ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΤΟΥ KDB TREE

Κώστας Αποστόλου

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων Καθηγητής: Γ.Ευαγγελίδης
Εξεταστές: Γ.Ευαγγελίδης, Ν.Σαμαράς, Μ.Σατραζέμη,

ΔΠΜΣ στα Πληροφοριακά Συστήματα

Πανεπιστήμιο Μακεδονίας
Θεσσαλονίκη

Φεβρουάριος 2010

ΠΕΡΙΛΗΨΗ

Στην διπλωματική εργασία γίνεται υλοποίηση της πολυδιάστατης δομής δεικτοδότησης KDB Tree με βάση τους αλγορίθμους που προτείνονται στο [Robinson 81] Η υλοποίηση της δομής αντιμετωπίζεται και από την σκοπιά του σχεδιασμού λογισμικού και γίνεται περιγραφή της διαδικασίας σχεδιασμού που θεωρήθηκε πιο κατάλληλη για την ανάπτυξη ενός τέτοιου προγράμματος, μαζί με παρουσίαση τεχνολογιών λογισμικού που μπορούν να βοηθήσουν σε αυτό το σκοπό. Στην δομή εκτελούνται δοκιμές για συλλογή στατιστικών και μελέτη της απόδοσης της καθώς οι διαστάσεις αυξάνονται ενώ αναπτύχθηκε και εφαρμογή γραφικής απεικόνισης της σε δύο διαστάσεις.

ΠΕΡΙΕΧΟΜΕΝΑ

1. ΠΕΡΙΓΡΑΦΗ ΤΗΣ ΕΡΓΑΣΙΑΣ

Στην διπλωματική εργασία μου ανατέθηκε να προγραμματίσω μια υλοποίηση της δομής δεικτοδότησης kdb tree. Σε αυτό το κείμενο παρουσιάζω από θεωρητική πλευρά το kdb tree με αφετηρία το αρχικό paper του Robinson που δημοσιεύτηκε το 1981 ενώ επισημαίνω και κάποια σημεία που δεν παρουσιάστηκαν σε αυτό με λεπτομερή τρόπο, σχετικά κυρίως με τα προβλήματα που έχει η υλοποίηση των δοθέντων στο paper αλγορίθμων.

Επίσης γίνεται μια συνοπτική παρουσίαση του προβλήματος της πολυδιάστατης δεικτοδότησης με σκοπό το kdb tree να τοποθετηθεί σε ένα γενικότερο πλαίσιο ανάλογης λειτουργικότητας δομών και να αναφερθούν οι διαφορές του με αυτές.

Άλλα μέρη αυτού του κειμένου έχουν να κάνουν με τον σχολιασμό του κώδικα C# με τον οποίο έγινε η υλοποίηση, με την αντιμετώπιση του προβλήματος υλοποίησης μιας τέτοιας δομής από πλευράς σχεδιασμού λογισμικού, καθώς και με την σύντομη παρουσίαση κάποιων άλλων τεχνολογιών όπως οι XSLT LINQ και XAML που βοήθησαν στο να κατασκευαστεί το πρόγραμμα.

Τέλος αναλύονται τα αποτελέσματα μετρήσεων αποδοτικότητας της δομής με τις ίδιες μετρικές που είχαν προταθεί από τον Robinson.

Εκτός από το κείμενο η εργασία περιλαμβάνει κώδικα που υλοποιεί ένα kdb tree για οποιαδήποτε πλήθος διαστάσεων(πλην των διαδικασιών διαγραφής που δεν ζητήθηκαν), και ένα πρόγραμμα όπου δίνεται πρόχειρο visualization της δομής καθώς γίνεται εισαγωγή σημείων στον δύο διαστάσεων χώρο.

2. ΕΙΣΑΓΩΓΗ ΣΤΟ KDB TREE

2.1 Ορισμός

Το kdb-tree είναι μια δομή δεδομένων για την δυναμική δεικτοδότηση αρχείων με πολυδιάστατα κλειδιά εγγραφών, κατάλληλη για χρήση με μονάδες δευτερεύουσας μνήμης.

2.2 Συσχετισμοί με άλλες δομές δεδομένων

Η δομή προτάθηκε το 1981 από τον Robinson και αποτελεί μια προσπάθεια να συνδυαστούν τα χαρακτηριστικά του B-tree και του kd-tree.

Το b-tree είναι ένα ισοζυγισμένο δένδρο με μεγάλο fan out στους εσωτερικούς κόμβους. Θεωρείται η βέλτιστη δομή για δυναμική δεικτοδότηση με χρήση δευτερεύουσας μνήμης και για σύνολα δεδομένων που αναγνωρίζονται από μονοδιάστατα κλειδιά.

Για περισσότερες πληροφορίες για το b-tree και για μια περιγραφή της έννοιας δομή δεικτοδότησης που εδώ θεωρείται προαπαιτούμενη γνώση μπορεί κανείς να διαβάσει το [Cromer 79] ενώ για το kd tree υπάρχουν πληροφορίες στην ενότητα 2.4

2.3 Υλοποιήσεις και χρήση

Το kdb tree έχει αποτελέσει πεδίο έρευνας σε πολλά papers. Παραμένει σε μεγάλο βαθμό μια θεωρητική δομή. Έχουν κατασκευαστεί υλοποιήσεις kdb trees για την διενέργεια δοκιμών και μετρήσεων απόδοσης. Σε αυτές είναι σε λειτουργία μόνο ένα μέρος των δυνατοτήτων της δομής καθώς η υλοποίηση μερικών διαδικασιών όμως όπως το delete (διαγραφή) είναι δύσκολη και υπολογιστικά μάλλον μη αποδοτική.

Συνάμα δεν έχουνε εξασφαλιστεί σε θεωρητικό ή σε επίπεδο υλοποίησης ιδιότητες όπως η αντοχή σε σφάλματα και το concurrency (ταυτοχρονισμός). Το δένδρο προτάθηκε με την ελπίδα ότι θα λειτουργεί κοντά στα πρότυπα αποδοτικότητας του b-tree αλλά αυτό δεν έχει αποδειχθεί θεωρητικά και στην πράξη δεν εγγυημένο, αν και είναι πιθανόν να ισχύει με βάση τις ως τώρα δοκιμές [Robinson 81].

Προβλήματα στην απόδοση προκαλούν το ότι η διάσπαση κόμβων δεν είναι πάντα τοπική (ένας κόμβος που διασπάται μπορεί να προκαλέσει διάσπαση στους απογόνους του κάτι που δεν συμβαίνει στο B-tree) και το ότι δεν υπάρχει κάποια εγγύηση για το utilization (ποσοστό χρησιμοποίησης) των κόμβων, η οποία στο b-tree είναι στο 50%.

Λόγω του ότι δεν παρέχει εγγυήσεις απόδοσης και του ότι δεν έχει υλοποιηθεί πλήρως το δένδρο αυτό δεν χρησιμοποιείται σε DBMS, όπως και οι περισσότερες δομές πολυδιάστατης δεικτοδότησης που έχουν προταθεί [Gaede-Gunther 1998]

2.4 KD Tree

Το kd tree είναι δενδρική δομή που οργανώνει την πρόσβαση σε ένα πλήθος σημείων του ν-διάστατου χώρου έτσι ώστε αυτή να γίνεται σε logN χρόνο όπου N το πλήθος των σημείων. Μπορεί να αποτελέσει μια πολυδιάστατη εκδοχή δυαδικού δένδρου αλλά συνήθως χρησιμοποιείται σαν στατική δομή δενδρικής δυαδικής διαμέρισης του χώρου (Binary Space Partitioning Structure)

Στο kd tree γίνεται οργάνωση των σημείων ενός συνόλου μέσω γεωμετρικής διαμέρισης του χώρου, ώστε να αποφευχθεί η ακολουθιακή αναζήτηση στο σύνολο. Όπως στο δυαδικό δένδρο σε κάθε κόμβο του kd tree αντιστοιχεί και ένα εισηγμένο στοιχείο. Στο kd tree αυτό δεν είναι μια τιμή (μονοδιάστατα δεδομένα) αλλά ένα σημείο του ν διάστατου χώρου

Κάθε κόμβος αντιστοιχεί σε ένα σημείο κι αντίστροφα. Για διευκόλυνση το κάθε σημείο Χ θα συμβολίζεται με μορφή Κ.Χ όπου Κ είναι ο κόμβος όπου βρίσκεται το Χ στο δένδρο.

Το Χ είναι σημείο του Δ-διάστατου χώρου και η τιμή της συντεταγμένης του στην διάσταση δ για κάθε 0<= δ < Δ θα συμβολίζεται εδώ με Κ.Χ[δ].

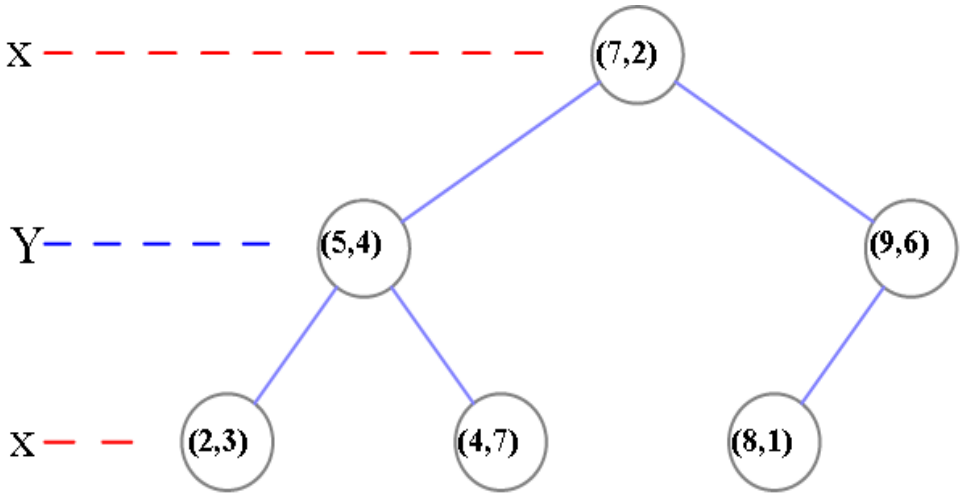
Στο kd tree σε κάθε εσωτερικό κόμβο (node) Κ αντιστοιχεί και μια διάσταση από τις Δ του διαμεριζομένου χώρου, θα συμβολίζεται εδώ με Κ.εδ.

Κάθε εσωτερικός κόμβος έχει ακριβώς δύο υιούς (children), οι οποίοι ονομάζονται δεξιός και αριστερός κόμβος, όπως γίνεται και στο δυαδικό δένδρο.

Δεξιά κάθε κόμβου Κ (πιο αυστηρά ο όρος δεξιά του κόμβου Κ αναφέρεται στο υποδένδρο με ρίζα τον δεξιό υιό του Κ) ισχύει ότι για κάθε σημείο Λ.Υ θα πρέπει Λ.Υ[Κ.εδ]>=Κ.Χ[Κ.εδ]. Συμμετρικά στο αριστερό υποδένδρο του Κ τα σημεία Ι.Ζ θα έχουν Ι.Ζ[Κ.εδ]<Κ.Χ[Κ.εδ]

Γεωμετρικά ο Κ χωρίζει τον ν διάστατο χώρο σε δύο τμήματα με βάση ένα ν-1 διάστατο υπερεπίπεδο ΥΕ κάθετο στον βασικό άξονα της Κ.εδ με το Κ.Χ να ανήκει στο ΥΕ. Τα σημεία δεξιά του υπερεπιπέδου πηγαίνουν στο δεξί υποδένδρο και τα αριστερά στο αριστερό

Όπως στο δυαδικό δένδρο σε κάθε κόμβο πλην των φύλλων λαμβάνεται μια απόφαση μέσω μιας σύγκρισης. Στο δυαδικό δένδρο η σύγκριση είναι με βάση την τιμή του Κ.Χ στον μονοδιάστατο άξονα ενώ στο Kd-tree η σύγκριση γίνεται με την τιμή του Κ.Χ[Κ.εδ]



(Στην εικόνα το kd tree σαν συγκρίνει σε διαφορετική διάσταση ανάλογα με το ύψος του κόμβου, πηγή εικόνας Wikipedia)

Δεν γίνεται σύγκριση πάντα στην ίδια διάσταση γιατί αυτό οδηγεί σε προβλήματα που περιγράφονται στην ενότητα 2.4. Αν σε κάθε κόμβο επιλέγονταν η ίδια διάσταση δ, το kd tree θα εκφυλιζόταν σε δυαδικό δένδρο που συγκρίνει μόνο τις τιμές X[δ] των σημείων, και αγνοεί τις άλλες διαστάσεις.

2.4.1 Τακτική διαμέρισης του χώρου

Η διαμέριση του χώρου μέσω υπερεπιπέδων (hyperplanes) γίνεται ώστε δύο σημεία κοντά σε απόσταση να τοποθετηθούν μαζί σε χαμηλού ύψους υποδένδρο. Τότε οι διαδρομές από την ρίζα προς αυτά θα είναι σε μεγάλο βαθμό κοινές κάτι που κάνει πιο αποδοτικούς τους αλγορίθμους εύρεσης όμοιων σημείων (similarity queries). Επειδή η απόσταση στον ν διάστατο χώρο είναι μέτρο όλων των συντεταγμένων (πχ όλες οι διαστάσεις συμμετέχουν στην ευκλείδεια απόσταση) πρέπει στην διαμέριση του kd tree να μην γίνεται χρήση υπερεπιπέδων κάθετων μόνο σε ένα υποσύνολο των αξόνων των διαστάσεων. Το ζήτημα αυτό αναφέρεται με τον όρο k cubicality

και εξετάζεται και στις ενότητες 3 και 4.2.1.2.

Ο συνηθέστερος τρόπος για να επιτευχθεί αυτό είναι να αλλάζουν με κυκλικό pattern (κυκλική εναλλαγή) οι διαστάσεις που επιλέγονται (Κ.εδ) ανάλογα με το ύψος του Κ στο δένδρο.

Η κυκλική εναλλαγή δεν είναι αποδοτική όταν κάποια διάσταση δ μπορεί δέχεται πιο πολλά queries από τις υπόλοιπες. Τότε ωφελεί να γίνουν επιπλέον διαχωρισμοί στην δ ώστε να περιορίζεται γρηγορότερα η αναζήτηση **(1)**.

Μπορεί κανείς να θεωρήσει το (1) ως αλλαγή του μέτρου απόστασης με βάση το αναμενόμενο εύρος των αναζητήσεων σε κάθε διάσταση πχ αν η y αγνοείται στα περισσότερα range queries (αναζητήσεις εύρους βλ. ενότητα 3) τότε το αναμενόμενο εύρος της είναι όλος ο άξονας y, αν η χ έχει μικρά όρια επιλογής σε όλα τα range queries το αναμενόμενο εύρος της είναι μικρό. (βλ και ενότητα 3.3 - selectivity).

2.4.2 Το KD Tree σαν στατική ή δυναμική δομή

Αν το kd tree χρησιμοποιηθεί σαν δυναμική δομή, (που επιτρέπει την εισαγωγή και διαγραφή εγγραφών μετά την αρχική κατασκευή και ανάμεσα στην εκτέλεση αναζητήσεων) οι εισαγωγές και διαγραφές θα το κάνουν μη ισοζυγισμένο.

Κατά την εισαγωγή ενός σημείου Ρ ο αλγόριθμος του kd tree ακολουθεί διαδρομή από την ρίζα προς τα φύλλα με βάση τις συντεταγμένες του Ρ. Στο τέλος τοποθετεί το Ρ ως υιό του φύλλου όπου καταλήγει.

Έστω ένα ισοζυγισμένο kd δένδρο με ρίζα R Αν ένα πλήθος από σημεία P1..Pn εισαχθούν κατά σειρά, έχοντας όλα Pi[R.εδ]> R[R.εδ]τότε θα τοποθετηθούν όλα στο δεξί υποδένδρο της R και έτσι το δένδρο θα πάψει να είναι ισοσταθμισμένο, αφού κανένα σημείο δεν εισάγεται στο αριστερό υποδένδρο.

Ένα δένδρο kd μπορεί να είναι ισοζυγισμένο εφόσον κατασκευαστεί στατικά μέσω ανάλυσης του συνόλου των σημείων. Η εισαγωγή σημείων “on line” το ένα μετά το άλλο δεν οδηγεί σε ισοζυγισμένο δένδρο.

Η στατική κατασκευή του kd δένδρου το γίνεται με έναν divide and conquer αλγόριθμο.. Σε κάθε βήμα επιλέγεται μια διάσταση δ, και ένα σημείο Χ με Χ[δ] ίση με την median τιμή του συνόλου Υ[δ] όπου το Υ ανήκει στο σύνολο των σημείων S. Το Χ τοποθετείται στην ρίζα του δένδρου και τα υπόλοιπα σημεία χωρίζονται σε δύο σύνολα SL και SR ανάλογα με το αν έχουν Υ[δ]>Χ[δ]. Στο SL εφαρμόζεται αναδρομικά η ίδια διαδικασία για να κατασκευαστεί το αριστερό υποδένδρο της ρίζας και στο SR για να κατασκευαστεί το δεξιό.

2.4.3 Βελτιώσεις του KDB Tree πάνω στο KD Tree

Το kdb tree μπορεί να θεωρηθεί ως τροποποίηση του b tree ώστε να λειτουργεί με ν διάστατα κλειδιά ή ως τροποποίηση του kd tree ώστε να κάνει αποδοτική χρήση της δευτερεύουσας μνήμης **(1)** και να παραμένει ισοσταθμισμένο κατά την δυναμική εισαγωγή σημείων**(2)**.
Όσον αφορά το (1) το μειονέκτημα του kd tree είναι το ότι δεν έχει μεγάλο fan out. Το fan out είναι το πλήθος ακμών που φεύγουν από έναν κόμβο του δένδρου, πόσους κόμβους υιούς μπορεί να έχει. Σε ένα δυαδικό δένδρο και στο kd tree ο αριθμός αυτός είναι 2. Έτσι οι ακμές που φεύγουν από κάθε κόμβο είναι λίγες κάτι που αυξάνει το ύψος του δένδρου. **(3)**

Όταν χρησιμοποιείται δευτερεύουσα μνήμη κάθε κόμβος του δένδρου αντιστοιχεί συνήθως σε ένα block, όπου το block είναι η μονάδα φυσικής οργάνωσης της δευτερεύουσας μνήμης (με την σημερινή τεχνολογία ισοδυναμεί σε 4kbyte περίπου). Για λόγους βελτιστοποίησης της λειτουργίας του hardware δεν επιτρέπεται η εγγραφή ή η ανάγνωση ποσότητας δεδομένων μικρότερης του ενός block.

