered as the most efficient among the R-tree variations. Based on a number of carefully designed heuristics, it addresses the deficiencies of the original R-tree algorithms about query performance, improving the insertion phase. R*-tree introduces the forced reinsertion technique which avoids splits by reinserting a fraction of entries from an overflowed node. Regarding node splitting, R*-tree takes several factors into account: overlap between nodes, node perimeters and storage utilization. Also, it uses the plane-sweep technique to separate the node entries. However, deletion and searching are identical to the respective R-tree algorithms. The following theorem ([12]) summarizes the R*-tree performance:

Theorem 1. *A set S of n geometric items can be accommodated in a R*-tree using $O(n/M)$ space so that a search for an item has linear worst-case*

time complexity, an insertion of an item can be completed in $O(\log_M n \times M)$ *worst-case I/O time, while, given the location of an item, it can be deleted in* $O(\log_M n \times M)$ *worst-case I/O time (*M *denotes the node capacity or block size and I/O time refers to the number of block retrievals.)*

Additionally, we must note that several analytical works for the query operations do exist (see, for example, [15, 17, 23, 24]) and are characterized by limited generality since they simply derive approximate estimates based on a number of assumptions, like uniformity of the underlying distribution, known aspect ratio of MBRs, etc. Only in [2] a static R-tree variant with worst case performance guarantee is presented.

So, in the presence of actively changing datasets, R*-tree involves performance tuning with forced reinsertion on internal nodes v, i.e., a process that performs a kind of local, partial rebuilding of respective subtree. This motivated investigating of the replacement of the original update algorithms by carefully triggered perfect rebuilding of unbalanced subtrees. Bulk-loading techniques, that build the structure from scratch and achieve better utilization and search performance in the average case, exist for R-trees and are adopted to achieve the desired perfect balance.

A.3.2 Definition

The WeR-trees are defined as follows:

Definition 1. *A WeR-tree is an R-tree in which every internal node* v *is in perfect balance. An internal node* v *is in perfect balance if the respective subtree* T_v *is equivalent to a bulk-loaded R-tree on the same data set* S_v.

The invariant of the definition of the WeR-tree is enforced by carefully triggered subtrees' rebuilding, as specified by the update operations. Contrary to B+-trees, which can be reconstructed in a single way, R-tree-like structures permit a number of solutions, as we saw in Section A.1.

Insertion. Figure A.2 illustrates the insertion algorithm. Given the item p to be inserted, we firstly locate the leaf l into which p should reside. The search process depends on the kind of R-tree the adopted auxiliary bulk-loading method constructs—this will be exemplified in Section A.5, where a concrete experimental set up is described. Subsequently, we locate the deepest ancestor v of l such that the rebuilding of the subtree T_w of its child w on the path towards the tree root r would have bigger height from the other children of v, whereas the rebuilding of T_v would yield a subtree of the same height as before the insertion of p. In this way, we ensure that all leaves are at equal distance from r, while node capacities are within limits. Finally, the data items of T_v are passed to the bulk-loading procedure so that a new space and query efficient subtree is built. Figure A.3 illustrates a rebuilding example after inserting K. Since rebuilding at node w would increase the height of T_w, node v is chosen and T_v is bulkloaded from scratch.

Alternatively, one can use the insertion procedure described in Fig. A.4, that enforces the following invariant:

Invariant 1. *The subtree T_v of every node v of height h_v accommodates N_v items, such that $(cM)^{h_v} \leqslant N_v \leqslant M^{h_v}$, with M being the page capacity and $c < 1$ a properly chosen constant.*

Algorithm INSERT(**item** p, **node** r)

Input: The item p to be inserted into the WeR-tree with root node r

Output: The new instance of the WeR-tree accommodating p

1. Find the leaf page l that should accommodate p using the R-tree search procedure

2. Locate the deepest ancestor v of l such that

   (i) the rebuilding of the subtree T_w of its child w on the $l \longrightarrow r$ path would have bigger height from the other children of v; and

   (ii) the rebuilding of T_v would yield a subtree of the same height as before the insertion

3. Rebuild T_v, using bulk-loading

end of INSERT

Figure A.2: Insertion algorithm

Similarly to the previous algorithm, in the beginning we find the leaf l that should accommodate new item p. Next, we locate the highest ancestor v of l respecting the invariant while having a child w with more descendants

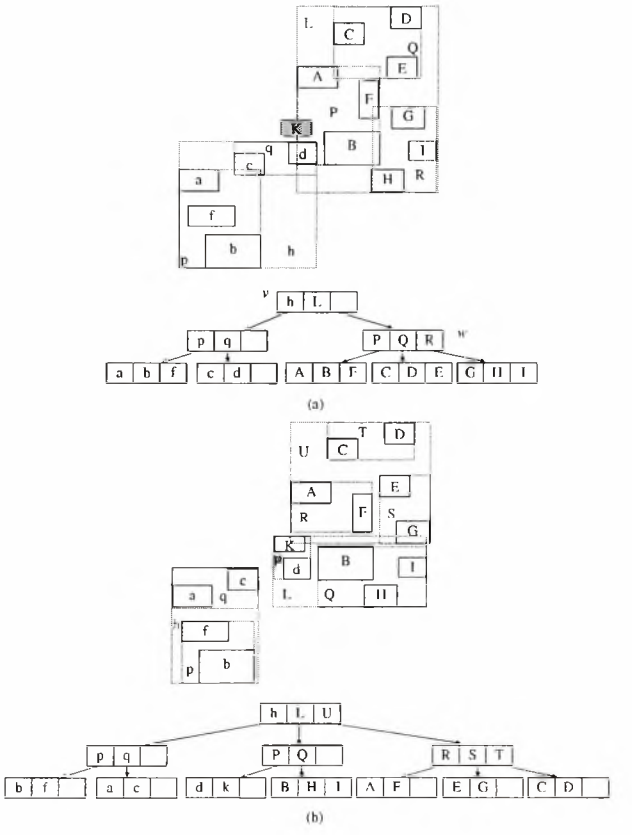


Figure A.3: A rebuilding example.

than the number the invariant suggests. After that, we substitute T_v with a brand new bulk-loaded subtree.

Deletion. As usually, deletion (Fig. A.5) is symmetrical to the insertion operation: After locating item p to be deleted, we locate the deepest ancestor v of accommodating leaf l such that the rebuilding of the subtree T_w of its child w on the path towards root r would have smaller height from

---

Algorithm INSERT2(**item** p, **node** r)

Input: The item p to be inserted into the WeR-tree with root node r

Output: The new instance of the WeR-tree accommodating p

1. Find the leaf page l that should accommodate p using the R-tree search procedure

2. Locate the highest ancestor v of l such that

   (i) it fulfills invariant 1; and

   (ii) its child w has more descendants than the number the invariant indicates

3. Rebuild T_v, using bulk-loading

end of INSERT2

Figure A.4: Alternative insertion algorithm

the other children of v, while the rebuilding of T_v would yield a subtree of unaltered height, and therefore, all leaves are at the same level while node capacities are within limits. Finally, T_v is rebuilt by the adopted bulk-loading procedure.

Alternatively, one can use the deletion procedure of Figure A.6, that

enforces invariant 1. Here, the goal is to find the highest ancestor v of involved leaf l such that obeys invariant 1, while having a child w with less descendants than the number the invariant suggests.

A.4 Theoretical Bounds

Since R-trees have linear worst case performance, we cannot prove better bounds. However, in the average case, one expects that the query performance of the WeR-trees is better than R*-trees, since its subtrees are regularly rearranged by the adopted bulk-loading procedure; Section A.5 provides evidence on this claim.

On the other hand, the reorganization is not prohibitively costly, as the following theorem describes:

Theorem 2. *The insertion and deletion procedures have $O(\log_M n\mathcal{BL}(n)/n)$ amortized complexity, with n the number of stored items, M the page capacity and $\mathcal{BL}(n)$ the time complexity of bulk-loading reconstruction.*

Proof. We will focus on the insertion procedure—the arguments about the deletion case are symmetrical. Let v and w be the farther-target of the rebuilding and the child whose height must be increased, respectively. Since the height h_v of v remains constant before and after bulk-loading, the subtree T_v contains n_v leaf pages, such that:

$$m^h \leqslant n_v \leqslant M^h, \quad m = cM,\ c < 1.$$

On the other hand, let n_w, n'_w be the number of leaf pages T_w contains immediately after the last rebuilding of v and immediately before the new

rebuilding of v, respectively. Then, the following equations hold:

$$\begin{aligned} m^{h_v-1} &\leqslant n_w \leqslant M^{h_v-1} \\ m^{h'} &\leqslant n'_w \leqslant M^{h'}, \quad h_v - 1 < h'. \end{aligned}$$

Hence, we have:

$$\begin{aligned} n'_w - n_w &\geqslant m^{h'} - M^{h_v-1} \geqslant (cM)^{h'} - M^{h_v-1} \\ &\geqslant c^{h'} M^{h'} - M^{h_v-1} \geqslant kM_v^h \end{aligned}$$

with k a properly chosen constant. Therefore, between the last and the new rebuilding, $\Omega(n_v)$ insertions took place. This means $O(\mathcal{BL}(n_v)/n_v)$ amortized complexity per insertion due to node v. Summing up the amortized costs along the leaf-to-root path, the claimed insertion complexity follows. □

Usally, bulk-loading algorithms cost $O(n \log_M n)$ time, which gives an $O(\log_M^2 n)$ amortized update complexity. The previous proof was based on the first version of insertion/deletion algorithms, which are height-balanced ones, while the alternative versions can be characterized as weight-balanced. In [13] the equivalence of height-balanced and weight-balanced binary trees was proved. While R-trees are k-way trees, they can easily be converted into binary ones. By extending the arguments of [13], one can show that the same bounds also hold for the alternative versions of the update operations.

A.5 Performance Results

A.5.1 Experimental Set up and Heuristics

We have implemented all examined methods in C, using the same components for the common tasks among the methods. All experiments were performed on a machine with a Pentium IV processor at 2.3 GHz, with 1GB main memory and with a 10 GB hard-drive. We used both real and synthetic data sets which contain two, three and four-dimensional points. Specifically, we present results for the LB data set, which contains 53,000 two-dimensional points representing postal address at Long Beach, and the NE data set, which contains 123,593 two-dimensional points representing postal addresses of three metropolitan areas (New York, Philadelphia and Boston). Both the aforementioned data sets have been used as benchmarks in prior work (e.g., [16]). For synthetic data sets we present results for two, three and four -dimensional points following uniform and zipfian distribution. We used a default page size equal to 8KB (other page sizes gave analogous results).We also used the k-nearest neighbor algorithm that is described in [6].

We tested our structure using two bulk-loading techniques. The first one is inspired by the algorithms described in [16], that is, the data points at every time are sorted according to their Hilbert value. The second technique is based on the approach that is described in [7] where the data points that are about to be bulk loaded are sorted consequently in each dimension according to their values. The first and the second implementation of our WeR-tree will be termed in the sequel as WeR-A and WeR-B

respectively. Notice that in contrast with WeR-A, WeR-B sorts the data points of a subtree that is going to be rebuilt, invoking the quicksort algorithm, so we expect bigger execution times compared with these of R^*-tree and WeR-A tree. More over, we have to mention that the bulk load technique that is adopted by the WeR-B tree is not efficient for low dimenional points as the author of [7] imply. The superiority of this technique is apparent in data sets that contain points of dimensionality over than 8. In our case, where the dimensionality is 4 at most, someone would expect that WeR-B tree query performance is worst than R^* tree. On the contrary, we will show that, althought the bulk loading technique of WeR-B is not suggested for low dimenional points, the well tuned WeR structure balances the bad performance of this technique.