Σε δομές που κάνουν χρήση δευτερεύουσας μνήμης, ένας δείκτης σε κόμβο του

δένδρου δεν δείχνει την διεύθυνση μιας περιοχής μνήμης στην RAM αλλά την διεύθυνση ενός block στην δευτερεύουσα μνήμη.

Σε ένα kd tree που χρησιμοποιεί δευτερεύουσα μνήμη αυξάνονται λόγω του (3) οι αναγνώσεις blocks κατά την διαδρομή από την ρίζα προς τα φύλα.

Λόγω της τεχνολογίας του hardware, η εγγραφή και η ανάγνωση blocks στον δίσκο είναι ιδιαίτερα χρονοβόρα, σε σημείο που συνήθως να ξεπερνάει σε χρονική διάρκεια την επεξεργασία του κόμβου από τον εκάστοτε αλγόριθμο (συγκρίσεις κλειδιών, διασπάσεις κτλ). Με άλλα λόγια το κόστος I/O κυριαρχεί του κόστους επεξεργασίας κόμβων.**(4)**

Για αυτό μια δομή κατάλληλη για χρήση δευτερεύουσας μνήμης θα πρέπει να προσπαθεί να ελαχιστοποιεί το πλήθος εργασιών I/O ακόμα και αν αυτό σημαίνει την αύξηση του χρόνου επεξεργασίας των κόμβων.

Στην αναζήτηση σε ένα binary tree (δυαδικό δένδρο) ο κάθε κόμβος απαιτεί μόνο μια εντολή σύγκρισης κατά την επεξεργασία του. Με αυτήν την σύγκριση λαμβάνεται η απόφαση σε ποιο υποδένδρο θα συνεχιστεί η αναζήτηση.

Στο b-tree απαιτείται μεγαλύτερος αριθμός συγκρίσεων, καθώς στο εσωτερικό του κόμβου δεν υπάρχει 1 τιμή και 2 δείκτες, αλλά d τιμές και d+1 δείκτες. Μια σύγκριση δεν αρκεί και θα πρέπει να γίνει κάποια σειριακή ή δυαδική αναζήτηση στο σύνολο των d τιμών **(5)** .

Το binary tree για τον ίδιο αριθμό σημείων έχει μεγαλύτερο ύψος από το b-tree λόγω του χαμηλότερου fan out (d+1 αντί για 2 υποδένδρα ανά κόμβο) . Έτσι αν και τα δύο χρησιμοποιούσαν με δευτερεύουσα μνήμη το b-tree θα χρειάζονταν λιγότερες ενέργειες I/O και παρόλο που καταναλώνει περισσότερο χρόνο σε συγκρίσεις λόγω του (5) θα εκτελούσε αναζητήσεις πιο γρήγορα λόγω του (4).

Το kdb tree προτάθηκε ως μια δομή βασισμένη στο kd tree αλλά κατάλληλη και για χρήση με δευτερεύουσα μνήμη. Οι βελτιώσεις στο kd tree γίνονται με βάση τον τρόπο λειτουργίας του b-tree ώστε να αυξηθεί το fan out. Έτσι το kdb tree είναι πιο πεπλατυσμένο από το ανάλογο kd tree και έχει καλύτερες επιδόσεις όταν χρησιμοποιείται με σκληρούς δίσκους κτλ

(Πηγές κεφαλαίου 2 [Robinson 81], [Cromer 79], Wikipedia)

3. ΠΟΛΥΔΙΑΣΤΑΤΗ ΔΕΙΚΤΟΔΟΤΗΣΗ

3.1 Χρησιμότητα

Οι πολυδιάστατες δομές δεικτοδότησης είναι δομές που ευρετηριοποιούν σημεία κάποιου ν διάστατου χώρου έτσι ώστε να εκτελούνται αποδοτικά οι αναζητήσεις τύπου Range Query και Nearest Neighbour Query. Ο όρος αναφέρεται implicitly μόνο σε δομές που επιτρέπουν την χρήση δευτερεύουσας μνήμης.

Τα ν διάστατα σημεία που τοποθετούνται σε αυτές τις δομές μπορεί να είναι τα ίδια τα δεδομένα, ή να αποτελούν τα κλειδιά κάποιων εγγραφών. Κλειδιά με την έννοια ότι το ν διάστατο σημείο προσδιορίζει μονοσήμαντα την εγγραφή. Επειδή το κλειδί ανήκει σε κάποιον γεωμετρικό χώρο, οι γεωμετρικές ιδιότητες του σε αυτόν μπορούν να αντιστοιχίθούν και στην εγγραφή που προσδιορίζει.

Στον χώρο η απόσταση μεταξύ 2 διανυσμάτων δείχνει το πόσο όμοια είναι μεταξύ τους. Για χωρικά δεδομένα ο χώρος είναι ο τρισδιάστατος ή ο δισδιάστατος ευκλείδειος χώρος και η εγγύτητα και ομοιότητα των σημείων υπολογίζεται με το μέτρο της ευκλείδειας απόστασης μεταξύ τους.

Μη χωρικά δεδομένα αποκτούν γεωμετρικές ιδιότητες όπως η απόσταση αφού μετασχηματιστούν από το λεγόμενο object space των αρχικών εγγραφών, σε ευκλείδεια διανύσματα ενός ν διάστατου χώρου(feature space, feature vector) με έναν μετασχηματισμό που μετασχηματίζει όμοια αντικείμενα του object space σε πλησία στο χώρο διανύσματα στο feature space. Με αυτόν τον τρόπο αναζητήσεις ομοιότητας εγγραφών μπορούν να μετατραπούν σε αναζητήσεις πλησιέστερου γείτονα και αναζητήσεις range στον ευκλείδειο χώρο (οι δύο τύποι αναζητήσεων που είναι οι βασικοί στις πολυδιάστατες δομές δεικτοδότησης).

3.2 Αναζητήσεις ν διαστάσεων

Μια δομή που υποστηρίζει αναζήτηση εγγραφών με πολυδιάστατα κριτήρια στην ουσία είναι μια δομή βελτιστοποιημένη για κάποιον από τους 2 παρακάτω τύπους αναζητήσεων.

- Range Query
- Nearest neighbour Query

Και οι δύο αυτοί τύποι αναζητούν διανύσματα (κατ επέκταση εγγραφές) που πληρούν κάποια γεωμετρικά κριτήρια. Τα κριτήρια αυτά είναι τέτοια ώστε να επιλέγουν όμοιες εγγραφές (δηλαδή κοντινά στον χώρο διανύσματα του feature space). Έτσι οι αναζητήσεις αυτές λειτουργούν ως *similarity queries*, δηλαδή βρίσκουν εγγραφές που είναι όμοιες με κάποιο υπόδειγμα.

3.2.1 Range Queries

Στα range queries (αναζητήσεις εύρους) ορίζεται για κάθε διάσταση **δ** του ευκλείδειου χώρου και από ένα διάστημα πραγματικών αριθμών **Ιδ**. Τα σημεία που συμφωνούν με τα κριτήρια αναζήτησης θα πρέπει να έχουν συντεταγμένη στην διάσταση δ που να περιορίζεται σε αυτό (**X[δ] in Ιδ**). Κάθε **Ιδ** μπορεί να εκτείνεται ανάμεσα σε μια ελάχιστη και μέγιστη τιμή (πχ το διάστημα **[0,5)**) ή να εκτείνεται ως το άπειρο (πχ το **[0,+inf)**. Γεωμετρικά τα κριτήρια αναζήτησης προσδιορίζουν ένα ν-διάστατο παραλληλόγραμμο, του οποίου όμως οι ακμές μπορούν να έχουν και άπειρο μήκος. Αλγεβρικά προσδιορίζεται το σύνολο των σημείων που ανήκουν στο καρτεσιανό γινόμενο των **Ιδ.**

Υποκατηγορίες των range queries έχουν ειδικές μορφές στα **Ιδ**. Αν για κάθε διάσταση το διάστημα **Ιδ** εκφυλίζεται σε σημείο τότε το range query ονομάζεται point query. Αν για κάποια από τις διαστάσεις τεθεί Ιδ το **(-inf,inf)** τότε η αναζήτηση

ονομάζεται partial range query. Τα partial range queries επιτρέπουν να μην ληφθεί υπόψην μια διάσταση στα κριτήρια αναζήτησης (εξ ου και ο χαρακτηρισμός partial, δεν συμμετέχουν όλες οι διαστάσεις)

Το range query περιορίζει την αναζήτηση σε μια υποπεριοχή του ν διάστατου χώρου. Με αυτόν τον τρόπο για κάποιο σημείο P μπορεί να γίνει ένα similarity query επιλέγοντας αρκετά μικρά διαστήματα αναζήτησης Ιδ στην κάθε διάσταση έτσι ώστε μόνο σημεία σε ένα αρκετά μικρό ν διάστατο παραλληλόγραμμο που περιλαμβάνει το P να αναζητηθούν.

3.2.2 Nearest Neighbour Query

Στα nearest neighbour queries η αναζήτηση γίνεται με βάση ένα σημείο στον χώρο P και μια παράμετρο K όπου το K είναι φυσικός αριθμός. Αυτό που ζητείται είναι τα K πλησιέστερα στο P σημεία του συνόλου των δεδομένων. Η εγγύτητα καθορίζεται από το μέτρο απόστασης του χώρου π.χ. ευκλείδεια απόσταση, απόσταση Μανχάταν. Επομένως Τα K πλησιέστερα στο P σημεία είναι τα πρώτα K σημεία σε μια ταξινόμηση του συνόλου των σημείων σε αύξουσα σειρά απόστασης από το P.

Για τα nearest neighbour queries δεν είναι αναγκαίο το feature space να είναι ευκλείδειος χώρος και αρκεί να έχει οριστεί κάποιο μέτρο απόστασης σημείων (metric space). Υπάρχουνε δομές ειδικές, για δεδομένα που μπορούν να τοποθετηθούν σε metric space, και συνήθως σε μεγάλο αριθμό διαστάσεων είναι πιο αποτελεσματικές από τις αντίστοιχες για ευκλείδειο χώρο. [Πηγή M-Tree Project http://www-db.deis.unibo.it/Mtree/, 16 February 2010]

3.3 Σύγκριση με ν ανεξάρτητες μονοδιάστατες δομές

Όταν η πλειονότητα των αναζητήσεων είναι τύπου range ή nearest neighbour, μια πολυδιάστατη δομή δεικτοδότησης έχει καλύτερη αποδοτικότητα από την χρήση n διαφορετικών ανεξάρτητων μονοδιάστατων δομών (δηλαδή μια για κάθε κλειδί του feature ή χωρικού διανύσματος).

Ένας λόγος είναι το ότι για ν δομές χρειάζονται και ν ανανεώσεις του ευρετηρίου σε κάθε εισαγωγή και διαγραφή. Αυτή η εξάρτηση από το πλήθος των διαστάσεων αποτρέπει την χρήση ν μονοδιάστατων δομών για δεδομένα πολλών διαστάσεων (πρακτικά περισσότερες από 5 διαστάσεις συνήθως είναι ήδη μεγάλος αριθμός [Gaede-Gunther 1998])

Ένας άλλος λόγος είναι το ότι τα χαρακτηριστικά των αναζητήσεων range και nearest neighbour είναι τέτοια ώστε να μην μπορούν να εκμεταλλευτούν τις πληροφορίες των ν δομών για την εκπλήρωση τους.

Κατ' αρχάς στα partial range queries κάποια από τις ν δομές δεικτοδότησης θα είναι άχρηστη (καθώς επιστρέφοντας τα σημεία που ικανοποιούν την συνθήκη Χ[δ] ανήκει Ιδ=(-inf+inf) θα επιστρέψει αναγκαστικά όλο το σύνολο των σημείων**(1)**). Αυτό σημαίνει ότι θα πρέπει να υπάρχει κάποιας μορφής περαιτέρω οργάνωση της χρήση των ν δομών δεικτοδότησης, γιατί η απλή παράλληλη χρήση τους με συνδυασμό/απόρριψη των αποτελεσμάτων μπορεί να οδηγήσει σε άσκοπη χρήση I/O λόγω του (1).

Ακόμα και σε μια διάσταση που το διάστημα του κριτηρίου αναζήτησης Ιδ είναι μικρό και όχι το (-inf,+inf), είναι δυνατόν να έχει συγκεντρωθεί σε αυτό όλο σχεδόν το πλήθος των σημείων του συνόλου. Μια διάσταση δ' χαρακτηρίζεται selective σε κάποιο Query q, όταν το Ιδ' είναι τέτοιο που να περιορίζει σε μεγάλο βαθμό το σύνολο έγκυρων σημείων. Αν προηγουμένως το index για μια selective διάσταση δs έχει ήδη δώσει κάποιες εγγραφές, τότε χρησιμοποιώντας ακολούθως το index της

non selective δns θα γίνει μια περιττή νέα ανάγνωση τους. Αν μόνο μια διάσταση είναι selective για το συγκεκριμένο Query, το ποσοστό αχρείαστων I/O είναι ακόμα πιο μεγάλο.

Δεν είναι εκ των προτέρω γνωστό ποιες διαστάσεις είναι selective σε ποιες αναζητήσεις. Αυτό επιτυγχάνεται μόνο καθώς χρησιμοποιείται κάθε index (καθώς κατεβαίνει η διαδικασία αναζήτησης προς τα φύλλα). Έτσι είναι αναπόφευκτο το να χρησιμοποιηθούν οι ν δομές δεικτοδότησης με κάποιο "παράλληλο" τρόπο γιατί αυτό συνεπάγεται την εκτέλεση περιττών λειτουργιών I/O οι οποίες διαβάζουν πολλές φορές τα ίδια δεδομένα. Με βάση τα παραπάνω η χρήση ν δομών είναι μη αποδοτική. Οι ν δομές θα χρησίμευαν μόνο αν όλα τα queries περιλάμβαναν μόνο μια διάσταση στα κριτήρια αναζητήσεις.

Όταν αυτό δεν ισχύει το πλεονέκτημα σε απόδοση έχουν οι πολυδιάστατες δομές που όμως αντίστροφα υστερούν σε μονοδιάστατες αναζητήσεις (αυτό το trade off θεωρείται αρκετά σημαντικό ώστε να εμποδίζει την γενική χρήση των πολυδιάστατων δομών στα DBMS [Gaede Gunther 1998])

Στις πολυδιάστατες δομές, όπως και στις ν μονοδιάστατες, γίνεται διαμέριση του χώρου. Σε αυτήν όμως όλες συμμετέχουν όλες οι διαστάσεις μαζί αντί η κάθε μια ξεχωριστά να διαμερίζει τον χώρο στο δικό της μονοδιάστατο indexing structure. Με αυτόν τον τρόπο η πολυδιάστατη δομή δεν αχρηστεύεται επειδή μια διάσταση τυχαίνει να είναι non selective. Αυτή η λογική διέπει και την δομή του kd tree (ενότητα 2.3.1)

3.4 Κατηγορίες δομών πολυδιάστατης δεικτοδότησης

Για τους παραπάνω λόγους αναπτύσσονται δομές πιο κατάλληλες για range και nearest neigbour queries σε point δεδομένα. Αυτές ονομάζονται δομές πολυδιάστατης δεικτοδότησης. Κατατάσσονται σε 3 κύριες κατηγορίες

- Δομές με one dimensional mapping
-Grid files
-Ιεραρχικές δομές

Σε μεγάλο βαθμό η λειτουργία και η απόδοση των δομών αυτών (και γενικά όσων προσανατολίζονται για χρήση I/O) επηρεάζεται από τον τρόπο διαχείρισης της κεντρικής μνήμης από το λειτουργικό σύστημα, και από το αν γίνεται χρήση εικονικής μνήμης (virtual memory). [Gaede and Gunther 98]. Τότε εφόσον διατηρούνται περισσότερες σελίδες στην ΚΜ τα κόστη ενεργειών I/O μειώνονται σε κάποιο βαθμό (αν και απαιτήσεις για fault tolerance μπορεί να καταστούν αναγκαίες τις εγγραφές ακόμα και σελίδων που δεν έχουν βγει από την ΚΜ). Η πολιτική εναλλαγής σελίδων στο memory management που ταιριάζει σε τέτοιες δομές είναι η LRU καθώς σε depth first traversals θα τείνει να κρατάει στην μνήμη τους κόμβους προγόνους που χρησιμοποιούνται συχνά λόγω backtracking. Αν γίνει χρήση της FIFO τα αποτελέσματα δεν θα είναι καλά γιατί οι κόμβοι πρόγονοι στους οποίους επιστρέφει το DFT δεν βρίσκονται στην μνήμη, αντίθετα θα στέκονται σε αυτήν κόμβοι που χρησιμοποιήθηκαν πρόσφατα αλλά που όμως το search στο υποδένδρο τους έχει ολοκληρωθεί.

3.4.1 Ιεραρχικές δομές

Στην πρώτη κατηγορία, στην οποία ανήκει και το kdb tree, ανήκουν δομές οι οποίες βασίζονται σε modifications του btree. Το btree δεν μπορεί να δεικτοδοτήσει παρά μονοδιάστατα δεδομένα. Επομένως πρέπει να τροποποιηθεί η δομή για να δέχεται πολυδιάστατα .

3.4.1.1 Διαίρεση του συνόλου δεδομένων και του χώρου στο b-tree

Ο τρόπος λειτουργίας του b tree στηρίζεται στο ότι τα μονοδιάστατα δεδομένα έχουν την καλά ορισμένη διάταξη(ordering) του R (γραμμής των πραγματικών αριθμών) και ότι σε αυτήν την διάταξη τα κοντά στην διάταξη σημεία βρίσκονται και κοντά σε γεωμετρική απόσταση.

Ένας κόμβος του b tree μπορεί να θεωρηθεί ως ένα σύνολο από κλειδιά K και δείκτες δκ .Κάθε κλειδί αντιστοιχεί σε μία τιμή του R και κάθε δείκτης δκ βρίσκεται "ανάμεσα"σε δύο κλειδιά K1 και K2 και δείχνει προς το υποδένδρο του b tree όπου βρίσκονται όλα τα εισαχθέντα σημεία που ακολουθούν σε διάταξη το K1 ενώ προηγούνται από το K2. Διασχίζοντας το δένδρο με διαδοχικές επιλογές δείκτη δκi σε κάθε κόμβο, τα K1i K2i πλησιάζουν σε απόσταση μεταξύ τους και το ενδιάμεσο διάστημα μικραίνει. Έτσι ένα σημείο P και τα γειτονικά του ως ένα όριο μέγιστης απόστασης σημεία θα βρίσκονται όλα σε ένα υποδένδρο ΥΔ του b-tree με σχετικά χαμηλό ύψος, ενώ καθώς το όριο απόστασης αυξάνει θα αυξάνει και το ύψος του ΥΔ.

Αυτή η μορφή οργάνωσης δεν γίνεται να εφαρμοστεί ως έχει σε πολυδιάστατα δεδομένα γιατί αυτά δεν έχουν κάποια διάταξη ανάλογη του R. Κάθε πιθανή διάταξη των σημείων του ν διάστατου χώρου αντιστοιχεί σε μία συνάρτηση από αυτόν προς τον μονοδιάστατο. Όμως για καμία από αυτές τις συναρτήσεις δεν είναι εγγυημένο ότι σημεία κοντά σε απόσταση στον ν διάστατο χώρο θα σταλούν και κοντά σε διάταξη στον μονοδιάστατο**(1)**. Για παράδειγμα ταξινομώντας ένα σύνολο από δισδιάστατα σημεία σαν πλειάδες πρώτα ως προς χ και μετά ως προς y, το σημείο (10,1) πιθανόν να είναι πολύ μακριά σε σειρά από το (11,1) αν και κοντά στον χώρο σε αυτό, καθώς θα μεσολαβούν ανάμεσα τους όλα τα σημεία της μορφής (10,υ>1). Στην ενότητα 3.4.3 παρουσιάζονται συναρτήσεις για τις οποίες η πιθανότητα να συμβεί το (1) είναι κατά το δυνατόν η ελάχιστη.

3.4.1.2 Τροποποίηση του b tree στις ν διαστάσεις

Σε πιο αφηρημένη θεώρηση κάθε κόμβος Κ του b-tree αντιπροσωπεύει ένα διάστημα του μονοδιάστατου χώρου, δηλαδή ένα μονοδιάστατο παραλληλόγραμμο R.. Αγνοώντας προς το παρών την διάταξη και τα κλειδιά Κ1 και Κ2, κάθε δείκτης στον Κ οδηγεί σε έναν κόμβο Κ2 ο οποίος αντιπροσωπεύει ένα μονοδιάστατο παραλληλόγραμμο R2 που βρίσκεται εσωτερικά του R στον χώρο. Καθώς κατεβαίνει κανείς το δένδρο τα παραλληλόγραμμα γίνονται αναγκαστικά (λόγω της απαίτησης R2 in R1) όλο και πιο περιορισμένα σε έκταση με αποτέλεσμα τα σημεία κοντά σε απόσταση στον χώρο να ανήκουν όλα σε υποδένδρο χαμηλού σχετικά ύψους.

Με βάση αυτήν την παρατήρηση κατασκευάζονται οι ν διάστατες παραλλαγές του b tree. Αυτό που αλλάζει είναι η εσωτερική δομή του κάθε κόμβου, με την αντικατάσταση των δεικτών Κi της μονοδιάστατης διάταξης, με δείκτες που αντιπροσωπεύουν εσωτερικά παραλληλόγραμμα όπως το R2.

Πιο αφηρημένα κάθε κόμβος διαιρείται σε υποπεριοχές και προκύπτει μια ιεραρχική οργάνωση. Οι υποπεριοχές μπορεί να έχουν και άλλα σχήματα αντί για παραλληλόγραμμα, όπως για παράδειγμα σφαίρες.

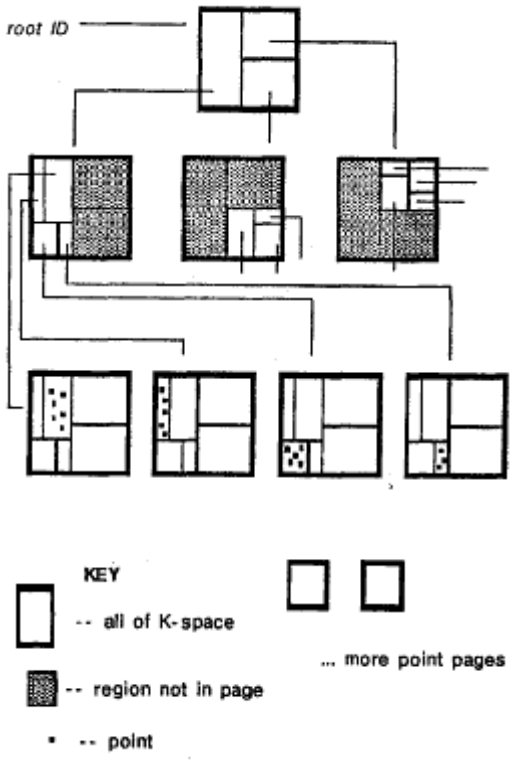
Διαμέριση συνόλου σημείων R αποτελούν μια ομάδα από άλλα σύνολα σημείων Ri, για την οποία η ένωση των Ri είναι το R και η τομή κάθε ζεύγους Ri,Rj είναι το κενό σύνολο (δηλαδή δεν υπάρχει αλληλοκάλυψη, overlap,στον χώρο).

Οι υποπεριοχές Ri της περιοχής R κάθε κόμβου μπορούν να αποτελούν διαμέριση της R ή να υπάρχει overlap. (σε κάθε περίπτωση οι Ri είναι υποσύνολα και γεωμετρικά εντός της R)

Δομές που λειτουργούν με τέτοιο τρόπο είναι πχ το kdb tree και το hb tree (με

διαμέριση κάθε περιοχής) αλλά και δομές όπως τα R-trees (με overlap)

(Στην εικόνα Ιεραρχική διαμέριση του χώρου στο kdb-tree, πηγή εικόνας [Robinson 81])



3.4.2 Grid Files

Τα grid files είναι δομές δεικτοδότησης που δεν είναι ιεραρχικές και δεν στηρίζονται στα b trees. Η λογική τους είναι η διαίρεση του ν διάστατου χώρου μέσω κάθετων στους βασικούς άξονες υπερεπιπέδων. Αυτά περιγράφονται με εξισώσεις τύπου χ=σταθερά, υ=σταθερά κτλ.

Καθώς τα υπερεπίπεδα τέμνονται σχηματίζονται ανάμεσα τους κελιά (cells) τα οποία είναι ν διάστατα παραλληλόγραμμα που διαμερίζουν τον χώρο. Στην κεντρική

μνήμη αποθηκεύεται για το κάθε cell και από ένας δείκτης σε ένα block δευτερεύουσας μνήμης (το οποίο ονομάζεται bucket και μπορεί να αντιστοιχεί σε περισσότερα του ενός cells αν αυτά είναι αρκετά αραιά σε εισηχθέντα σημεία). Η δομή δεικτών στην ΚΜ ονομάζεται grid . Grid ονομάζεται και η διαμέριση του χώρου σε cells σαν μαθηματική δομή.

Στο bucket υπάρχουν πληροφορίες για τα σημεία-εγγραφές που βρίσκονται εντός του cell. Τα grid files είναι βασισμένα σε μια πολυδιάστατη εκδοχή του hashing. Το block/bucket κάποιου σημείου εντοπίζεται στο grid και στη συνέχεια μεταφέρεται στην κύρια μνήμη με ένα μόνο βήμα I/O.

Αν σε ένα cell C γίνει εισαγωγή τόσων σημείων ώστε το C να ξεπερνάει σε μέγεθος το bucket/block, τότε προστίθεται στο C ένα ακόμα υπερεπίπεδο διαχωρισμού ΥΔ χωρίζοντας το C, αλλά και όλα τα άλλα cells που τέμνονται από το ΥΔ στα δύο. Αυτό το το split των cells δεν γίνεται με ιεραρχικό και τοπικό τρόπο (τοπικό με την έννοια ότι ένα split στην περιοχή R να μην μπορεί να προκαλέσει split της R2 όταν η R2 βρίσκεται μακριά από την R στον χώρο) γιατί το ΥΔ μπορεί να τέμνει και απομακρυσμένα στον χώρο cells. Σε ομοιόμορφης κατανομής δεδομένα αυτό δεν είναι πρόβλημα γιατί κάθε cell έχει πάνω κάτω το ίδιο πλήθος σημείων. Όταν το cell C1 είναι έτοιμο για διάσπαση, κάποιο άλλο cell C2 θα είναι ούτως ή άλλως πολύ πιθανόν επίσης έτοιμο για διάσπαση και αυτό. Σε μη ομοιόμορφα δεδομένα όμως τα επιπλέον μη τοπικά splits γίνονται σε cells τα οποία μπορεί να έχουν πολύ λίγα σημεία. Σε τέτοιες περιπτώσεις μια πιο ιεραρχική δομή όπως αυτές που βασίζονται στο b tree έχει καλύτερα αποτελέσματα.

Το βασικό πρόβλημα των grid like δομών όμως είναι το ότι η μη τοπικότητα των splits οδηγεί στην γρήγορη και υπεργραμμική [Gaede Gunther 98] αύξηση του μεγέθους του grid με αποτέλεσμα να μην μπορεί να τοποθετηθεί ολόκληρο στην κύρια μνήμη και να μην λειτουργεί η δομή. Αυτό λύνεται εν μέρει με grid files πολλαπλών επιπέδων.

Πάντως ο χώρος σε δευτερεύουσα μνήμη και μνήμη RAM είναι φτηνός, ενώ αυτό που είναι ακριβό είναι το πλήθος ενεργειών I/O.. Επειδή η χωρητικότητα των μονάδων μνήμης (δευτερεύουσας και μη) αυξάνεται, ενώ η ταχύτητα πρόσβασης σε αυτές παραμένει σχετικά σταθερή, οι δομές τύπου grid πιθανόν να γίνουν πιο δημοφιλείς στο μέλλον καθώς έχουν σταθερό κόστος 1 ενέργεια I/O ανά ανάγνωση.

3.4.3 One dimensional mapping

3.4.3.1 Διάταξη και απόσταση σε ν διαστάσεις

Στην κατηγορία πολυδιάστατων δομών δεικτοδότησης όπου χρησιμοποιείται one dimensional mapping (αντιστοίχηση στην μία διάσταση) τα πολυδιάστατα δεδομένα μετασχηματίζονται με μια συνάρτηση R^n -> R σε δεδομένα μιας διάστασης ώστε να μπορούν να τοποθετηθούν σε μια δομή όπως το B-tree.

Όπως αναφέρθηκε παραπάνω δεν υπάρχει κάποια διάταξη των σημείων ενός ν-διάστατου χώρου τέτοια που να διατηρεί την τοπικότητα. Κάποια από τα σημεία που βρίσκονται κοντά σε απόσταση στο R^n θα βρεθούν αναγκαστικά μακριά στο R όταν διαταχθούν (h διάταξη ισοδυναμεί με την δημιουργία μιας συνάρτησης από το R^n στο R).

3.4.3.2 Space Filling Curves

Υπάρχουν πάντως συναρτήσεις που καθιστούν την πιθανότητα να γίνει κάτι τέτοιο πολύ μικρή, δηλαδή για την πλειονότητα των σημείων να μην υπάρχουν κάποια σημεία κοντά τους στο χώρο και μακριά τους στην διάταξη. Οι συναρτήσεις αυτές προκύπτουν με την κατασκευή μιας καμπύλης στον ν-διάστατο χώρο η οποία να περνάει από όλα σχεδόν τα σημεία του (space filling curve).

Η καμπύλη κατασκευάζεται συνήθως με μια ακολουθιακή διαδικασία παρόμοια με αυτή των fractals. Παραμετροποιώντας την καμπύλη ως προς κάποια παράμετρο τ απεικονίζεται κάθε σημείο P του ν διάστατου χώρου στην τιμή της τ όταν η καμπύλη περνάει από το P. Επειδή η καμπύλη είναι fractal like διατηρεί την τοπικότητα σε μεγάλο βαθμό και για τα περισσότερα σημεία στέλνει την γειτονιά τους στο R^n σε κοντινές τιμές της τ **(1)**. Ο υπολογισμός του τ για κάποιο σημείο καθώς και ο υπολογισμός του διαστήματος του τ που αντιστοιχεί σε ένα range του R^n μπορεί να είναι δύσκολος και χρονοβόρος **(2)**, ενώ για τα λίγα "προβληματικά" σημεία όπου δεν ισχύει το (1) μπορεί να χρειαστεί να αναζητηθούν πολυάριθμα διαστήματα τιμών της R (πολλά I/O των blocks του b-tree δηλαδή)**(3)** .

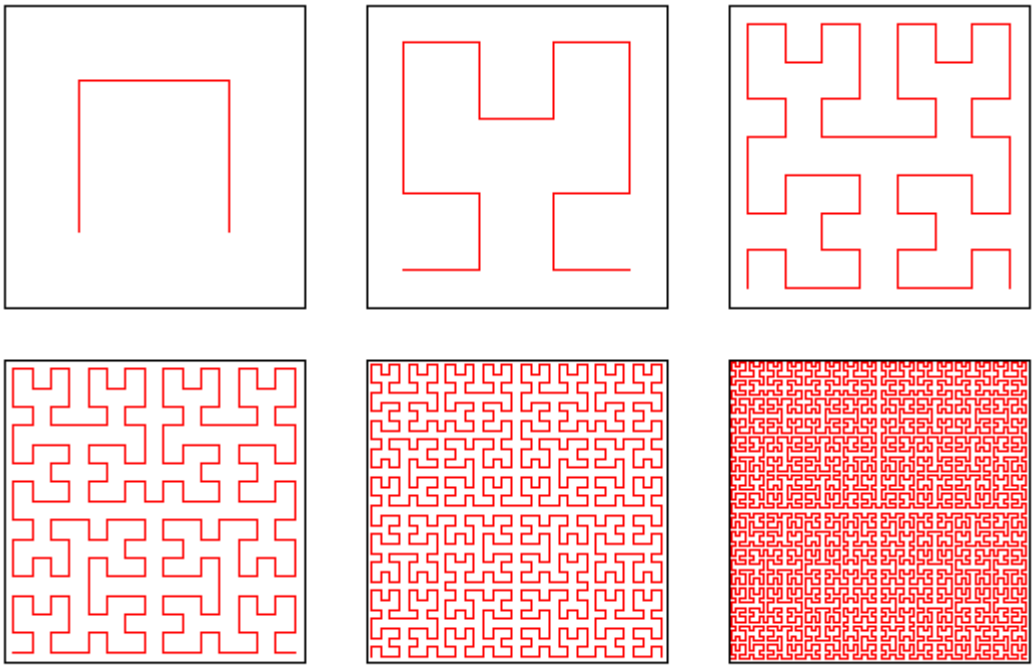
Η υπόθεση αυτής της μεθόδου είναι ότι ο χαμένος χρόνος στο (2) είναι μικρός σε σχέση με το κέρδος από το εγγυημένο 50-50 utilization του b-tree με τα μετασχηματισμένα δεδομένα (το υψηλό utilization συνεπάγεται μικρότερο ύψος δένδρου και λιγότερες ενέργειες I/O). Επίσης γίνεται η παραδοχή ότι τα σημεία όπου υπάρχει το πρόβλημα (3) είναι πολύ λίγα σε πλήθος, και έτσι κατά μέσο όρο σε όλο το σύνολο των σημείων, η δομή παραμένει γρήγορη.

Το σίγουρο πλεονέκτημα που έχουν αυτές οι μέθοδοι είναι το ότι επιτρέπουν την χρήση δοκιμασμένων δομών μονοδιάστατης δεικτοδότησης όπως το b-tree. Για αυτές προβλήματα συγχρονισμού και fault tolerance (αντοχή σε λάθη) έχουνε λυθεί και υπάρχουνε βελτιστοποιημένες υλοποιήσεις σε αντίθεση με τις πολυδιάστατες δομές.. Αυτό το θέμα αποτελεί εμπόδιο για την χρήση πολυδιάστατων δομών σε dbms καθώς ο συγχρονισμός και το recoverability (ανακτησιμότητα) είναι σημαντικά για μια multi user και προορισμένη για production environment βάση δεδομένων.

Οπότε οι πολυδιάστατες δομές με χρήση space filling curves είναι οι μόνες που έχουνε υλοποιηθεί σε dbms (πχ η Oracle υποστήριζε z curves). Πεδίο έρευνας αποτελεί η εξασφάλιση αυτών των ιδιοτήτων σε άλλες πολυδιάστατες δομές δεικτοδότησης. Οι καμπύλες που χρησιμοποιούνται συχνότερα ως space filling

curves είναι η z-curve και η hilbert curve.

(Στην εικόνα η ακολουθιακή κατασκευή της Hilbert Curve, πηγή εικόνας Wikipedia)



3.5 Κατηγοριοποίηση του KDB Tree

Το kdb tree όπως αναφέρθηκε πιο πάνω το kdb tree ανήκει στις δομές πολυδιάστατης δεικτοδότησης των οποίων ο τρόπος λειτουργίας βασίζεται σε αυτόν του b-tree. Τα χαρακτηριστικά του b-tree που διατηρούνται, πέρα από την ιεραρχική διαίρεση του χώρου από τους κόμβους, είναι, η ανάπτυξη από τα φύλλα, η εισαγωγή δεικτών στον κόμβο γονέα κατά την διάσπαση και η σταθερή απόσταση όλων των φύλλων από την ρίζα.

Στο kdb tree τα δεδομένα αποθηκεύονται μόνο στα φύλλα του δένδρου με τρόπο παρόμοιο με τον τρόπο αποθήκευσης του B+tree (παραλλαγής του B-tree). Το kdb tree δεν έχει ίδιες εγγυήσεις απόδοσης με το b-tree. Δεν υπάρχει εγγύηση για το utilization τον κόμβων, που στο B-tree είναι άνω του 50%. Επίσης το πλήθος ενεργειών I/O για τις βασικές λειτουργίες αλλάζει. Ένα insert(εισαγωγή) στο b-tree

θα χρειαστεί (Η το ύψος του δένδρου) Η αναγνώσεις δίσκου και το πολύ 2*Η εγγραφές δίσκου (ανάλογα με το πόσες διασπάσεις κόμβων θα γίνουν) .

Στο kdb tree μεγαλώνει το άνω όριο αναγνώσεων και εγγραφών κατά την εισαγωγή . Αυτό συμβαίνει γιατί η διάσπαση ενός κόμβου μπορεί θεωρητικά να επιφέρει και την διάσπαση όλων των απογόνων του. Δεν ακολουθείται δηλαδή μια "γραμμική" πορεία από τη ρίζα σε ένα φύλλο και πίσω.