The rebuilding in both implementations follows a top down fashion. During its way down, the algorithm decides the appropriate number of entries (subtrees) at every node. If the current node v (with weight of N_v data points) has height h_v then the fanout of this node is defined as:

$$x = \frac{\min(\frac{N_v}{m^{h_v-1}}, M) + \max(\frac{N_v}{M^{h_v-1}}, m)}{2},$$

and the weight of each of the x subtrees is therefore defined as $\frac{N_v}{x}$. With this method, all the leaves have the same number of accommodated data points and most important they are filled with the average allowed number. What we succeed is that we limit the bulk loading operations in the presence of mixed update operations because the leaves are neither fully nor least filled. During its way up, the top down algorithm adjust the corresponding MBRs (and the Largest Hilbert Values in the case of WeR-A implementation) as a classic depth-first algorithm would do.

For convenience, we adopt the approach of prior work and consider square-shaped range queries, characterized by the size of the square. We are interested in the relative performance of the examined methods; therefore, we use a path-buffer (containing the current path from root to leaf) but no other buffer space, to clearly examine the behavior of the methods regardless the effect of buffering.

Experimental Results

Our first experiment considers range search queries on both real and synthetic data sets that contain two dimensional points. As performance measure we use the number of page accesses. We assume six query files (Q1)-(Q6) of 100 intersection window queries each, all of them uniformly distributed over the data space. The area of the window queries of each query file (Q1)-(Q6) varies from 0.001%, 0.01%, 1%, 1.5%, 1.8% to 2% relatively to the area of the whole data space. Figures A.7–A.8 illustrate the results with respect to the range query size; the latter is expressed in terms of the percentage (%) of the work space.

Considering the real data sets, we observe in Figure A.7 that WeR-A and WeR-B trees clearly outperform R*-tree:savings vary between 20% and 35%. Analogous conclusions can be drawn from the synthetic data set with uniformly distributed points (Fig. A.8(a)), where gains reach 35%. For the synthetic data set with points following a Zipf distribution (Fig. A.8(b)), the WeR trees compares favorably to the R*-tree. The WeR-tree family presents an improvement between 68% and 85%. This can be justified by the demanding nature of the zipfian data set; the distribution of both

the points and the queries is very skewed. On the other hand, WeR-trees demonstrate better organization which evidently deals with this case.

Then, we studied the k-nearest neighbor queries. We present results only for the real data sets since the findings for the synthetic ones are analogous. For every different value of parameter k, we performed 100 queries with point queries uniformly distributed over the whole data space. The average page accesses are given in Figure A.9. It is apparent that in all cases, the WeR trees have better performance than the R*-tree; savings vary from 27% to 49%. Even though the WeR-B invokes a inefficient bulk loading technique, finally it is still better than the R^*-tree.

Our next experiments evaluate the behavior of the update operations. As performance main measure we use the wall-clock time. Update operators involve sorting, plane-sweeping of node entries (during R*-tree split), finding the entries to be reinserted (during R*-tree reinsertion), calculating the subtrees for bulk-loading (during WeR-A and WeR-B tree construction phase) and other operators which require non-negligible CPU cost, additional to the I/O cost (which is not the case for search queries, where I/O cost is the dominant one). Therefore, wall-clock time considers both these two cost factors.

Insertion is the first operation we deal with. We give results only for the NE data set; the other data sets led to the same conclusions. We tested all methods by initially inserting a number of points from the data set, and measuring the insertion time with respect to the number of remaining points. The number of remaining points is expressed as percentage of the total number of points in the data set. Figure A.10(a) illustrates the

findings. As depicted, the WeR trees outperform the R*-tree. Although WeR trees spend much time to reconstruct the subtrees, they consume much less time to find the appropriate leaf in where the new point will reside. This is because the ChooseSubtree() method of the R*-tree costs $O(pM \log n)$ worst case while the relative method of the WeR-A tree only $O(M \log n)$. Additionally, the good MBR packing of WeR structures leads to fewer intersections and fewer activated paths. As a result, despite the fact that WeR-B has an identical ChooseSubtree() function to R^*-tree and invokes the time consuming quicksort algorithm, it is slower than R^*-tree. For this experiment we also give in Figure A.11, the average number of page accesses per operation, which follows the theoretical analysis.

Additionally, we investigated the execution time of the deletion operation. Figure A.10(b) illustrates the results with respect to the number of deleted points. Similarly as above, the WeR trees perform better than the R*-tree.

Next, we simulated a typical workload with a sequence of mixed deletion, insertion and range query operations in order to examine the impact of interleaved updates on query performance. Initially, all the dataset points were inserted and no deletion is performed. For the remaining points, the ratio, C, of the number of inserted points to the number of deleted points defines the experimental parameter. Here we must note that deletions were carried out by removing already accommodated points. Interleaved with insertions and deletions, range queries were performed and the average number of disk accesses needed by the range queries was determined. Figure A.12(a) illustrates the results with respect to C for the LB dataset.

As shown, the WeR-A clearly outperforms the R*-tree, while the performance of WeR-B is close to R^* but still better: performance gains range from 18.5% to 20.3%.