(βασική πηγή του κεφαλαίου 3 [Gaede and Gunther 1998])

4. ΔΟΜΗ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ ΤΟΥ KDB TREE

4.1 Δομή

4.1.1 Διαφορές με το KD tree

Η κύρια διαφορά ανάμεσα στην δομή του kdb tree και αυτή του kd tree είναι ότι στο kdb tree το fan out έχει αυξηθεί. Το kd tree είναι δυαδικό. Κάθε εσωτερικός κόμβος είχε δύο το πολύ υποδένδρα. Με επιλογή υπερεπίπεδου διαχωρισμού κάθετου στον βασικό άξονα μιας διάστασης τα σημεία πηγαίνουν στο αριστερό η το δεξί υποδένδρο. (ενότητα 2.3)

Στο kdb tree ο κάθε κόμβος μπορεί να έχει περισσότερες μια επιλογές υπερεπιπέδου, και άρα περισσότερα των δύο υποδένδρα. Ο μέγιστος αριθμός είναι ανάλογος του μέγιστου αριθμού τιμών σε έναν κόμβο του b-tree.

Κάθε κόμβος αντιπροσωπεύει μια περιοχή του χώρου που διαμερίζεται από τις περιοχές που αντιπροσωπεύουν αντίστοιχα οι κόμβοι ρίζες των υποδένδρων του. Η ρίζα όλης της δομής αντιπροσωπεύει όλο το χώρο και οι υπόλοιποι κόμβοι τον χωρίζουν με ιεραρχικό τρόπο (3.4.1.1)

Όταν προστίθεται ένα νέο υπερεπίπεδο επιλογής στον κόμβο, έστω το ν-οστό, αυτό διαχωρίζει κάποια από τις υποπεριοχές που προέκυψαν από τα προηγούμενα ν-1 επίπεδα και όχι όλη την περιοχή του κόμβου.**(1)**

Έτσι ο χώρος τμηματοποιείται ιεραρχικά, Στην ουσία κάθε κόμβος (πλην των φύλλων όπου αποθηκεύονται οι εγγραφές) του kdb tree αντιστοιχεί σε ένα ολόκληρο τμήμα δένδρου τύπου kd tree (αντί για έναν κόμβο kd tree μόνο) το οποίο ορίζει την παραπάνω διαδικασία τμηματοποίησης.

Στην ρίζα του kd δένδρου αυτού βρίσκεται το πρώτο υπερεπίπεδο διαχωρισμού YE1. Οι δύο υιοί της ρίζας κόμβοι είναι, ο αριστερός το υπερεπίπεδο YE1A που χωρίζει την αριστερή πλευρά του διαχωρισμού μέσω YE1, και ο δεξιός το υπερεπίπεδο YE1Δ που χωρίζει την δεξιά. Η διαδικασία συνεχίζεται αναδρομικά στο δεξί και αριστερό υποδένδρο όπως προβλέπει το (1)

* * *

Το εσωτερικό του non leaf κόμβου του kdb tree μπορεί να έχει την μορφή ενός ενσωματωμένου kd tree με δείκτες σε θυγατρικούς κόμβους (τακτική που ακολουθείται με παρόμοιο τρόπο στο hb tree).

Στο paper του Robinson προτείνεται άλλος τρόπος υλοποίησης. Σε κάθε non leaf κόμβο K αποθηκεύεται ένας πίνακας από εγγραφές που αναπαριστούν ν διαστάσεων παραλληλόγραμμα (regions). Τα παραλληλόγραμμα είναι αυτά που προκύπτουν εφαρμόζοντας ιεραρχικά την διαίρεση με το kd tree του K (δηλαδή δεν αποθηκεύεται το ίδιο το kd tree αλλά μόνο οι υποπεριοχές/παραλληλόγραμμα στα οποία διαμερίζει την περιοχή του K).

Οι δύο προσεγγίσεις δεν επηρεάζουν το αποτέλεσμα της εφαρμογής των αλγορίθμων . Όμως στην πρώτη η διαμέριση του χώρου είναι precomputed (προϋπολογισμένη) ενώ στην άλλη υπολογίζεται ξανά κάθε φορά που επισκέπτεται ο αλγόριθμος τον κόμβο.

Με την διατήρηση regions αντί για kd trees αφαιρείται κάποιο επαναλαμβανόμενο υπολογιστικό κόστος από την χρήση του kd tree. Επίσης όταν χρησιμοποιείται σκέτο kd tree οι αλγόριθμοι των λειτουργιών search insert split και delete γίνονται πιο πολύπλοκοι γιατί στην ουσία εμπεριέχουν ξανά τα βήματα υπολογισμού των regions. (πχ στο collision detection)

Από την άλλη ο πίνακας των regions ξοδεύει περισσότερο χώρο από το αντίστοιχο kd δέντρο κάτι που οδηγεί σε χαμηλότερο utilization και κατά συνέπεια fan out.

Επειδή το επιπλέον υπολογιστικό κόστος για την χρήση των εσωτερικών kd trees είναι μικρότερο από το χρονικό κόστος I/O που επιφέρει η μείωση του fan out, είναι τελικά μάλλον προτιμότερο να χρησιμοποιηθούν εσωτερικά kd trees. Πιθανόν ο Robinson δεν τα χρησιμοποίησε για να κάνει πιο εύκολη την επεξήγηση της λειτουργίας της όλης δομής και την απόδειξη της ορθότητας των αλγορίθμων.

4.1.2 Points

Τα δεδομένα που εισάγονται σε ένα kdb tree θεωρείται ότι έχουν την μορφή

(key-1,key-2,...,key-i,...,key-n,record)

ή εν συντομία

(point, record)

Τα key-1,key-2...keyn αποτελούν το n διάστατο διάνυσμα που περιγράφει την εγγραφή. Το διάνυσμα αυτό ονομάζεται point. Το record είναι η ίδια η εγγραφή ή κάποιος δείκτης προς την θέση της στην δευτερεύουσα μνήμη.

(Σημείωση Στην εργασία αυτή δεν υλοποιήθηκε η αποθήκευση του record.. Ο τρόπος αποθήκευσης των records εξαρτάται από εξωτερικούς παράγοντες, σχετικούς με την μορφή που έχουν τα εκάστοτε δεδομένα του object space, τις λεπτομέρειες υλοποίησης των file systems ή dbms κτλ. Τέτοια ζητήματα θεωρήθηκαν εκτός του πεδίου του εργασίας. Το πως θα αποθηκευόταν τα records δεν επηρεάζει την συμπεριφορά μιας υλοποίησης του kdb tree ως προς την αποδοτικότητα σε πλήθος ενεργειών I/O των βασικών λειτουργιών

query/insert/delete.)

Οι τιμές key-i ανήκουν η κάθε μια σε ένα σύνολο domain-i το οποίο πρέπει να είναι διατεταγμένο (να έχει οριστεί διάταξη των στοιχείων του). Στην πιο απλή περίπτωση το κάθε domain-i είναι υποσύνολο του R οπότε ο χώρος στον οποίο βρίσκονται τα διανύσματα είναι υποσύνολο του R^n

Σε έναν υπολογιστή δεν γίνεται να αποθηκευτεί κάθε δυνατή τιμή του R λόγω της περιορισμένης ακρίβειας των αριθμών κινητής υποδιαστολής. Οπότε το κάθε domain-i αντιστοιχεί πιο συγκεκριμένα σε κάποιο πεπερασμένο υποσύνολο του R (πχ τα σύνολα κάθε επιτρεπτού float, double, ή int και τα υποσύνολα αυτών)

Ο τρόπος που θα αποθηκευτεί κάθε point στην δομή εξαρτάται από τα domains. Για παράδειγμα αν κάθε domain είναι το σύνολο των επιτρεπτών floats, τότε το κάθε point μπορεί να αποθηκευτεί σε έναν πίνακα από floats. Μια σελίδα με n points θα υλοποιείται με n τέτοιους πίνακες.

4.1.3 Regions

Στο εσωτερικό κάθε non leaf node (εσωτερικού κόμβου μη φύλλο) K η αναπαράσταση της διαμέρισης γίνεται είτε με χρήση kd trees (περίπτωση που δεν περιγράφεται εδώ).ή μέσω ενός συνόλου ν διαστάσεων παραλληλογράμμων. Εδώ περιγράφεται η δεύτερη περίπτωση που είναι και αυτή που χρησιμοποίησε ο Robinson

Τα παραλληλόγραμμα στο εσωτερικό των μη φύλων κόμβων ονομάζονται και regions. Κάθε region δείχνει και σε ένα κόμβο, ο οποίος είναι η ρίζα του υποδένδρου που στα φύλλα του υπάρχουν όλα ακριβώς τα εισηγμένα σημεία που εντάσσονται γεωμετρικά στο region. Αντίστροφα κάθε κόμβος K αντιστοιχεί σε ένα region. Όταν ο K είναι η ρίζα αυτό είναι το region όλου του χώρου. Όταν ο K δεν είναι η ρίζα αυτό είναι το region που δείχνει στον K. Όταν ο κόμβος K αντιστοιχεί στο region R

τότε λέμε πως “το R είναι η περιοχή του K” **(1)**

Στον non leaf κόμβο KNL αποθηκεύεται μια συλλογή από καταχωρήσεις που αντιπροσωπεύουν n στο πλήθος regions (ν διαστάσεων παραλληλόγραμμα) R1...Rn. Τα regions αυτά ονομάζονται subregions του KNL.

Τα subregions του KNL συναποτελούν γεωμετρική διαμέριση της περιοχής R του KNL. Δηλαδή δεν έχουν κανένα κοινό σημείο στο ν διάστατο χώρο και η ένωση τους δίνει το R **(2)**.

Τα regions μπορεί να τα δει κανείς γεωμετρικά ως παραλληλόγραμμα, και αλγεβρικά ως ένα καρτεσιανό γινόμενο n διαστημάτων του R Η δεύτερη θεώρηση μπορεί να διευκολύνει κάπως την κατανόηση ιδιοτήτων όπως η (2) και αλγορίθμων όπως το collision detection ν διαστάσεων. Περισσότερες λεπτομέρειες δίνονται λίγο πιο κάτω.

Η καταχώριση κάποιου region Rj στην συλλογή από subregions ενός κόμβου K έχει την μορφή:

Rj: (mind1 - maxd1, mind2 - maxd2 ... mindi - maxdi ... mindn - maxdn, link to page)

ή πιο σύντομα
(region, link to page)

Τα mindi-maxdi δείχνουν την μέγιστη και την ελάχιστη τιμή του Rj στην διάσταση i. Το mindi μπορεί να πάρει την τιμή -inf και το maxdi την τιμή +inf και έτσι τα παραλληλόγραμμα να έχουν άπειρου μήκους ακμές (σημ.. αν η περιοχή του χώρου από όπου λαμβάνονται τα σημεία είναι bounded η μέγιστη και η ελάχιστη τιμή

μπορεί να άλλη από τις -inf +inf)

Αυτός ο τρόπος προσδιορισμού των Rj είναι αντίστοιχος με τον ορισμό κριτηρίων αναζήτησης σε κάποιο range query. Για κάθε διάσταση d δίνεται και ένα ζεύγος τιμών mind, maxd, τιμές που είναι τα άκρα ενός διαστήματος Id. Όλα τα σημεία του Rj ανήκουν στο καρτεσιανό γινόμενο Id1xId2...xIdix...xIdn.

Υπάρχει μια ιδιαιτερότητα στην μορφή των Rj που έχει να κάνει με το ότι υπάρχει η απαίτηση να αποτελούν διαμέριση της περιοχής R του κόμβου K στον οποίον ανήκουν. Κατ αρχάς ο λόγος αυτής της απαίτησης είναι το ότι εξασφαλίζει πως για κάθε σημείο P του R υπάρχει μια και μόνο μια υποπεριοχή Rj που να το περιέχει. Έτσι η αναζήτηση κάποιου σημείου από την ρίζα προς τα φύλλα γίνεται σε μια μοναδική διαδρομή και χωρίς backtracking.

Τα boundaries (όρια, ακμές) ενός Rj πρέπει να τελειώνουν πριν αρχίσουν αυτά του διπλανού του σε χωροθέτηση. Αλλιώς τα σημεία πάνω σε αυτά τα boundaries θα άνηκαν σε δύο Rj. Τα Rj μπορούνε να έχουνε αυτήν την ιδιότητα αν τεθεί το δεξί (δεξί με την έννοια ορίου μέγιστης τιμής) boundary κάποιου Rj εκτός του Rj, και το αριστερό boundary εντός του. (το αντίστροφο θα είχε πάλι το επιθυμητό αποτέλεσμα). Δηλαδή τα Id που δίνουν το καρτεσιανό γινόμενο να είναι της μορφής [α,β), το δεξί άκρο ανοιχτό και το αριστερό κλειστό. Έτσι το αριστερό όριο είναι μέρος του Rj αλλά το δεξί "συμπληρώνεται" από κάποιο άλλο Rj'.

Με βάση τον παραπάνω περιορισμό κάθε Rj αναπαριστά το σύνολο σημείων P για τα οποία ισχύει

Rj.min-δ<=P[δ]<Rj.max-δ για κάθε διάσταση δ = Id1xId2...xIdix...xIdn

Το πεδίο link to page στην καταχώριση του Rj είναι δείκτης προς την θέση στην δευτερεύουσα μνήμη του κόμβου K που αντιπροσωπεύει το region Rj σύμφωνα με το (1). Επειδή κάθε κόμβος αποθηκεύεται ως ένα block στην δευτερεύουσα μνήμη ο

δείκτης θα περιέχει την φυσική διεύθυνση αυτού του block σε κάποιο δίσκο, και όχι κάποια διεύθυνση της κεντρικής μνήμης.

Η υλοποίηση των καταχωρίσεων regions γίνεται με τρόπο ανάλογο της υλοποίησης των καταχωρίσεων σημείων. Κάθε maxdij mindij μπορεί να αποθηκευτεί σε ένα πεδίο float int κτλ. Για παράδειγμα στην εργασία αυτή η υλοποίηση έγινε με δύο πίνακες μήκους ν (όπου ν ο αριθμός των διαστάσεων) τον max[] και τον min[].

4.1.4 Pages

Στο kdb, όπως και στα b+tree, οι κόμβοι που είναι φύλλα έχουνε διαφορετική οργάνωση από τους εσωτερικούς κόμβους και δίνεται σε αυτούς ξεχωριστή ονομασία. Οι κόμβοι φύλλα ονομάζονται point pages και οι κόμβοι μη φύλα ονομάζονται region pages. Όλα τα δεδομένα που εισάγονται από κάποιον χρήστη αποθηκεύονται στα φύλλα όπως στο b+tree, οι εσωτερικοί κόμβοι περιέχουν μόνο δεδομένα σχετικά με την οργάνωση της δομής.

Χρησιμοποιείται ο όρος page αντί του κόμβος καθώς στο kdb tree κάθε κόμβος αντιστοιχεί και σε μια σελίδα (block) στον δίσκο. Αντίστροφα κάθε σελίδα που χρησιμοποιείται από την δομή αντιστοιχεί και σε κάποιον κόμβο.

-Point Pages
-Region Pages

Ένα point page είναι κόμβος που περιέχει μια συλλογή από σημεία του ν διάστατου χώρου μαζί με δείκτες προς την θέση της εγγραφής που αντιπροσωπεύει το κάθε ένα (ή τις ίδιες τις εγγραφές).

Ένα region page περιέχει μια συλλογή από regions και δείκτες σε άλλα pages/κόμβους

Ο όρος region page είναι μάλλον εμπνευσμένος από την συγκεκριμένη υλοποίηση που πρότεινε ο Robinson για του εσωτερικούς κόμβους. (λίστα από regions αντί για εσωτερικό kd tree)

Πιο πάνω επίσης περιγράφτηκε ο τρόπος αποθήκευσης των δεδομένων στα region pages και τα point pages (συλλογές από (point,record) ή (region, link to page). Η λίστα καταχωρήσεων ενός point page μπορεί να ονομαστεί pointlist, ενώ αυτή ενός region page, subregionlist.

4.1.5 Invariants

Η δομή γίνεται κατανοητή από τις ιδιότητες invariant που διατηρεί το δένδρο μετά από κάθε λειτουργία insert ή delete.

1) Αν θεωρηθούν τα pages κόμβοι και οι δείκτες στις subregionlist ακμές, τότε αποτελούν μαζί ένα δένδρο του οποίου τα φύλλα είναι όλα point pages. Κάθε region page έχει ένα μη μηδενικό πλήθος δεικτών προς άλλα region pages ή point pages. Όλα τα pages πλην της ρίζας αποτελούν τον στόχο ενός δείκτη που βρίσκεται σε κάποιο άλλο region page.

2) Κάθε σημείο αποθηκευμένο σε point page πρέπει να ανήκει στο καρτεσιανό γινόμενο των πεδίων ορισμού κάθε διάστασης domain1 x domain2 x..... x domainn (δηλ στο σύνολο του χώρου)

3) Σε κάθε region page RP με γονέα κόμβο το region page parentRP, τα regions που είναι καταχωρημένα στην subregionlist του RP αποτελούν διαμέριση του region R για το οποίο υπάρχει καταχώριση (R, link στο RP) στο subregionlist του parentRP Διαμέριση σημαίνει πως δεν έχουνε κάποιο κοινό σημείο στον χώρο, και η ένωση τους είναι το R. (αν RP είναι η ρίζα και δεν έχει κόμβο πατέρα βλ 6) .

4)Το πλήθος καταχωρίσεων στις subregionlists είναι πάντα μικρότερο ή ίσο από κάποιον ακέραιο αριθμό όριο MAXSR. Το πλήθος καταχωρίσεων στις pointlists είναι πάντα μικρότερο η ίσο από κάποιον ακέραιο αριθμό όριο MAXP.

5) Σε κάθε point page PP με γονέα κόμβο το region page parentPP, τα σημεία που είναι καταχωρημένα στην pointlist του ανήκουν στο region R για το οποίο υπάρχει καταχώριση (R, link to PP) στον parentPP (αν το PP είναι η ρίζα βλ. 2)

6) Αν η σελίδα που είναι στην ρίζα του δένδρου είναι region page, τότε η ένωση των regions που έχει καταχωρημένα στην subregionlist της είναι ίση με τον συνολικό χώρο (δηλαδή με το καρτεσιανό γινόμενο των domains κάθε διάστασης).

Αυτοί οι περιορισμοί εξασφαλίζουν την ιεραρχική διαμέριση του χώρου. Κάθε σημείο του χώρου έχει για κάθε επίπεδο του δένδρου και μια μοναδική καταχώριση region που να το περιέχει(δεν υπάρχει overlap). Τα φύλλα αντιπροσωπεύουν το τελευταίο επίπεδο διαχωρισμού και εκεί αποθηκεύονται τα σημεία.