We also studied the scale-up properties of each method with respect to the dataset size, using synthetic datasets which follow the uniform distribution. Figure A.12(b) depicts the results for the range search query (with uniformly distributed windows queries from the $Q6$ query file) for increasing number of points in the dataset (given in thousands). As shown, WeR trees outperform R*-tree, since they require 24%–38% less page accesses. Moreover, we observe that the performance of the WeR-A tree grows linearly with respect to the number of inserted data points in a slow way. This can be explained as follows: the Hilbert values of the data points are distributed uniformly over the whole data space because the Hilbert value is a linear function of uniform variables (the coordinates of the data points follow uniform distribution). This means that after consecutive reconstructions, where the data points are sorted according to their Hilbert value, the data points in the WeR-A tree are uniformly distributed in the leaves. Hence, queries with uniformly distributed windows (and this is the case) activate only a few paths. On the contrary, the packing of the R*-tree does not guarantee that the data points are stored in the leaves uniformly. As a result, the more the inserted points are, the more paths are activated due to bad packing.

In addition, the WeR*-tree owes his good performance to the storage utilization it achieves since the data points are distributed uniformly among the leaves. While R*-trees use the 70.4% of their resources at most, WeR-A

trees reach up to 98.1%. Figure A.13 shows the storage utilization for all the data sets used in our experiments, after the insertion of all the data points in each case. The superiority of the WeR-A trees is obvious in every data file. As far as the R*-tree is concerned, we observe a performance improvement in the uniform data file but still is much worst than the WeR-A tree.

We also display, in Figure A.14 and Figure A.15, the range query performance with the six query files (Q1)-(Q6) with respect to the size of cache. This is our last experiment involving two dimensional points with which we want to show that WeR family trees are better independently of any cache size. Here, we use the LB real data set. We also present, in Figure A.16, the k-nn query performance (using the NE real data set) with respect to the size of cache as well to enhance our claim that the performance of WeR trees is better in any cache size. Similar results appear for all data files in both cases.

The rest of this section considers three and four-dimensional points. We used a data set of 123,000 3D points that follow uniform distribution. We measured the range query and k nearest neighbor performance with respect to the cache size. For the range queries, we used nine query files (Q1)-(Q9) that consist of 100 window queries each. Every window query in each file is distributed uniformly over the whole data space and has volume equal to 0.001%, 0.01%, 1%, 1.5%, 1.8%, 2%, 5%, 10% and 20% of the whole data volume respectively. Figure A.17 and Figure A.18 depicts the results. Obviously, the WeR family trees have better performance compared to this of R^*-tree: saving gains vary from 23% to 32%. Analogous

conclusions are derived from Figure A.19, where the k nearest neighbor performance is displayed with respect to cache size: gains reaches up to 35%.

Finally, we examined the performance of our structures regarding four dimensional points. For the following experiments we used a data set that contains 123,000 4D points. These points are distributed uniformly over the whole data volume. We measured the range query performance with the nine query files (Q1)-(Q9) that we have already defined in the above paragraph, varying the size of cache. The results are shown in Figure A.20. We observe that WeR-A is much better than R^*-tree.The performance of WeR-B is similar to this of R^*-tree but still better. The superiority of WeR-B is more obvious when large window queries take place. Saving gains reaches up to 35%. Our last experiment invokes k nearest neighbor queries. The results are depicted in Figure A.21 with respect to the cache size. In all cases, WeR tree structures outperform the R^*-tree. Savings vary from 15% to 35%.

A.6 Conclusion

We presented a new scheme, called WeR-tree, for the dynamic manipulation of large datasets. WeR-trees are the first members of the R-tree family which employ the technique of partial rebuildings, i.e., a gradual reconstruction of subtrees in a carefully triggered way. The update operations are theoretically investigated and proven to be of $O(\log_M n\mathcal{BL}(n)/n)$ amortized complexity. Additionally, the basic geometric queries are thor-

oughly tested with carefully designed experiments which confirm the applicability of the proposed method and demonstrate its superiority over R*-trees: utilization reaches up to 98.1%, range query savings vary 20%-35% on real data sets and 15% - 85% on synthetic data, knn queries are 27%–49% cheaper, during mixed operations performance savings are between 18.5% and 20.8%, while the scheme scales up linearly with respect to the number of inserted points achieving 24%–38% less page acesses. These gains resulted from a careful employment of a top-down reconstruction method. For future work, we would like to investigate how repeated reconstruction can be tuned in conjunction with designing specially tailored bulk-loading methods. Towards this end, merging subtree methods ([25]) may be proven helpful.

Acknowledgements. The authors would like to thank D. Katsaros and A. Nanopoulos for their useful comments and constructive discussions.

Algorithm DELETE(**item** p, **node** r)

Input: The item p to be deleted from the WeR-tree
with root node r

Output: The new instance of the WeR-tree
with p deleted

1. Find the leaf page l that accommodates p
using the R-tree search procedure

2. Locate the deepest ancestor v of l such that

(i) the rebuilding of the subtree T_w of its child w
on the $l \longrightarrow r$ path would have smaller height from
the other children of v; and

(ii) the rebuilding of T_v would yield a subtree of
the same height as before the deletion

3. Rebuild T_v, using bulk-loading

end of DELETE

Figure A.5: Deletion algorithm

Algorithm DELETE2(**item** p, **node** r)

Input: The item p to be deleted into the WeR-tree
with root node r

Output: The new instance of the WeR-tree
with p deleted

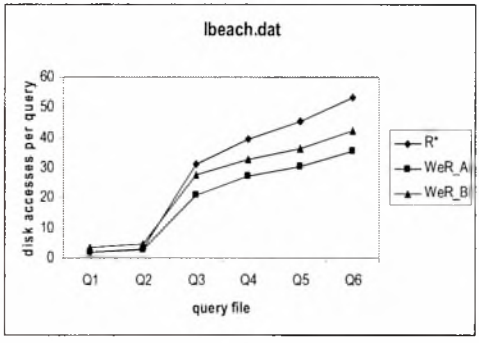
1. Find the leaf page l that accommodates p
using the R-tree search procedure