Οι ιδιότητες αυτές αποτελούν αναλλοίωτο με την έννοια ότι οποιαδήποτε ενέργεια insert ή delete πάνω στην δομή θα πρέπει μετά το πέρας της να μην την έχει τροποποιήσει έτσι ώστε να μην ισχύουν.

Στην πράξη τα invariants χρησιμοποιήθηκαν στην υλοποίηση του kdb tree για να προκύψουν τα preconditions και τα postconditions των συναρτήσεων για το delete insert split κτλ κάτι που διευκόλυνε την διαδικασία του debugging.

Παρατηρεί κανείς στα invariants την απουσία κάποιας εγγύησης για το ελάχιστο πλήθος καταχωρίσεων σε ένα page. (Σε ένα b-tree το 50% ενός block θα είναι πάντα γεμάτο). Η μέγιστη τιμή καταχωρήσεων στα subregionlists και pointlists καθορίζεται από κάποιες σταθερές MAXP και MAXSR. Όταν το πλήθος καταχωρίσεων ξεπεράσει τις σταθερές όρια, τότε ο κόμβος μεταβαίνει σε κατάσταση υπερχείλισης

και ακολουθεί διάσπαση του.

Όσον αφορά το ελάχιστο πλήθος καταχωρίσεων, στα pointlists μπορεί το πλήθος points να είναι και μηδέν (λόγω forced splitting από ψηλότερο επίπεδο που δεν στέλνει όλα τα points στα αριστερά ενώ κανένα κανένα point στα δεξιά). Σε subregionlists είναι τουλάχιστον 1 (πάντα μια εισαγωγή region σε γονέα κόμβο τοποθετεί σε αυτόν τα 2 region ή point pages που δημιουργήθηκαν σε κάποιο split, ενώ η διαγραφή των points και regions δεν ζητήθηκε να υλοποιηθεί)

4.2 Αλγόριθμοι

Οι βασικές λειτουργίες του kdb tree είναι 3, εισαγωγή σημείου, αναζήτηση με range query, και διαγραφή σημείου. Η διάσπαση κόμβου δεν αποτελεί λειτουργία. Δεν γίνεται ο χρήστης της δομής να προκαλέσει απευθείας την διάσπαση ενός κόμβου, αυτή γίνεται πάντα στα πλαίσια κάποιας εισαγωγής ή διαγραφής.

Η διάσπαση πάντως περιλαμβάνεται στα βήματα των αλγορίθμων των insert και delete, και κατά κάποιον τρόπο αποτελεί και το πιο σημαντικό βήμα τους, καθώς είναι αυτή που ορίζει πως μεταβάλλεται η δομή ώστε να διατηρούνται τα invariants.

Κατά συνέπεια πρώτα παρουσιάζεται η διαδικασία διάσπασης αναλυτικά και στην συνέχεια παρουσιάζονται και οι υπόλοιπες λειτουργίες.

4.2.1 Splitting

Το splitting (διάσπαση) είναι η διαδικασία μέσω της οποίας λύνεται το πρόβλημα της υπερχείλισης των κόμβων/pages και εξασφαλίζεται εν τέλει η αύξηση του ύψους του δένδρου ώστε να μπορεί να περιέχει περισσότερους κόμβους. Το splitting γίνεται τόσο σε region pages όσο και σε point pages.

Σύμφωνα με τους αλγόριθμους του Robinson ένα split έπεται χρονικά ενός αρχικού insert κάποιου σημείου σε point page. (Επίσης split μπορεί να ακολουθήσει και την διαγραφή σημείου, θυμίζω ότι σε αυτήν την εργασία η υλοποίηση της διαγραφής δεν ζητήθηκε). Σε αντίθεση με το b-tree όπου ένα split κόμβου μπορεί να προκαλέσει split μόνο σε προγόνους του, στα kdb trees ένα split στον κόμβο Κ ενδέχεται να προκαλέσει splits και σε κόμβους απογόνους του Κ.

Έτσι είναι δύσκολο να προβλεφθεί το πλήθος ενεργειών I/O που αντιστοιχούν σε ένα split του κόμβου Κ. Στην καλύτερη περίπτωση θα γίνουν μόνο 3 εγγραφές (των δύο νέων κόμβων που προκύπτουν και μια για την ενημέρωση του γονέα) Από εκεί και

πέρα μπορεί να διασπαστούν από κανείς μέχρι και όλοι οι κόμβοι του υποδένδρου του K, οπότε μπορεί να προστεθούν επιπλέον εγγραφές ανάλογα με το ύψος του K, και ανάλογα με την κατανομή regions και σημείων στο υποδένδρο του.

Η εξάρτηση από την κατανομή σημείων και regions οφείλεται στο ότι οι αλγόριθμοι διάσπασης δεν χρησιμοποιούν κριτήρια διάσπασης ανεξάρτητα από τα εισηγμένα δεδομένα. Λόγω αυτής γίνεται δύσκολη και η πιθανοθεωρητική ανάλυση για την εύρεση κάποιου μέσου όρου ενεργειών I/O.

4.2.1.1 Διάσπαση των point pages

Η πιο απλή περίπτωση split κόμβου είναι όταν αυτός είναι point page. Τότε η διαδικασία μοιάζει αρκετά με αυτήν την στατικής κατασκευής ενός kd tree. Στο paper του Robinson υπάρχει μόνο περιγραφή της διαδικασίας και δεν δίνονται βήματα κάποιου αλγορίθμου. Με βάση την περιγραφή αυτή (και την συγκεκριμένη υλοποίηση της στην εργασία) τα βήματα που ακολουθούνται είναι τα εξής:

0. Έστω γίνεται διάσπαση του point page PP με γονέα το region page PPP. Θέσε ως region R το region reg για το οποίο υπάρχει καταχώριση (reg, link to pp) στον PPP. Αν το PP ήταν η ρίζα και δεν έχει γονέα κόμβο θέσε ως R ολόκληρο τον χώρο (domain1 x domain2 x ... domainN)

1.Επέλεξε την διάσταση δ στην οποία θα διαχωριστεί το R (και θα διασπαστεί το PP).

2. Επέλεξε την τιμή v που προσδιορίζει την θέση του υπερεπιπέδου διαχωρισμού στον άξονα της d. Η v θα πρέπει να είναι μεγαλύτερη η ίση από το min-d του R στην συντεταγμένη d και μικρότερη από το max-d. Δηλαδή ο διαχωρισμός θα γίνει με ένα υπερεπίπεδο κάθετο τον άξονα της d που να τέμνει το R. min-d<=v< max-d

3. Χώρισε με βάση τα d,v του βήματος 2 το R στα regions LR,RR

LR: min-d = R.min-d, max-d = v
RR: min-d = v, max-d = R.max-d

Χώρισε το PP σε δύο νέα point pages LPP, RPP

4.Τα σημεία του PP αντιγράφονται στο LPP ή RPP ανάλογα με το ποιο subregion τα εμπεριέχει στο χώρο Στο LPP τοποθετούνται τα σημεία που ανήκουν στο LR και στο RPP όσα ανήκουν RR

5.Αν το PP έχει κόμβο γονέα τον PPP, αφαιρείται από το subregionlist του PPP η καταχώριση (R, link to PP). Στην θέση της εισάγονται δύο νέες καταχωρίσεις (LR, link to LPP) (RR, link to RPP). Αν το PP ήτανε η ρίζα ο PPP δεν υπάρχει, οπότε δημιουργείται μια νέα ρίζα και τοποθετούνται σε αυτήν οι καταχωρήσεις (LR, link to LPP) (RR, link to RPP).

6.Εφόσον το PP δεν ήταν η ρίζα ελέγχεται αν ο γονέας PPP υπερχείλισε κατά την εισαγωγή των καταχωρήσεων του βήματος 5 (συνολικά οι καταχωρήσεις του subregionlist του PPP θα έχουν αυξηθεί κατά μια) Αν έχει γίνει υπερχείλιση τότε ακολουθεί διάσπαση region page του PPP. Το PP πλέον δεν είναι μέρος της δομής.

Στο βήμα 1 η διάσπαση είναι τοπική. Εκτός των PP και R άλλα regions και point pages δεν επηρεάζονται άμεσα. (Εκτός αν ο γονέας του PP υπερχειλίσει και αυτός στο βήμα 6)

Τα βήματα 1 και 2 της επιλογής διάστασης και τιμής διαχωρισμού επηρεάζουν σε μεγάλο βαθμό την αποδοτικότητα της δομής. Για αντιπαραβολή στα b-trees το βήμα επιλογής διάστασης δεν υπάρχει καθώς μόνο μια διάσταση είναι δυνατή. Το βήμα επιλογής τιμής διαχωρισμού επιλέγει πάντα την median τιμή της ταξινομημένης

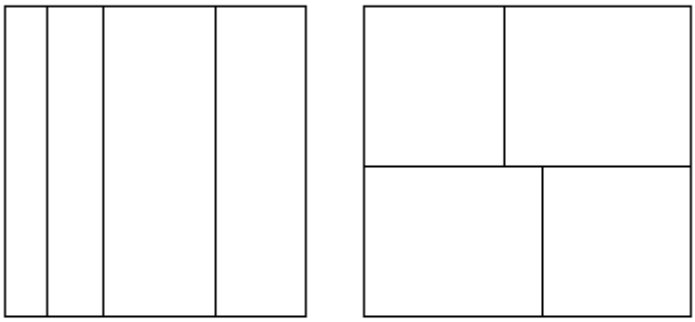
λίστας μονοδιάστατων σημείων της σελίδας. Έτσι κατά τον διαχωρισμό γίνεται μοίρασμα τον δεδομένων στους δύο νέους κόμβους και διατηρείται το ελάχιστο utilization του 50%.

Στα kdb trees όμως δεν υπάρχει median τιμή των σημείων ενός κόμβου αφού δεν ορίζεται κάποια διάταξη πολυδιάστατων στοιχείων με τις αναμενόμενες γεωμετρικές ιδιότητες (3.4.1.1). Έτσι το βήμα 2 είναι κάπως διαφορετικό από το αντίστοιχο στα b-trees. Το πως γίνεται αυτό το βήμα καθώς και αυτό της επιλογής διάστασης (βήματα 1,2) περιγράφεται στις επόμενες 2 ενότητες.

4.2.1.2 Κυκλική επιλογή διαστάσεων

Το βήμα 1 της επιλογής διάστασης διαχωρισμού (το οποίο δεν υφίσταται στο B-tree) δεν είναι trivial βήμα. Όπως αναφέρθηκε στην ενότητα 2 για το kd tree, πρέπει να χρησιμοποιηθούν όλες τις διαστάσεις στον διαχωρισμό του χώρου, καθώς όλες συμμετέχουν εξ ίσου στο μέτρο απόστασης.

Ένας διαχωρισμός που θα αγνοούσε κάποια διάσταση θα τοποθετούσε απομακρυσμένα σημεία στην ίδια υποπεριοχή του χώρου. Αν επιλέγεται μόνο ένα υποσύνολο διαστάσεων και όχι όλες ο χώρος δεν διαχωρίζεται με το πόσο κοντά στον χώρο είναι τα σημεία, αλλά εκφυλίζεται η διαδικασία στην ουσία σε διαχωρισμό με βάση την απόσταση της προβολής κάθε σημείου στον υποχώρο που σχηματίζουν οι επιλεγμένες διαστάσεις.



(Στην εικόνα το αριστερό region έχει διαμεριστεί με 3 διασπάσεις σε subregions που δεν είναι κυβοειδή. Στην δεξιά πάλι με 3 διασπάσεις τα subregions έχουν σχήμα πιο κοντά στον κύβο. Αυτό συνέβη γιατί αριστερά η διάσταση διάσπασης ήτανε σταθερά η χ, ενώ δεξιά μετά την πρώτη διάσπαση έγινε εναλλαγή στα 2 αρχικά subregions κι επιλέχτηκε η y. Αριστερά σημεία που βρίσκονται στο ίδιο subregion μπορεί να έχουνε μεγάλη απόσταση στον χώρο λόγω του ότι οι ακμές στην διάσταση y είναι πιο μακριές. Δεξιά αυτό το πρόβλημα μετριάζεται και έτσι είναι πιο μικρή η πιθανότητα μακρινά (και άρα όχι όμοια) σημεία να βρεθούν στο ίδιο region.)

Από την άλλη δεν υπάρχει κάποιο ordering των συντεταγμένων κάθε σημείου σε ένα υποσύνολο περισσότερων της μιας διαστάσεων, επομένως σε κάθε ξεχωριστό διαχωρισμό κόμβου πρέπει να επιλεγεί μόνο μια διάσταση. Επομένως πρέπει η επιλογή να γίνεται κάθε φορά με τέτοιον τρόπο ώστε σε ένα σύνολο τέτοιων επιλογών να συμμετέχουν όλες οι διαστάσεις εξ ίσου.

Η πιο "δίκαια" και ταυτόχρονα απλή στην υλοποίηση λύση είναι η επιλογή διαστάσεων με round robin δηλαδή κυκλικό τρόπο.

Αν ένα region R διαχωρίζεται στα R1 και R2 με βάση την d1, τότε τα R1 και R2 θα διαχωριστούν σε R11 R12 R21 R22 με βάση την d2, τα Rij σε Rijk με βάση την d3 κ.ο.κ μέχρι μετά την dn να ξαναέρθει η σειρά επιλογής της d1 και η διαδικασία να αρχίσει απ την αρχή.

Με την κυκλική επιλογή διαστάσεων τα regions γίνονται "k-cubical" δηλαδή τείνουν να έχουν ίδιο μήκος ακμών σε κάθε διάσταση και να είναι σαν ν διάστατοι κύβοι αντί για ν διάστατα παραλληλόγραμμα. Αυτό είναι επιθυμητό λόγω του μέτρου απόστασης που δίνει ίδια σημασία σε κάθε διάσταση.

Σε περιπτώσεις πάντως που η σημασία κάποιας διάστασης d είναι μεγαλύτερη από τις άλλες μπορεί η επιλογή των διαστάσεων να μην γίνεται με round robin τρόπο αλλά να επιλέγεται πιο συχνά για διάσπαση η d. Για παράδειγμα όταν είναι γνωστό πως η πλειονότητα των αναζητήσεων range θα είναι partial queries που αγνοούν μια διάσταση d2,. ο διαχωρισμός σε αυτήν δεν θα συνεισφέρει στον εντοπισμό εγγραφών (αν αναζητούνται σημεία που ανήκουν σε διάστημα της d και ο κόμβος είναι χωρισμένος με βάση την d2, αντί να οδηγηθεί η αναζήτηση σε ένα υποδένδρο αναγκαστικά τα ακολουθεί όλα) Η ιδέα περιγράφεται λεπτομερώς και στην ενότητα 2.4.1.

4.2.1.3 Προβλήματα στην διάσπαση λόγω κατανομής σημείων

Μετά την επιλογή κάποιας διάστασης δ, πρέπει να γίνει στο βήμα 2 η επιλογή τιμής ν στον άξονα της δ η οποία θα λειτουργήσει σαν threshold (κατώφλι) του διαχωρισμού. Η τιμή που θα επιλεγεί μπορεί να είναι ομοίως με την περίπτωση του b-tree η median τιμή στην διάσταση δ των σημείων, δηλαδή η median τιμή των τιμών Χ[δ] για κάθε σημείο Χ του διασπώμενου κόμβου.

Αυτή η πολιτική επιλογής της ν είναι παρόμοια με αυτή της κατασκευής των kd trees.

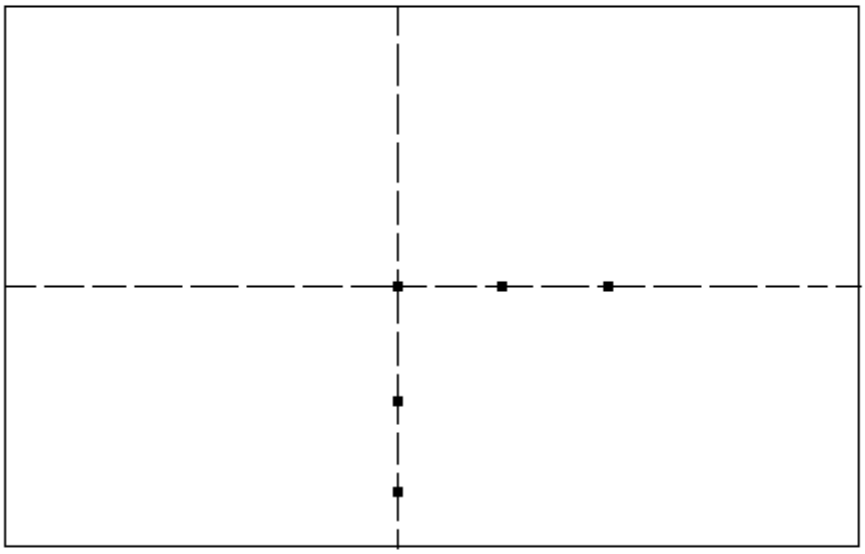
Θεωρητικά εξασφαλίζεται ότι τα σημεία μοιράζονται εξ ίσου τους δύο κόμβους που προκύπτουν από την διάσπαση. Όμως όταν μεγάλο μέρος των σημείων είναι ευθυγραμμισμένα πάνω σε μια μόνο τιμή m της συντεταγμένης δ αυτό δεν είναι δυνατόν να συμβεί.

Επειδή όλα σχεδόν τα σημεία θα έχουν X[δ]=m αναγκαστικά η m θα είναι και η median τιμή, οπότε v=m. Λόγω του ορισμού των regions που θέλει το δεξί μέρος κάθε ακμής/boundary του region ανοικτό και το αριστερό κλειστό, πιο πολλά σημεία θα πάνε στο δεξιό κόμβο από ότι στον αριστερό. Όλα τα σημεία με X[δ]=m=v πέφτουνε πάνω στο boundary διαχωρισμού και άρα πάνε δεξιά.

Μάλιστα αν όλα τα σημεία του point page είναι ευθυγραμμισμένα τότε στο αριστερό point page δεν θα πάει κανένα σημείο.

Μια πρώτη λύση σε αυτό είναι να γίνει νέα επιλογή διάστασης (παρεμπιπτόντως αυτό σημαίνει ότι τα δ και v πρέπει να επιλέγονται αναγκαστικά μαζί σε ένα βήμα και όχι ξεχωριστά το δ) και νέα επιλογή κατωφλίου.

Όμως για γεωμετρικούς λόγους και σε αυτήν την περίπτωση είναι δυνατόν να μην υπάρχει τρόπος να γίνει ισομερής κατανομή των σημείων.