2. Locate the highest ancestor v of l such that
(i) complies with invariant 1; and
(ii) its child w has less descendants than the number
the invariant indicates

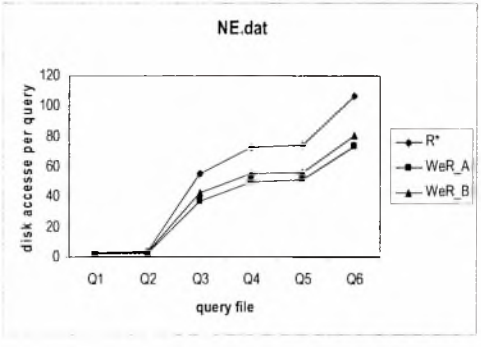
3. Rebuild T_v, using bulk-loading

end of DELETE2

Figure A.6: Alternative deletion algorithm

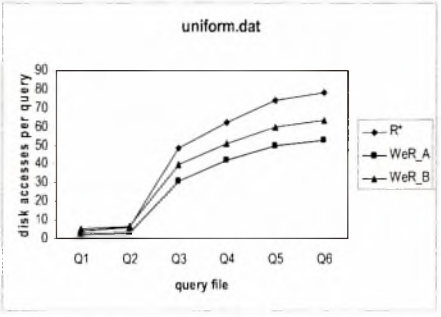


(a)

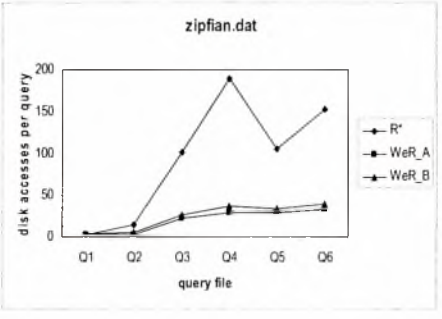


(b)

Figure A.7: Range queries on real data sets: (a) Long Beach, (b) North East.

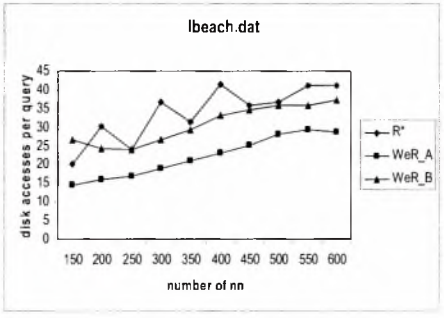


(a)

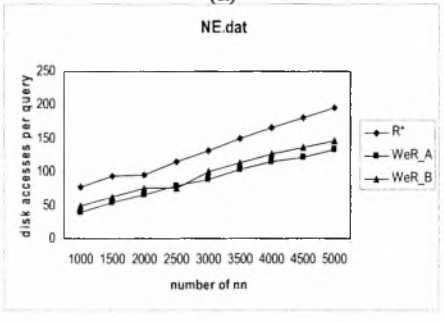


(b)

Figure A.8: Range queries on synthetic data sets: (a) Uniform, (b) Zipfian.

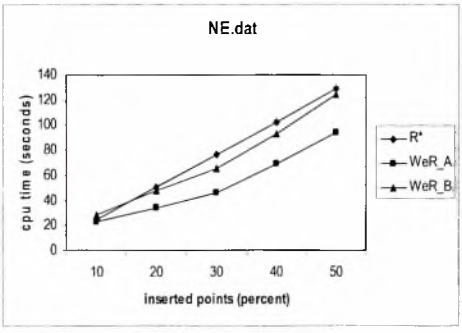


(a)

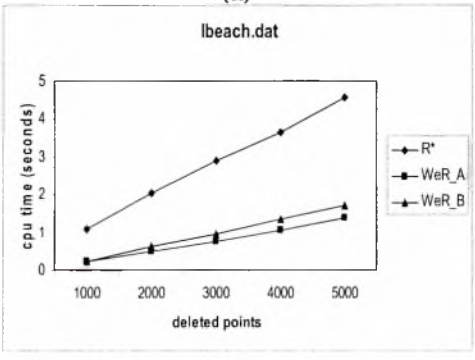


(b)

Figure A.9: k nearest neighbor query: (a) Long Beach real data set. (b) North East real data set.



(a)



(b)

Figure A.10: Update operations: (a) Insertions. (b) Deletions.

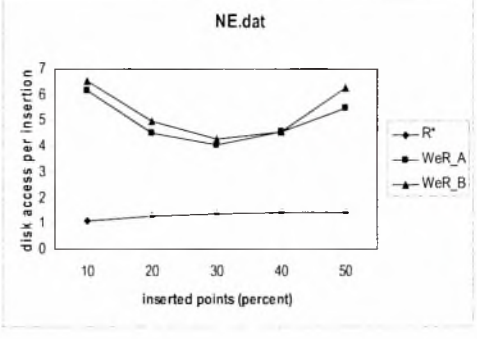
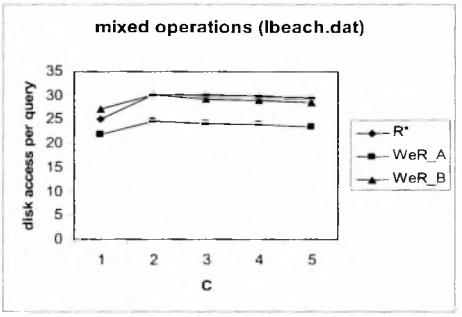
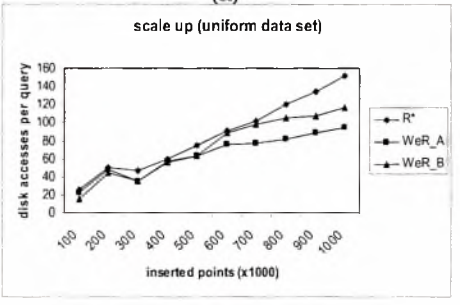


Figure A.11: Average number of page accesses per insertion.



(a)



(b)

Figure A.12: Query performance with respect to: (a) Mixed operations, (b) Scale-up.

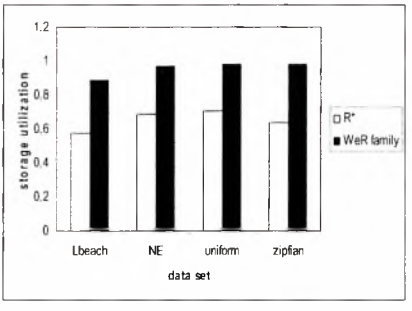
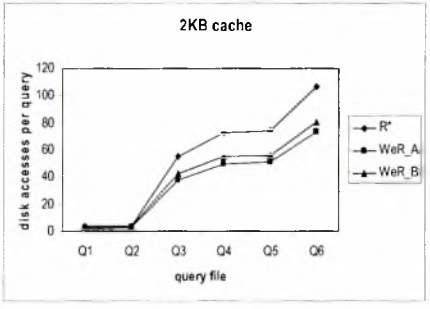
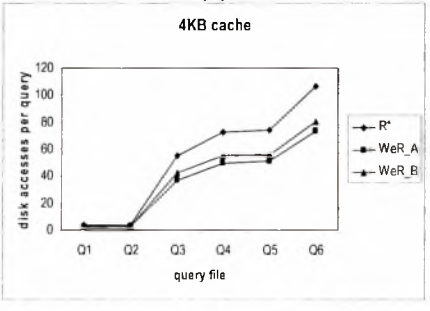


Figure A.13: Storage utilization.

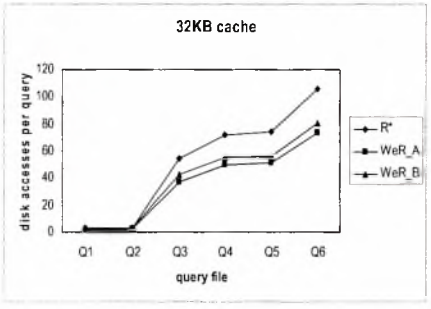


(a)

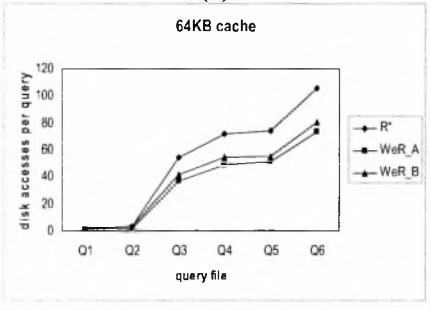


(b)

Figure A.14: Range Query Performance with respect to: (a) 2KB cache. (b) 4KB cache.

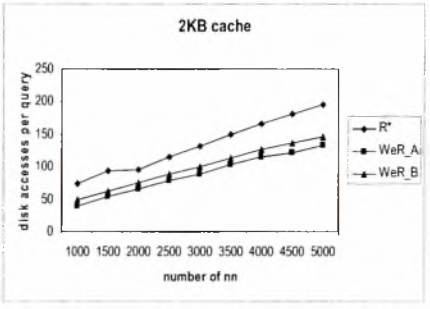


(a)

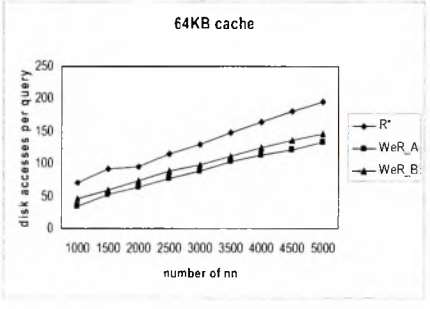


(b)

Figure A.15: Range Query Performance with respect to: (a) 32KB cache. (b) 64KB cache.

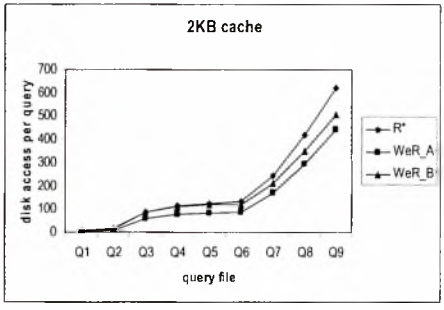


(a)

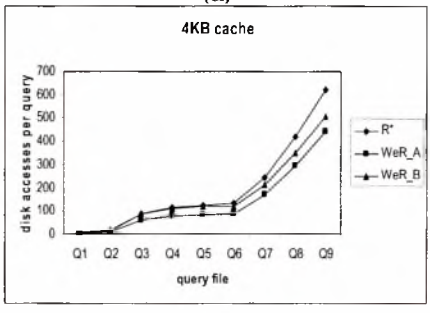


(b)

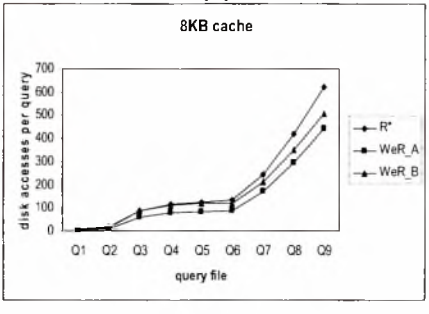
Figure A.16: k Nearest Neighbor Query Performance with respect to: (a) 2KB cache. (b) 64KB cache.