(Στην εικόνα όποιο και από τα δύο μονοδιάστατα υπερεπίπεδα -με τις

διακεκομμένες γραμμές- και να επιλεγεί το ένα subregion που θα προκύψει θα πάρει όλα τα Points ενώ το άλλο κανένα)

Στην χειρότερη των περιπτώσεων, αν όλα τα αποθηκευμένα σημεία έχουνε σε κάθε διάσταση τις ίδιες συντεταγμένες τότε δεν υπάρχει διαχωρισμός που να μην δώσει ένα εκ των 2 νέων point pages κενό. (στην δομή όπως την περιγράφει ο Robinson όμως δεν επιτρέπονται τέτοιες πολλαπλές καταχωρήσεις στο ίδιο σημείο)

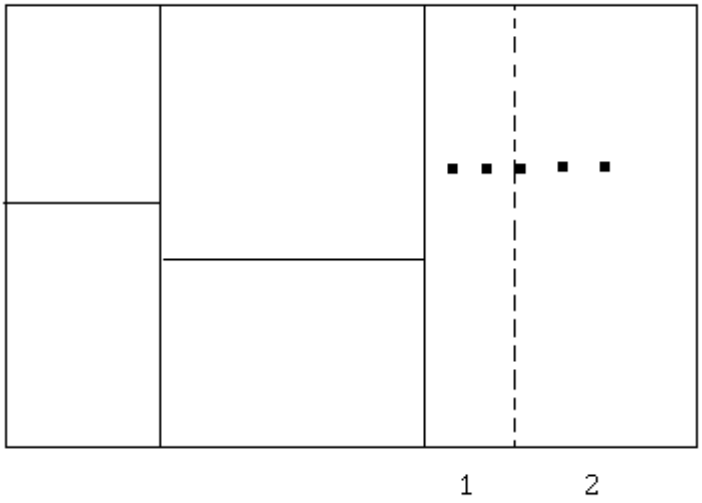
Χωρίς πολλαπλές καταχωρίσεις είναι σίγουρο ότι μια κυκλική εναλλαγή των διαστάσεων, με επιλογή της median τιμής για ν κάθε φορά, θα βρει εν τέλει έναν διαχωρισμό που δεν θα δίνει κάποια από τις δύο νέες point pages κενή.

Αν δηλαδή στην χειρότερη περίπτωση όλα τα σημεία είναι ευθυγραμμισμένα σε όλες πλην μια διάστασης

για κάθε X1,X2,d!=δ X1[d]= X2[d]

Τότε απλώς ο διαχωρισμός γίνεται στην εναπομείνουσα διάσταση δ.

Η αλλαγή διάστασης κατά αυτόν τον τρόπο μπορεί να μην διατηρεί το ν-cubicality της προηγούμενης ενότητας



(Στην εικόνα η διάσπαση του δεξιού region στην διάσταση y θα δώσει αναγκαστικά ένα κενό point page καθώς όλα τα σημεία είναι ευθυγραμμισμένα στην y. Από την άλλη η διάσπαση στην χ και στο μεσαίο σημείο θα κάνει τα subregions 1 και 2 να μην έχουν σχήμα κύβου)

Είναι επιλογή πολιτικής αν θα επιτραπούν κενά point regions με σκοπό να διατηρηθεί το cubicality. Κενά point regions πάντως είναι πιθανόν να προκύψουν ούτως ή άλλως από forced splits (αναγκαστικές διασπάσεις) εξαιτίας της διάσπασης κόμβων σε ανώτερα επίπεδα (4.2.1.5)

4.2.1.4 Διάσπαση των Region Pages

Η διαδικασία διάσπασης ενός region page είναι πιο πολύπλοκη από αυτήν των point pages. Μάλιστα στον αλγόριθμο που προτείνει ο Robinson μπορεί κανείς να εντοπίσει και σημεία που υλοποιούνται με μη trivial (με στοιχειώδη βήματα) τρόπο για τον οποίο δεν δίνονται περισσότερες λεπτομέρειες.

Η διάσπαση ξεκινάει όταν σε κάποιο region page RP παραβιαστεί η συνθήκη overflow (υπερχείλιση). Αυτή έχει να κάνει με τα μέγιστα όρια πλήθους καταχωρήσεων στα suregionlists. Η υπερχείλιση συμβαίνει χρονικά μετά από την

διάσπαση ενός κόμβου υιού του RP η οποία αφαιρεί από το subregionlist του RP ένα region και προσθέτει 2 νέα.

Η διάσπαση ενός region page έχει κάποια κοινά στοιχεία με την διάσπαση ενός point page. Και στις δύο περιπτώσεις το page αντιπροσωπεύει ένα παραλληλόγραμμο R ν διαστάσεων για το οποίο είτε υπάρχει καταχώριση τύπου (R, link στο page) στον γονέα του page, είτε, όταν το page βρίσκεται στην ρίζα της δομής, αποτελεί ολόκληρο τον χώρο. Και στις δύο περιπτώσεις επίσης αναζητείται μια διάσταση και μια τιμή σε αυτήν που θα χωρίσει εσωτερικά το region R σε δύο subregions.

Κατά την διάσπαση, στα point pages μοιράζονται σε δύο νέα point pages τα points, ενώ στα region pages, μοιράζονται σε δύο νέα region pages τα subregions του region page που διασπάται. Το μοίρασμα θα πρέπει να γίνει κατά το δυνατόν εξ ίσου και στα δύο νέα τμήματα ώστε να επιβληθεί η ιεραρχικότητα και να υπάρχει μεγαλύτερο utilization στους κόμβους/pages.

Παρακάτω δίνεται ο αλγόριθμος όπως είναι στο paper του Robinson. Σε αυτόν υποτίθεται ότι έχει ήδη γίνει η επιλογή των δ και ν

S0. Επέλεξε διάσταση και θέση υπερεπιπέδου διάσπασης (d,v)

For each (subr, link to subrP) in RP:
Sl. If subr lies to the left of v, add
(subr, subrP) to the LRP.
S2. If subr lies to the right of v, add
(subr, subrP) to the RRP.

S3. Otherwise:

S3.1. Split the page referenced by subrP
along v, resulting in pages
subLP and subRP.
S3.2. Split subr along v, resulting in
regions leftsubr and rightsubr.
S3.3. Add (leffsubr, subLP) to LRP,
and (rightsubr,subRP) to RRP..

Στην ουσία μετά την επιλογή των d,v ο υπόλοιπος αλγόριθμος είναι απλός. Στα βήματα S1 και S2 έχει σημασία να οριστεί τι θα πει το να είναι κάποιο region αριστερά και δεξιά του v.

Αριστερά είναι όταν για την διάσταση d το max-d του region είναι μικρότερο του v. Δεξιά όταν το min-d είναι μεγαλύτερο ή ίσο του v.

Αν και είναι κάπως unintuitive το ότι αν το v είναι ίσο με το min-d το region θεωρείται ότι είναι δεξιά του v (δηλαδή είναι δεξιά του v αν και αυτό βρίσκεται στο εσωτερικό του και πάνω boundary) αυτό είναι λογικό καθώς σε αντίθετη περίπτωση ο κόμβος θα έπρεπε να διασπαστεί στο boundary κάτι που θα προκαλούσε προβλήματα που περιγράφονται στο 4.2.1.8

Άρα τα βήματα 1 και 2 μοιράζουν σε δύο νέους κόμβους τα subregions του subregionlist που δεν τέμνονται στο εσωτερικό τους από το υπερεπίπεδο d,v

Στο βήμα S3 είναι το σημείο που όσα regions τέμνονται εσωτερικά από το d,v διασπώνται αναδρομικά. Αυτό είναι το σημείο που οδηγεί στο forced splitting των κόμβων που δείχνουν αυτά τα regions. (βλ 4.2.1.5)

Οι διασπάσεις αυτές παράγουν τα διπλάσια regions/pages από όσα τεμνόταν από το d,v. Το κάθε ένα από αυτά όμως βρίσκεται αριστερά η δεξιά του d,v οπότε μπορεί να προστεθεί κατάλληλα στο LRP ή το RRP. Το LRP είναι το Left Region Pafe που προκύπτει από την διάσπαση ενώ το RRP το Right Region Page.

Επιπλέον θα πρέπει να ακολουθήσουν και τα βήματα τροποποίησης του πατέρα PRP του region page που διασπάται RP

S4. Αν το RP ήταν η ρίζα δημιούργησε μια νέα ρίζα region page και θέσε την γονέα του RP.
S5. Αφαίρεσε από τον γονέα PRP την καταχώριση για το RP και προσέθεσε δύο νέες για τα δύο νέα region pages που δημιουργήθηκαν
S6. Αν ο PRP είναι σε overflow κάνε split αλλιώς τερμάτισε

Στην υλοποίηση χρησιμοποιείται μια κάπως διαφορετική σειρά εκτέλεσης. Σε αυτήν η κατασκευή των δύο νέων pages γίνεται στο τέλος, δηλαδή τα βήματα S1, S2 γίνονται μετά το S3. Γίνεται πρώτα επιλογή όσων subregions τέμνονται, γίνεται διάσπαση τους, και τα νέα regions προστίθενται στην subregionlist η οποία πλέον περιέχει regions μόνο δεξιά και αριστερά του d,v. Συγκεκριμένα αν η σελίδα είχε k subregions και από αυτά τα l τέμνονταν από το d,v τότε πριν γίνει ο διαχωρισμός η subregionlist θα έχει (k-l)+2*l regions, με μέγιστη δυνατή τιμή τα 2*l regions όταν k=l

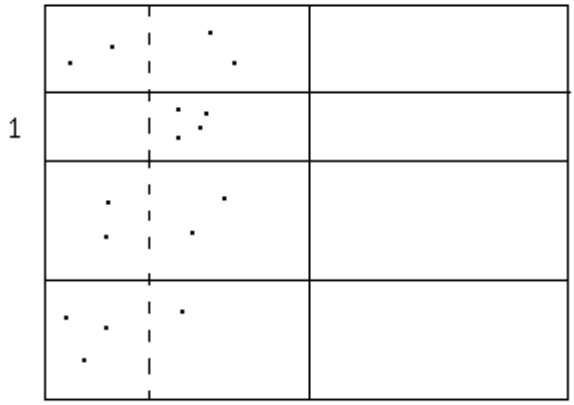
Στις επόμενες ενότητες παρουσιάζονται ορισμένα χαρακτηριστικά της διάσπασης. Πρώτα το forced splitting, και στην συνέχεια ορισμένα προβληματικά σημεία σχετικά με την επιλογή των d, v που δεν διευκρινίστηκαν στο paper του Robinson

4.2.1.5 Forced Splitting

Το πρόβλημα με την διάσπαση ενός region R κάποιου region page RP, όπου το R έχει χωριστεί σε διαμέριση από τα subregions της subregionlist του RP, είναι το ότι η διάσπαση αυτή προκαλεί συνήθως και την διάσπαση των subregions της διαμέρισης. Δηλαδή στο δένδρο η διάσπαση ενός κόμβου γονέα μπορεί να προκαλέσει την διάσπαση των άμεσων υιών του και αναδρομικά όλων των απογόνων του.

Αυτό δεν συμβαίνει στο b-tree και είναι ένας από τους λόγους που το kdb tree δεν θεωρείται αρκετά αποδοτικό για να χρησιμοποιηθεί σε συστήματα DBMS. Το φαινόμενο ονομάζεται forced splitting, όπου ο χαρακτηρισμός forced δηλώνει πως ο κόμβος αναγκάζεται να διασπαστεί ενώ δεν είναι σε κατάσταση overflow. Οι πολλές διασπάσεις οδηγούν σε αύξηση του ύψους του δένδρου. Ακόμα χειρότερα οι διασπάσεις δεν μπορούν να γίνουν με τρόπο που να εξασφαλίζει ότι οι δύο νέοι κόμβοι θα είναι κατά 50% γεμάτοι, να μοιραστούν δηλαδή εξ ίσου τις καταχωρίσεις του subregionlist του διασπώμενου κόμβου.

Η διάσπαση forced split δεν περιέχει την επιλογή d,v (επιλογή υπερεπιπέδου). Το d,v είχε επιλεγεί από κάποιον πρόγονο κόμβο όπου ξεκίνησαν τα forced splits. Αν το d,v επιλεγόταν εκ νέου οι κόμβοι απόγονοι δεν θα ήταν αριστερά ή δεξιά του υπερεπιπέδου διάσπασης του προγόνου.



(Στην εικόνα το forced split που δηλώνει η διακεκομμένη γραμμή διασπά αναγκαστικά το point page που δείχνει το 1 ώστε να προκύπτει ένα κενό από σημεία Point page)

Αναδρομικά τα forced splits κατεβαίνουν στο δένδρο μέχρι στην χειρότερη περίπτωση να φτάσουν στα φύλλα. Μιας και δεν γίνεται πουθενά προσπάθεια για να βρεθεί καλός συνδυασμός d,v οι διασπάσεις δεν μπορούν να έχουν εγγυήσεις για το utilization.

Αυτό συνεπάγεται χαμηλό utilization κόμβων, ενώ με τον αλγόριθμο που δίνει ο Robinson ενδέχεται αμέσως μετά την διάσπαση οι δύο νέοι κόμβοι να είναι και πάλι σε overflow και να χρειάζεται να διασπαστούν ξανά αμέσως μετά την δημιουργία τους βλ. 4.2.1.6 (αν και ο ίδιος αναφέρει ότι αυτή η συμπεριφορά δεν είναι επιτρεπτή, δεν εξηγεί λεπτομερώς την διαδικασία μέσω της οποίας αποφεύγεται αλγοριθμικά).

Το θετικό είναι ότι δεν γίνεται να προκληθεί overflow σε πρόγονο από την διάσπαση απογόνου σε forced split. Αυτό είναι σημαντικό γιατί σημαίνει δεν θα προκύπτουν καταστάσεις όπου ένα split σε κόμβο ψηλά στο δένδρο θα οδηγήσει σε αλλεπάλληλα forced splits κάτω από αυτόν με διαδρομή ζιγκ ζαγκ που επιστρέφει στον αρχικό splitted κόμβο ζητώντας του να διασπαστεί ενώ ακόμα δεν έχει ολοκληρωθεί η

διάσπαση του**(1)**

Αν ένας κόμβος διασπαστεί με forced split θα πρέπει αναγκαστικά να έχει σπάσει και ο γονέας του. Μιας και ο πατέρας έχει διασπαστεί θα έχει χώρο να δεχθεί την πληροφορία που ανεβάζει ο υιός κατά την δικιά του διάσπαση. Καθότι αν αυτή η διάσπαση δώσει τα νέα pages child1, child2 ταυτόχρονα λόγω του ότι αναγκαστικά έχει διασπαστεί ο γονέας, θα υπάρχουν ήδη και τα νέα pages father1,father2 ώστε το κάθε father page να πάρει από ένα child page και να μην προκληθεί κάποιο overflow.

Αν ο γονέας έχει διασπαστεί με forced split δεν θα ήτανε σε κατάσταση overflow όταν έγινε αυτό. Τότε ακόμα και στην χειρότερη περίπτωση που το forced split γίνεται σε όλα τα n subregions του γονέα, το πλήθος subregions που θα προκύψουν θα είναι το πολύ 2*n και για κάθε αριστερό του d,v region θα υπάρχει και ένα δεξί, οπότε θα μπορούν να χωριστούν στα δύο νέα pages της διάσπασης του γονέα.

Αναδρομικά αυτή η διαδικασία φτάνει στον πρόγονο που δεν διασπάστηκε με forced split αλλά με overflow. Αυτός έχει τόσα regions όσα το όριο του overflow οπότε το 2*n δεν χωράει σε δύο pages. Επειδή όμως γίνεται η υπόθεση ότι το split λόγω overflow επιλέγει τέτοια d,v ώστε τα regions που θα προκύψουν να είναι λιγότερα από 2*n και να διαμοιράζονται στα δύο νέα pages (4.2.1.6) η αναδρομή τελειώνει με επιτυχία αποδεικνύοντας το (1)

4.2.1.6 Προβλήματα με την συνθήκη overflow

Δίνεται αρχικά ένα παράδειγμα region page που θα βοηθήσει στο να κατανοήσει κανείς διαισθητικά τα προβλήματα.

Έστω σε 2 διαστάσεις έχουμε ένα region page RP που αντιπροσωπεύει το παραλληλόγραμμο R με όρια minx,miny =0 και maxx,maxy=100. Έστω ότι το όριο πλήθους subregions της subregionlist πριν βρεθεί ένα region page της σε κατάσταση overflow είναι τα 11 subregions. Αρχικά έστω ότι το RP χωρίζεται ήδη σε 10

subregions sr0...sr9 με το κάθε sri να έχει τα εξής όρια (χρησιμοποιείται εδώ συμβολισμός ανάλογος των γλωσσών προγραμματισμού όπου A.X σημαίνει πως το X είναι μια ιδιότητα του αντικειμένου A)

sri.minx=0+i*10
sri.maxx=0+(i+1)*10
sri.miny=0
sri.maxy=100

για i=0 έως 9

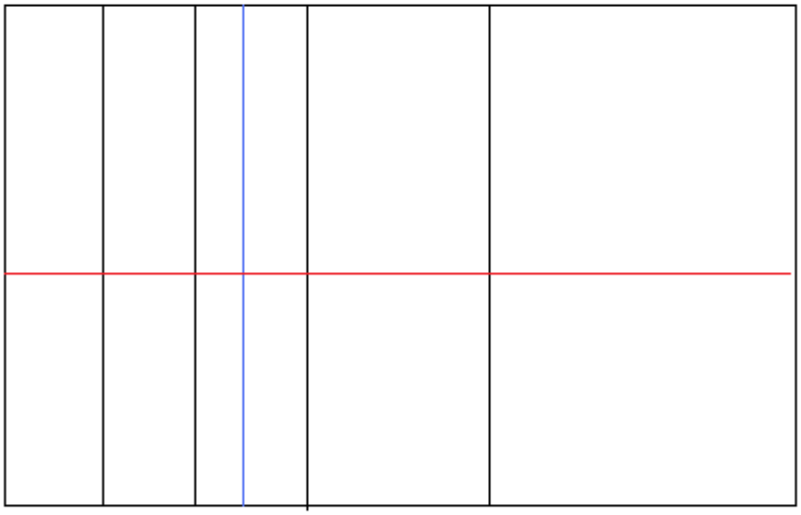
Δηλαδή το RP χωρίζεται οριζοντίως σε 10 εφαπτόμενα παραλληλόγραμμα με τις βάσεις τους τοποθετημένες ανά 10 μονάδες στον άξονα x και με ύψος ίσο με αυτό του R.