(a)

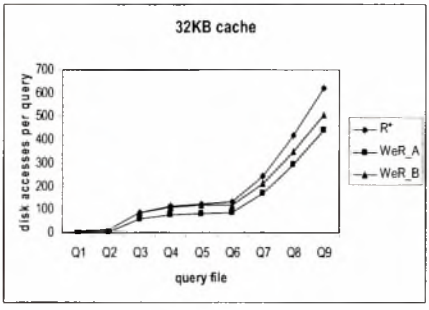


(b)

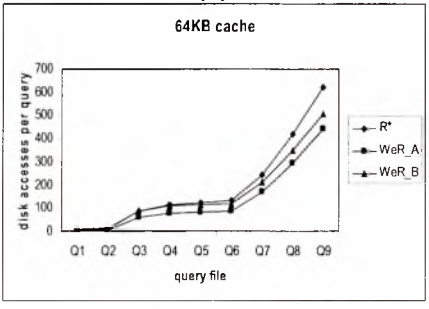


(c)

Figure A.17: Range Query Performance considering 3D points with respect to: (a) 2KB cache. (b) 4KB cache. (c) 8KB cache.

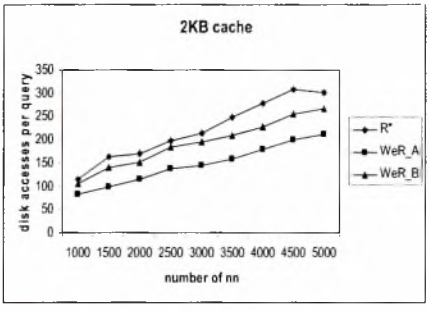


(a)

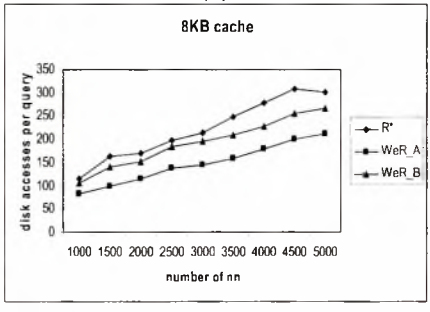


(b)

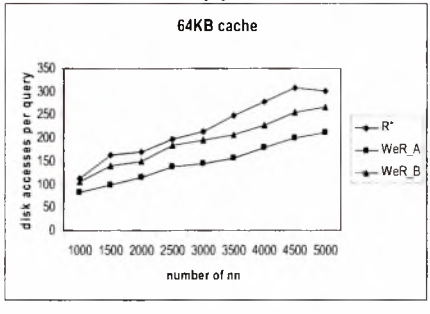
Figure A.18: Range Query Performance considering 3D points with respect to: (a) 32KB cache. (b) 64KB cache.



(a)

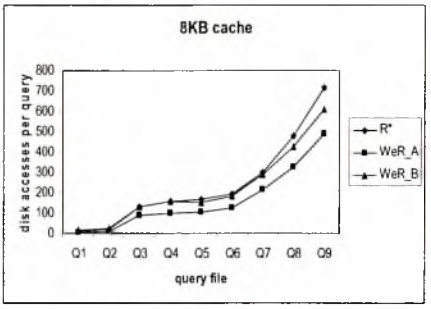


(c)

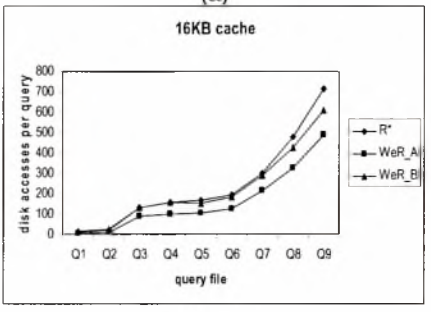


(e)

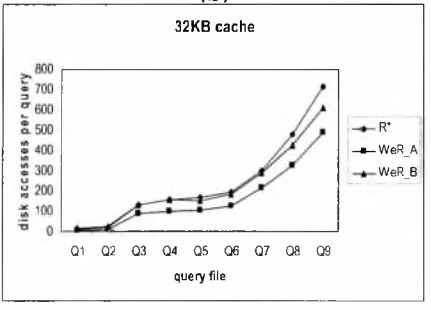
Figure A.19: k Nearest Neighbor Query Performance considering 3D points with respect to: (a) 2KB cache. (b) 8KB cache. (c) 64KB cache.



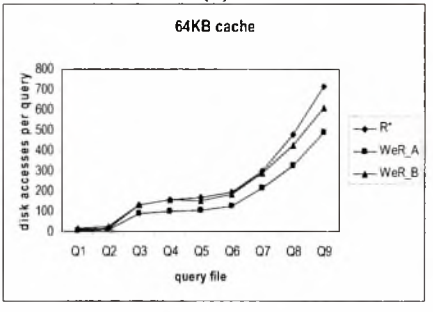
(a)



(b)



(c)



(d)

Figure A.20: Range Query Performance considering 4D points with respect to: (a) 8KB cache. (b) 16KB cache. (c) 32KB cache. (d) 64KB cache.