Αν ο κόμβος στον οποίο δείχνει κάποιο srj διασπαστεί τότε το RP μπαίνει σε κατάσταση overflow γιατί η subregionlist του έχει ήδη 10 καταχωρίσεις και η διάσπαση ενός υιού τις αυξάνει κατά ένα, οπότε συνολικά θα περάσουν σε πλήθος το όριο υπερχείλισης. Υποθέτω για το παράδειγμα ότι και ο srj διασπάτε κατά χ ώστε το σύνολο των subregions να παραμένει διατεταγμένο στον άξονα χ όπως πριν.

Όπως στη περίπτωση των point pages σε αυτό το σημείο πρέπει να γίνει επιλογή μιας διάστασης δ και μιας τιμής ν για να χωριστεί το region page. Γίνεται η υπόθεση ότι η διάσταση δ που επιλέγεται είναι η χ, στον άξονα της οποίας είναι διατεταγμένα τα subregions. Πιο μετά γίνεται παρουσίαση της περίπτωσης επιλογής άλλης διάστασης.

Αρχικά ερευνάται η περίπτωση που για ν επιλέγεται τιμή που ανήκει στο εσωτερικό κάποιου από τα 11 sri, έστω στο srk. Τότε το srk θα πρέπει να διασπαστεί. Από τα 11 regions στην subregionlist, όσα βρισκόταν δεξιά του ν στον άξονα χ (και άρα δεξιά του srk καθώς αυτό περιέχει το ν) θα πάνε όλα στο δεξί page της διάσπασης του RP.

Μαζί τους θα πάει και το δεξί τμήμα που θα προκύψει από την διάσπαση του subrk. Ομοίως όσα subregions βρισκότανε αριστερά του v θα πάνε στο αριστερό page της διάσπασης του RP μαζί με το αριστερό τμήμα του srk.



(Εικ.1 Στην εικόνα τα subregions ενός region διατεταγμένα στον άξονα χ. Όριο υπερχείλισης τα 5 subregions Ένα split στο μεσαίο region με την μπλε γραμμή δίνει καλό utilization και λίγα forced splits άλλα έχει πρόβλημα με το n-cubicality. Ένα split σε διάσταση πλην της χ όμως επιφέρει αναγκαστικά την δημιουργία 2*5 νέων subregions και δύο κόμβους που είναι πάλι σε overflow)

Στο παράδειγμα παρατηρείται ότι με την επιλογή v που έγινε μόνο το subrk πρόκειται να διασπαστεί και τα υπόλοιπα 10 subri είναι δεξιά ή αριστερά του v. Αν ειδικά το subrk είναι το "μεσαίο" σε σειρά ταξινόμησης στον άξονα χ από τα 11, τα subregions θα μοιραστούν εξ ίσου στον αριστερό και τον δεξιό νέο κόμβο (5 αριστερά, 5 δεξιά, 1 από την διάσπαση αριστερά, 1 από την διάσπαση δεξιά).

Αυτή η ίση κατανομή στο παράδειγμα δίνει μια αρχική ιδέα για τον τρόπο επιλογής του v. Κατ αρχήν αυτός ο τρόπος θα πρέπει να είναι ένας εύκολος στην υλοποίηση

και γρήγορος στην εκτέλεση αλγόριθμος που θα χρησιμοποιεί μόνο τοπικές πληροφορίες γιατί αλλιώς θα χρειάζεται να κάνει μεγάλο πλήθος ενεργειών I/O για να συλλέξει πληροφορίες από το υποδένδρο του κόμβου RP που διασπάται. Δηλαδή ο αλγόριθμος θα πρέπει να έχει ευρεστικά στοιχεία. Με βάση το παράδειγμα μια πρώτη προσέγγιση κατασκευής αυτού του αλγορίθμου είναι η ακόλουθη.

Προσέγγιση 1:
Βήμα 1: Βρες το "μεσαίο" με βάση την επιλεγμένη διάσταση subregion srk στην subregion list του διασπώμενου κόμβου

Βήμα 2: Βρες μια τιμή l για την διάσταση δ στο εσωτερικό του subrk κσι θέσε v=l

Το βήμα 2 μπορεί να γίνει με την εύρεση της μέσης τιμής της δ στο subrk

Βήμα 2.1: l=srk.minδ + (srk.maxδ - srk.minδ)/2

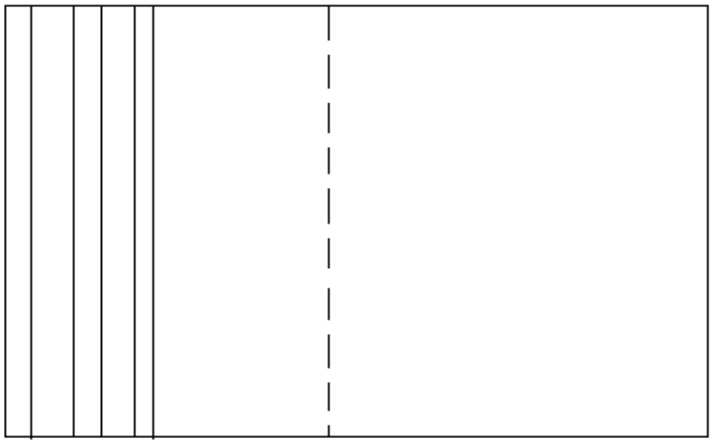
Αυτός ο τρόπος εύρεσης του v γίνεται με λίγες μόνο πράξεις, και αν τελικά δούλευε σε όλες τις περιπτώσεις θα ήταν ιδανικός γιατί θα εξασφάλιζε ίση διανομή των κόμβων. Παρακάτω όμως γίνεται η διαπίστωση ότι αν τα subregions δεν είναι ταξινομημένα ακριβώς πάνω στην διάσταση δ τότε τα αποτελέσματα του παραπάνω αλγορίθμου δεν είναι καλά.

Μια άλλη προσέγγιση, πιο "χοντροκομμένη" είναι η εξής

Προσέγγιση 2:
Βήμα 1: Θέσε το v στο μέσο στην επιλεγμένη διάσταση του region R του RP με τρόπο όμοιο με του βήματος 2.1 παραπάνω

Αυτή χρησιμοποιεί ακόμα λιγότερες τοπικές πληροφορίες από την πρώτη (μόνο το R αντί όλα τα sri)

Στην 2η προσέγγιση υπάρχει το εξής πρόβλημα. Μπορεί και τα 10 πρώτα subregions να βρίσκονται αριστερά του μέσου του R (και το 11ο να είναι πολύ μακρύ) και έτσι η διάσπαση να δώσει 11 subregions αριστερά, 10 τα αρχικά αριστερά και 1 από την διάσπαση του 11ου subregion.(Εικ. 2)



(Εικ 2. Στην εικόνα η διάσπαση στην μέση ως προς χ του εξωτερικού παραλληλογράμμου στέλνει τα περισσότερα παραλληλόγραμμα αριστερά 5:1)

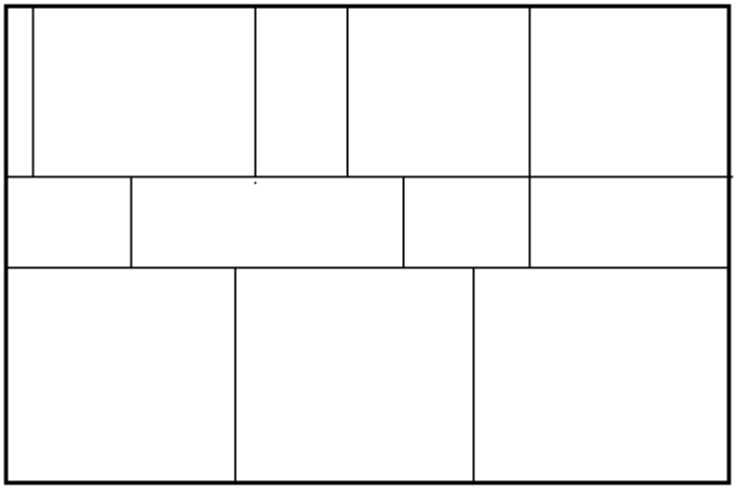
Το αριστερό αποτέλεσμα της διάσπασης του region page LRP θα θέλει ξανά σπάσιμο γιατί είναι πάλι σε overflow καθώς έχει 11 subregions (περισσότερα για αυτό παρακάτω) ενώ στο δεξί region page που παράγει η διάσπαση, RRP, θα πάει μόνο ένα subregion, το δεξί τμήμα της διάσπασης του 11ου subregion του RP. Έτσι η κατανομή δεξιά και αριστερά των subregions είναι άνιση (11 προς 1). Επομένως λόγω του χαμηλού utilization στον ένα κόμβο η 2η προσέγγιση δεν δίνει τόσο καλά αποτελέσματα όσο το να βρεθεί πρώτα το μεσαίο region και να γίνει η διάσπαση με επιλογή τιμής v μέσα σε αυτό.

Αν επικεντρωθεί κανείς στην πρώτη προσέγγιση παρατηρεί ότι, μόνο ένα subregion και αναδρομικά οι απόγονοι, φαίνεται να διασπάται και έτσι το forced split περιορίζεται σε ένα υποδένδρο ύψους κατά ένα χαμηλότερο από ότι αν κάθε sr ήταν να διασπαστεί.

Αυτή η εικόνα όμως δεν ισχύει στην γενική περίπτωση και η πρώτη προσέγγιση δεν εγγυάται πάντα ότι μόνο σε ένα subregion θα γίνει forced split. Αντίθετα έχει ισχύ μόνο όταν τα subregions "κατανέμονται" στον άξονα της επιλεγμένης διάστασης δ, όταν δεν υπάρχει δηλαδή τιμή l της διάστασης δ τέτοια ώστε το σύνολο των σημείων P με P[δ]=l να υπάρχουν εντός περισσότερων (η λιγότερων)του ενός subregions του RP.

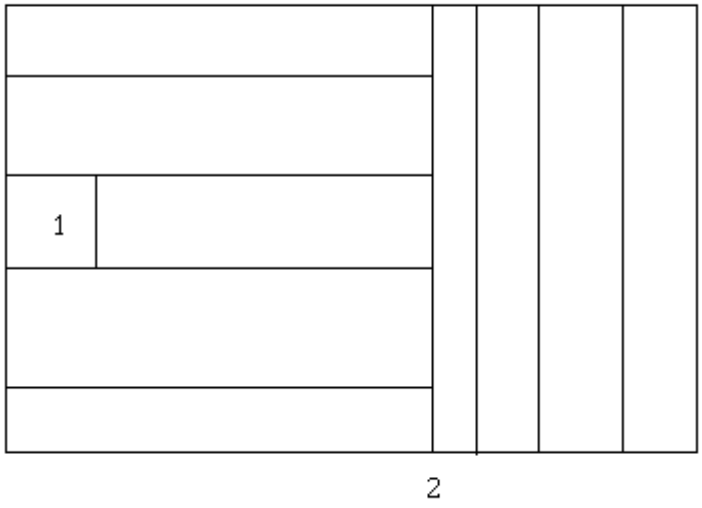
Όταν τα subregions διαμερίζουν έναν μονοδιάστατο άξονα τότε η σειρά τους σε αυτόν αποτελεί διάταξη και είναι εύκολο να βρεθεί το μεσαίο srk και στη συνέχεια να επιλεγεί η μέση τιμή της συντεταγμένης δ σε αυτό ώστε να τεθεί ως η v του split.

Δεν είναι εύκολο να βρεθεί το "μεσαίο" subregion αν τα subregions είναι έτσι χωροθετημένα ώστε περισσότερα του ενός να προβάλλονται σε κάποιο διάστημα του άξονα της δ. Αν για παράδειγμα δ=χ και τα subr1...subr5 είναι τοποθετημένα πάνω από τα subr6...subr9 με τις βάσεις των subr6..sunr9 πάνω στον άξονα χ, τότε δεν μπορεί να πει κανείς πιο από όλα είναι το μεσαίο ως προς την δ. (Εικ. 3)



(Εικ.3 Σε αυτήν την διαμέριση του εξωτερικού region δεν υπάρχει “μεσαίο” ως προς χ subregion)

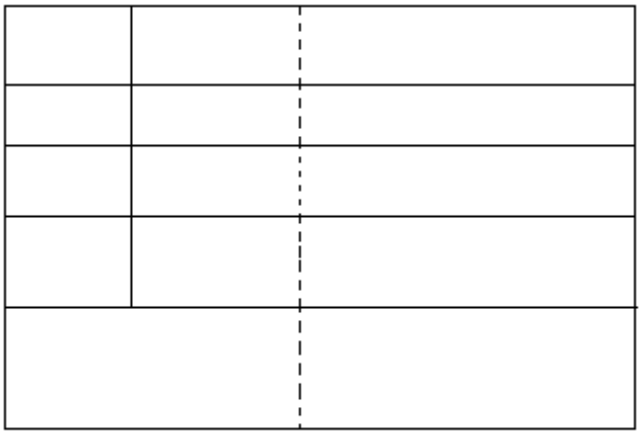
Μια λύση είναι να βρεθεί το μεσαίο subregion με βάση μια ταξινόμηση ως προς την ελάχιστη τιμή minδ του κάθε subregion στον άξονα της δ. Τότε τα subregions έχουνε μεν μια διάταξη αλλά αυτή μπορεί να είναι άλλες φορές παραπλανητική και άλλες ούτε καν διάταξη αφού ενδέχεται τα regions να έχουνε την ίδια ακριβώς minδ. Για παράδειγμα αν τα 7 πρώτα subregions έχουν κάθετη χωροθέτηση και όλα ως xmin την τιμή 0 ενώ το 5ο subregion (σε σειρά που είναι καταχωρημένα στην subregionlist και όχι στον χώρο) είναι στενότερο από τα άλλα, τότε επειδή όλα έχουν ίδιο xmin μπορεί επιλεχθεί το 5ο ως μέσο, και θέτοντας ως v την μέση τιμή χ αυτού τα υπόλοιπα 5 να διασπαστούν αναγκαστικά.(Εικ. 4)



(Εικ.4 Αν η διάσπαση γινότανε στην κάθετη γραμμή στο σημείο 2 η κατανομή που θα προέκυπτε θα ήταν η 6:4 και δεν θα χρειαζόταν forced split. Αν το region 1 θεωρηθεί 5ο σε σειρά ως προς minx και γίνει το split σε αυτό με την προσέγγιση 1 τότε θα γίνουν 4 διασπάσεις η κατανομή θα είναι 5:11 κι ο δεξιός κόμβος θα είναι πάλι σε overflow)

Ενώ δηλαδή αναζητήθηκε το “μεσαίο" subregion με σκοπό να διασπαστεί μόνο αυτό έγινε διάσπαση πολύ περισσότερων subregions. Αν από την άλλη το 5ο region είναι μακρύτερο, και η μέση του στην δ μεγαλύτερη από το μήκος των υπολοίπων subregions, τότε η κατανομή δεν θα γίνει ισομερής (θα πάνε όλα τα subregions αριστερά Εικ. 5). Το συμπέρασμα είναι ότι και αυτός "ευρεστικός" αλγόριθμος

επιλογής ν μπορεί να επιφέρει ή πολλές διασπάσεις ή/και ανισομερή κατανομή στα δύο νέα pages.



(Εικ. 5 Διάσπαση στο μέσον του 5ου κατά minx κάτω region δίνει 9:5 και 5 forced splits. Αν η διάσπαση γινόταν πάνω στην μη διακεκομμένη κάθετη γραμμή θα έδινε κατανομή 5:5 και μόνο ένα forced split)

Το πρόβλημα μάλλον δεν είναι των συγκεκριμένων αλγορίθμων αλλά γεωμετρικό και υπολογιστικό. Δεν υπάρχει κάποια τοπική στην κόμβο πληροφορία για την κατανομή κάτω από αυτόν. Δεν γνωρίζει ο κόμβος πως είναι κατανεμημένα σημεία και regions στο υποδένδρο του παρά μόνο φορτωθούν όλοι οι κόμβοι του υποδένδρου με I/O. Κάτι τέτοιο είναι εξαιρετικά δαπανηρό και σε πλήθος ενεργειών I/O ισοδυναμεί με την πλήρη αναδιάρθρωση του υποδένδρου.

Ακόμα και αν ο κόμβος K είχε πλήρη πληροφορία για το υποδένδρο, γεωμετρικά μπορεί να είναι αδύνατον να αποφευχθούν και οι πολλές διασπάσεις και η άνιση κατανομή regions. Τα subregions μπορεί να τοποθετηθούν στον χώρο με τέτοιο τρόπο ώστε για να γίνει ισομερής κατανομή να απαιτούνται και πολλές διασπάσεις. (βλ 4.2.1.7)

* * *

Ως τώρα παρουσιάστηκε (βάση παραδείγματος) η επιλογή του v (θέση υπερεπιπέδου διαχωρισμού στον άξονα δ) εφόσον το δ (διάσταση για την οποία γίνεται ο διαχωρισμός) έχει ήδη καθοριστεί. Η επιλογή του δ όμως δεν είναι ούτε και αυτή απλό βήμα.

Στο παράδειγμα που δόθηκε πιο πάνω υπάρχουνε 11 regions ταξινομημένα κατά σειρά στον άξονα χ και επιλέγεται διάσταση διάσπασης η χ, κάτι που επιτρέπει και την εύρεση ενός "μεσαίου" subregion. Δεν είναι όμως καλή ιδέα για ένα region διαμερισμένο κατά μια διάσταση σε subregions να διασπαστεί και πάλι σε αυτήν την διάσταση. Τότε τα regions που θα προκύψουν από την διάσπαση δεν είναι αρκετά v cubical (4.2.1.2, 2.4.1).

Καλό θα ήταν δηλαδή η διάσπαση του παραδείγματος να γίνει με τις προηγούμενες διαδικασίες αλλά στον άξονα y και όχι στον χ, ώστε τα region που θα προκύψουν να έχουν ίδια αναλογία μήκους ακμών για τις δύο διαστάσεις. Αν γίνεται διάσπαση συνέχεια κατά την διάσταση χ τα μήκη ακμών σε αυτήν την διάσταση θα είναι πολύ μικρότερα από αυτά στην διάσταση y που δεν διασπάται.

Οπότε η λογική λέει να επιλέγονται με round robin pages οι διαστάσεις διαχωρισμού, όπως γίνεται (πλην των forced splits) στα point pages. Τότε όμως μπορεί να προκύψουν (αν ακολουθηθεί ο αλγόριθμος του Robinson) προβλήματα σχετικά με την τήρηση της απαίτησης οι δύο νέοι κόμβοι μετά την διάσπαση να μην είναι σε overflow.