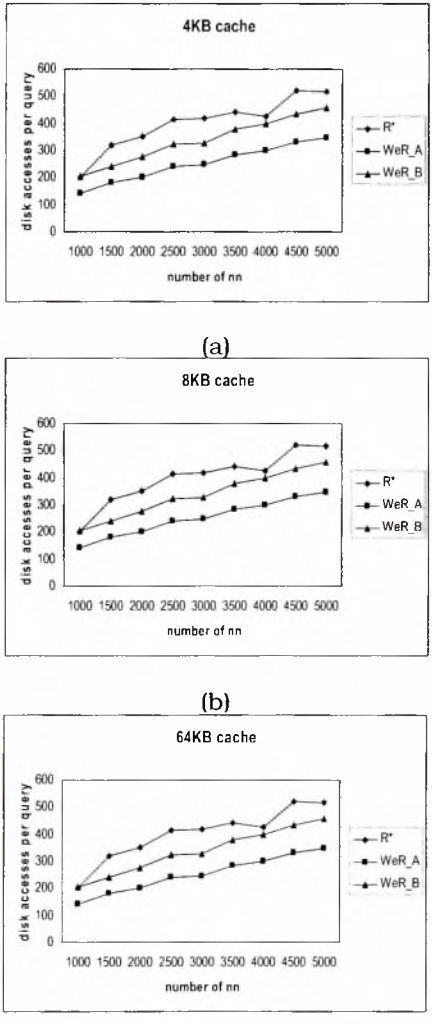


Figure A.21: k Nearest Neighbor Query Performance considering 4D points with respect to: (a) 4KB cache. (b) 8KB cache. (c) 64KB cache.

Bibliography

[1] P.K. Agarwal, L. Arge, O. Procopiuc, and J.S. Vitter, "A Framework for Index Bulk Loading and Dynamization", in *Proc. ICALP*, pp. 115-127, 2001.

[2] L. Arge, M. de Berg, H.J. Haverkort, K. Yi, "The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree", in *Proc. SIGMOD*, pp.: 347-358, 2004.

[3] L. Arge, K. Hinrichs, J. Vahrenhold, and J. Vitter, "Efficient Bulk Operations on Dynamic R-tress", in *Proc. ALENEX*, pp. 328-348, 1999.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles", in *Proc. SIGMOD*, pp. 322-331, 1990.

[5] J.L. Bentley, and J.B. Saxe, "Decomposable Searching Problems I: Static-to-Dynamic Transformations", *J. Algorithms*, **1**, 301-358, 1980.

[6] S. Berchtold, C. Böhm, D.A. Keim, and H.-P. Kriegel, "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space", in *Proc. PODS*, pp. 78-86, 1997.

[7] S. Berchtold, C. Böhm, and H.-P. Kriegel, "Improving the Query performance of High Dimensional Index Structures by Bulk Load Operations", in *Proc. EDBT*, pp. 216-230, 1998.

[8] C. Böhm, and H.-P. Kriegel, "Efficient Bulk Loading of Large High-Dimensional Indexes", in *Proc. DaWak*, 1999.

[9] J. Bercken, P. Widmayer, and B. Seeger, "A Generic Approach to Bulk Loading Multidimensional Index Structures", in *Proc. VLDB*, pp. 406-415, 1997.

[10] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees", in *Proc. SIGMOD*, pp. 237-246, 1993.

[11] P. Bozanis, "Indexing Mobile Objects: an Overview of Contemporary Solutions", Chapter XI in *Wireless Information Highways*, pp. 315-338, IRM Press, 2005.

[12] P. Bozanis, A. Nanopoulos, and Y. Manolopoulos, "LR-tree: a Logarithmic Decomposable Spatial Index Method", *Comput. J.*, 46(3), 319-331, 2003.

[13] I. Galpering, and R.I. Rivest, "Scapegoat Trees", in *Proc. SODA*, pp. 165-174, 1993.

[14] A. Guttman, "R-trees: a Dynamic Index Structure for Spatial Searching", in *Proc. SIGMOD*, pp. 47-57, 1984.

[15] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, "Spatial Joins Using R-trees: Breadth-first Traversal with Global Optimizations", in *Proc. VLDB*, pp. 396-405, 1997.

[16] I. Kamel, C. Faloutsos, "Hilbert R-tree: an Improved R-tree using Fractals", in *Proc. VLDB*, pp. 500-509, 1994.

[17] C.A. Lang, and A.K. Singh, "Modeling High-Dimensional Structures using Sampling", in *Proc. SIGMOD*, pp. 389-400, 2001.

[18] S. Leutenegger, M. Lopez, and J. Edgigton, "STR: a Simple and Efficient Algorithm for R-tree Packing", in *Proc. ICDE*, pp. 497-506, 1997.

[19] Y. Manolopoulos, Y. Theodoridis, and V. Tsotras, *Advanced Database Indexing*, Kluwer Academic Publishers, Boston, 1999.

[20] K. Mehlhorn, *Data Structures and Algorithms, vol. 1: Sorting and Searching*, Springer-Verlag, Berlin, Heidelberg, 1984.

[21] K. Mehlhorn, and M.H. Overmars, "Optimal Dynamization of Decomposable Searching Problems", *Inf. Proc. Lett.*, 12, 93-98, 1981.

[22] M.H. Overmars, *The Design of Dynamic Data Structures*, Springer-Verlag, Berlin, Heidelberg, 1983.

[23] B.-U. Pagel, H.-W. Six, and M. Winter, "Window Query-Optimal Clustering of Spatial Objects", in *Proc. PODS*, pp. 86-94, 1995.

[24] A. Papadopoulos, and Y. Manolopoulos, "Performance of Nearest-Neighbor Queries in R-trees", in *Proc. ICDT*, pp. 394-408, 1997.

[25] V. Vasaitis, A. Nanopoulos, and P. Bozanis, "Merging R-trees", in *Proc. SSDBM*, pp. 141-150, 2004.

[26] Li Chen, Rupesh Choubey and Elke A.Rundensteiner, "Bulk-Insertions into R-Trees Using the Small-Tree-Large-Tree Approach", in *Proc. ACM*, 1998.