Στο παράδειγμα όποια τιμή v στον άξονα y και να επιλεχθεί για θέση του υπερεπιπέδου διάσπασης, και τα 11 subregions θα τέμνονται από το υπερεπίπεδο και θα πρέπει να διασπαστούν (αφού όλα έχουνε τα ίδια miny maxy λόγω του τρόπου που είναι τοποθετημένα σε σειρά στον άξονα χ). Αφού θα διασπαστούν αναγκαστικά

όλα τα subregions, οι δύο νέοι κόμβοι που θα δημιουργηθούν θα έχουν από 11 subregions ο κάθε ένας και θα ξανά είναι σε κατάσταση overflow.

Αν έστω αυτό επιτρέπονταν τότε θα έπρεπε οι δύο νέοι κόμβοι να διασπαστούν αμέσως μετά την δημιουργία τους ώστε να τηρηθεί το invariant της δομής που θέλει συγκεκριμένο μέγιστο πλήθος subregions σε κάθε region. Επιλέγοντας σε αυτήν την άμεση διάσπαση την διάσταση χ (cyclic αλλαγή διαστάσεων διαχωρισμού) τότε προκύπτουν 4 τελικοί κόμβοι (διαισθητικά ο κάθε ένας έχει τις μισές πάνω ή κάτω υποπεριοχές) οι οποίοι δεν είναι σε overflow και η διαδικασία τελειώνει (αν και έχει ήδη προκαλέσει πλήθος διασπάσεων σε όλο το δένδρο στο ενδιάμεσο)

. Όταν οι διαστάσεις είναι n και όχι μόνο 2 ακολουθώντας την παραπάνω διαδικασία η δομή καταλήγει να έχει 2^n επιπλέον τελικούς νέους κόμβους (και επιπλέον και άλλους από forced split κτλ) και μεγάλο αριθμό διασπάσεων ουσιαστικά χωρίς λόγο. Αυτό συμβαίνει γιατί μόνο η αρχική διάσταση όπου ήταν ταξινομημένα τα regions μπορεί να έχει κάθετο επίπεδο διαχωρισμού που να μην τα τέμνει όλα.

Μπορεί κάποιος να πει βέβαια ότι η περίπτωση ταξινομημένων σε μια διάσταση regions είναι σπάνια λόγω του τρόπου που σπάνε cuclically τα point pages. Αφού τα point pages διασπώνται χωρίς να διατάσσονται σε έναν άξονα γιατί η διάσταση διάσπασης αλλάζει κυκλικά, αυτή η μη διάταξη μεταδίδεται σε ψηλότερα επίπεδα.

Πάντως θεωρητικά το πρόβλημα υπάρχει και όταν εμφανιστεί είναι σημαντικό. Αν πχ η ρίζα του δένδρου έχει subregions κατανεμημένα όπως στο παράδειγμα στην διάσταση χ, τότε όποια διάσταση και να ληφθεί πλην της χ, αυτή θα προκαλεί split σε όλα τα subregions, και τότε (δεν υπάρχει θεωρητική απόδειξη για αυτό βέβαια) είναι πιθανόν το δένδρο να συμπεριφέρεται κάνοντας συνέχεια split στην χ στα ψηλά επίπεδα (κάτι που κάνει μη ν-cubical τον διαχωρισμό)

Μια τακτική που δεν γλυτώνει από αυτήν την σπάνια χειρότερη περίπτωση αλλά βοηθάει σε ενδιάμεσες καταστάσεις είναι στο αρχικό split να προσπερνάμε την επόμενη διάσταση σε σειρά επιλογής αν αυτή διάσπαση προκαλεί διάσπαση σε όλα τα subregions (και κατά συνέπεια overflown νέους κόμβους). Αλλά είναι πιθανό όλες οι άλλες διαστάσεις να διασπούν το κάθε subregion, όπως στην παραπάνω "χειρότερη" περίπτωση.

Συνοψίζοντας η τακτική διέσπασε τους κόμβους και ας δώσουν overflow στους νέους κόμβους μπορεί να οδηγήσει σε άσχημα αποτελέσματα. Στο b-tree προφανώς κάτι τέτοιο δεν γίνεται και οι νέοι κόμβοι όχι μόνο δεν είναι σε overflow αλλά έχουν το πολύ το 50% των δεδομένων του αρχικού. Στην επόμενη ενότητα γίνεται η περιγραφή τρόπων επιλογής των d,v ώστε να αποφεύγεται το overflow στους νέους κόμβους.

* * *

Τα πρόβλημα του overflow στους νέους κόμβους περιορίζεται αν γίνει το split στο boundary (όριο) ενός από τα subregions της subregionlist

Αν έστω g είναι ο μέγιστος επιτρεπόμενος αριθμός subregions πριν το region page μπει σε overflow. Τότε στο overflow στο διασπώμενο region page θα υπάρχουν g+1 subregions.

Αν η επιλογή d,v γίνει έτσι ώστε το υπερεπίπεδο διαχωρισμού να πέφτει πάνω στο boundary B κάποιου subregion S1, αυτό θα είναι αναγκαστικά (λόγω γεωμετρίας) και boundary τουλάχιστον ενός ακόμα subregion S2, αυτού που συνορεύει με το S1 στο B.

Έτσι από από τα αρχικά subregions τουλάχιστον 2, τα S1, S2, δεν θα διασπαστούν. Τα υπόλοιπα στην χειρότερη θα διασπαστούν όλα δίνοντας δύο νέα regions για κάθε subregion που διασπάτε. Μιας και τα S1,S2 subregions είναι το ένα "δεξιά" και το

άλλο "αριστερά" του boundary, αφού συνορεύουν σε αυτό. το αριστερό νέο region page θα πάρει το πολύ

(g+1)-2+1=g regions

Τα αρχικά regions ήτανε g+1, αφαιρούνται τα S1 και S2, προστίθεται για κάθε ένα από όσα έμειναν ένα region, και προστίθεται στο τέλος και το S1. Το δεξί νέο region page παίρνει επίσης το πολύ g regions.

Άρα μιας και ένας κόμβος γίνεται overflown αν έχει g+1 subregions τουλάχιστον, κανείς από τους δύο νέους κόμβους που προκύπτουν από την διάσπαση δεν είναι σε overflow (συμπεριφορά όμοια με του b-tree). Παρόμοια είναι η διαδικασία που ακολουθείται στο hb-tree καθώς και εκεί η διάσπαση γίνεται στα boundaries μέσω του ενσωματωμένου kd δένδρου(βέβαια εκείνη η δομή είναι πιο εξελιγμένη και χρησιμοποιούνται holey regions αντί για regions που επειδή έχουν σχήμα παραλληλόγραμμου κάνουν αναγκαστική την χρήση forced splits).

Με βάση τα παραπάνω μια τακτική επιλογής των d,v είναι αυτή της επιλογής ενός από τα boundaries των subregions. Σίγουρα οι κόμβοι που θα δημιουργηθούν δεν θα είναι σε overflow ενώ δίνονται πολλές επιλογές v και d (μια για κάθε δυνατό boundary subregion, πλην αυτών των boundaries που πέφτουν πάνω στο boundary του όλου του κόμβου) ώστε να βρεθεί την καλύτερη με βάση τα κριτήρια του πόσο k cubical είναι οι δύο νέοι κόμβοι, πόσο ισομερής η κατανομή (utilization) και και πόσο μικρό είναι το πλήθος των forced splits. Τα 3 αυτά κριτήρια είναι εν μέρει αλληλένδετα μεταξύ τους.(4.2.1.7)

Για παράδειγμα ένας αλγόριθμος θα έπαιρνε όλα τα υποψήφια boundaries και θα τα βαθμολογούσε σε κάθε ένα από τα παραπάνω 3 κριτήρια. Στην συνέχεια θα ζύγιζε τις βαθμολογίες και θα έπαιρνε το boundary με την καλύτερη. Οι βαθμολογία θα

ζυγιζόταν κατάλληλα ανάλογα με το αν χρειάζονται λίγα splits ή καλύτερη κατανομή ή αποφυγή μη cubical regions. Στην επόμενη ενότητα παρουσιάζονται μερικές λεπτομέρειες σχετικές με την διάσπαση πάνω σε ένα boundary.

Η τακτική διάσπασης region pages που επιλέχτηκε να υλοποιηθεί ήταν αυτή. Να σημειωθεί ότι η χρήση της είναι εύκολη και σε ν διαστάσεις. Εκεί απλώς βλέπει κανείς τα boundaries τελείως αλγεβρικά, σαν μια λίστα από μέγιστο και ελάχιστο όριο σε κάθε διάσταση, χωρίς να ασχολείται με περίεργες γεωμετρικές ιδιότητες στον ν διάστατο χώρο όπως το πόσα region συνορεύουν με κάποιο άλλο.

4.2.1.7 Κριτήρια επιλογής υπερεπιπέδου

Σύμφωνα με την ενότητα 4.2.1.6 τα κριτήρια επιλογής υπερεπιπέδου είναι

-Πλήθος των forced splits (πρέπει να είναι χαμηλό)
-Utilization των 2 νέων κόμβων (πρέπει να τείνει στο 50 / 50
-Σχήμα των regions / χρήση όλων των διαστάσεων (πρέπει τα regions να είναι cubical)

Τα δύο πρώτα κριτήρια είναι σε κάποιο βαθμό αλληλένδετα. Οι διασπάσεις όχι μόνο απαιτούν ενέργειες I/O αλλά με τη σειρά τους οδηγούνε πάλι σε κακό utilization καθώς στο forced split τα παιδιά ενός κόμβου που διασπάτε δεν γίνεται να επιλέξουν κατάλληλη για αυτά d και ν.(4.2.1.5) Το κακό utilization αυξάνει το ύψος του δένδρου. Όσο πιο ψηλό είναι το δένδρο τόσο αυξάνεται το πλήθος I/O στην χρήση του, ενώ αυξάνεται και το πλήθος των πιθανών forced splits (αφού κάθε κόμβος θα έχει περισσότερους απογόνους). Επομένως γίνεται ένα είδος φαύλου κύκλου που τελικά αυξάνει τα I/O.

Επίσης το υπερεπίπεδο δεν πρέπει να δημιουργεί νέους κόμβους σε overflow (4.2.1.6). Αυτό δεν ισχύει ποτέ αν το υπερεπίπεδο είναι πάντα σε κάποιο boundary.

Αν επιτρέπονται και τα υπερεπίπεδα που δεν βρίσκονται πάνω σε boundaries τότε αποτελεί επιπρόσθετο κριτήριο.

4.2.1.8 Χειρισμός των boundaries

Έστω το κάθε υπερεπίπεδο διάσπασης περιγράφεται από τον συνδυασμό d,v όπου d η διάσταση στης οποίας τον άξονα είναι κάθετο το υπερεπίπεδο και v η θέση του στον άξονα αυτό.

Οι έννοιες αριστερό και δεξί του υπερεπιπέδου region βασίζονται στην αναπαράσταση του χ άξονα αλλά εξακολουθούν να έχουν την ίδια ονομασία και σε άλλες διαστάσεις. Τότε αριστερό region σημαίνει "όπου όλα τα σημεία του έχουν συντεταγμένη d μικρότερη του v" ενώ δεξί region "όπου όλα τα σημεία του έχουν συντεταγμένη d μεγαλύτερη ή ίση του v".

Με βάση τα παραπάνω υπάρχουν 3 τύποι subregions ανάλογα με το πως τέμνονται από το υπερεπίπεδο της τιμής v στην διάσταση δ. Τα αριστερά subregions, τα δεξιά subregions και αυτά που δεν είναι ούτε αριστερά ούτε δεξιά αλλά τέμνονται στο εσωτερικό τους (και όχι στο boundary) από το υπερεπίπεδο. Για τα τρίτα ισχύει ότι "υπάρχουν σημεία στο εσωτερικό τους με συντεταγμένες d άλλες φορές μικρότερη και άλλες μεγαλύτερη της v". Στην διάσπαση τα μη αριστερά ή δεξιά regions θα πρέπει σύμφωνα με τον Robinson να διασπαστούν δίνοντας ένα αριστερό και ένα δεξί subregion.

Προσοχή θέλει το ότι ένα subregion με αριστερό boundary πάνω στο v δεν διασπάται αλλά θεωρείται ότι βρίσκεται δεξιά του. Αυτό είναι κάπως non intuitive καθώς σημαίνει πως ένα region θεωρείται δεξιά ενός υπερεπιπέδου ενώ όμως το περιέχει στο εσωτερικό του και συγκεκριμένα στο αριστερό boundary. Αλλιώς όμως αν γινότανε split πάνω στο αριστερό boundary, το αριστερό subregion που θα προέκυπτε θα είχε mind=maxd που σύμφωνα με τα invariants της δομής είναι

άτοπο. Ένα region μπορεί να περιέχει κάποιο σημείο με P[d]=mind αλλά όχι ένα σημείο με P[d]=maxd. Όταν οι δεξιές πλευρές των εξισώσεων αυτών είναι ίσες το σημείο δεν θα μπορούσε να αποφασιστεί αν είναι εντός η εκτός του region.

Τα boundary like regions αυτού του τύπου μπορούν να αποφευχθούν ακόμα και αν ένα πλήθος από points βρίσκονται πάνω σε ένα τέτοιο boundary. Στην διάσπαση των point pages μπορούν να επιλεχθούν κατάλληλα regions κομμένα λίγο πιο μετά από το boundary ή σε άλλη διάσταση, και να μην γίνει το region boundary like.

4.2.2 Εισαγωγή

Η διαδικασία της εισαγωγής/insert ενός σημείου γίνεται με διάσχιση του δένδρου από τη ρίζα προς τα φύλα ακολουθώντας κάθε φορά την subregion του τρέχοντα κόμβου που περιέχει το σημείο. Σύμφωνα με το invariant αυτή η υποπεριοχή θα πρέπει να υπάρχει και να είναι μοναδική. Φτάνοντας σε ένα φύλλο τοποθετείται σε αυτό μια καταχώριση (Point, record) Στην συνέχεια γίνεται έλεγχος για το αν θα πρέπει να ξεκινήσει η διαδικασία διάσπασης του point page/φύλου).

Ο αλγόριθμος του insert ενός point (με βήματα όμοια με αυτά που δίνει ο Robinson) έχει ως εξής.

Εισαγωγή του σημείου P

1.Κάνε αναζήτηση στο δένδρο για να βρεις το ένα και μοναδικό point page όπου θα υπήρχε το P αν βρισκόταν ήδη στην δομή. Αν όντως το P υπάρχει ήδη τότε τερμάτισε με κάποιου είδους exception (η δομή του Robinson δεν υποστηρίζει περισσότερες της μίας αποθηκεύσεις του ιδίου σημείου)

2. Στο point page PP που εντοπίστηκε στο βήμα 1 εισήγαγε στο pointlist μια νέα εγγραφή για το P.
Αν το PP δεν είναι σε κατάσταση overflow μετά από αυτήν την εισαγωγή τερμάτισε αλλιώς πήγαινε στο βήμα 3

3. Αν το PP είναι η ρίζα δημιούργησε ένα νέο region page και θέσε το ως ρίζα και γονέα του PP.

4.Κάνε split το PP. (Αυτή η διαδικασία περιλαμβάνει την προσθήκη νέων καταχωρίσεων στον γονέα, διάσπαση του αν είναι σε overflow κ.τ.λ).

Εδώ η μόνη επιπλέον παρατήρηση έχει να κάνει με την ακολουθία εκτέλεσης ανάμεσα στην διαδικασία insert και την διαδικασία split του βήματος 4.

Αναλόγως πως θα υλοποιηθεί το πρόγραμμα μπορεί τυπικά να τερματίζει πρώτη είτε η συνάρτηση εισαγωγής είτε η συνάρτηση διάσπασης. Δεν έχει κανένα νόημα να παρουσιασθούν οι εναλλακτικές περιπτώσεις εδώ. Στο κεφάλαιο με την περιγραφή του κώδικα δείχνω πως έγινε η υλοποίηση αυτού του σημείου στην εργασία αυτή

4.2.3 Διαγραφή

. Ο αλγόριθμος διαγραφής δεν υλοποιήθηκε καθώς δεν θεωρήθηκε από τον επιβλέποντα καθηγητή ότι ήταν από τα ζητούμενα της εργασίας. Επίσης η περιγραφή που δίνει στο paper του ο Robinson δεν είναι αρκετή από μόνη της για να ορίσει πλήρως την διαδικασία. Ο Robinson δεν είχε υλοποιήσει την διαγραφή και ορισμένα από τα βήματα του αλγορίθμου που δίνει είναι από μόνα τους ξεχωριστά προβλήματα και συγκεκριμένα προβλήματα υπολογιστικής γεωμετρίας.

Σε γενικές γραμμές στην διαγραφή αφαιρούνται σημεία από point pages και αν τα σημεία που απομένουν είναι μικρότερα από κάποιο όριο (σελίδες σε τέτοια κατάσταση ονομάζονται under-full) τότε πρέπει να γίνει μια διαδικασία αντίστροφη

της διάσπασης. Ο Robinson δεν εξηγεί αν αυτή θα εκτελείται και όταν προκύπτουν point pages με λίγα ή καθόλου σημεία μετά από ένα forced split. Θα μπορούσαν αυτά τα point pages να παραμένουν under-full μέχρι κάποια διαγραφή region να τα συνενώσει ή να συμπεριλάβει η διαδικασία του forced split βήματα συνένωσης περιοχών ώστε να μην υπάρχουν under-full περιοχές στην δομή.

Το κρίσιμο σημείο είναι αυτό της συνένωσης. Η συνένωση είναι πιο δύσκολη από την διάσπαση καθώς εμπεριέχει αρκετά πολύπλοκα προβλήματα υπολογιστικής γεωμετρίας. Ο Robinson λέει πως όταν ένας κόμβος Κ που αντιπροσωπεύει το Region R είναι under-full, τότε από τα subregions του γονέα ΓΚ του Κ θα πρέπει να βρεθούν ένα σύνολο GR1, GR2...GRj ώστε το R με τα GRi να έχουνε ένωση πάλι ένα σύνολο σημείων που γεωμετρικά αποτελεί region. Έστω αυτό ονομάζεται RN.

Στην συνέχεια τα subregions όλων των GRi και του R θα πρέπει να μοιραστούν σε νέα subregions SRN που προκύπτουν από την διάσπαση του RN με τρόπο ώστε κανένα από τα SRN να μην είναι underfull. Δεν δίνεται κάποιος αλγόριθμος για αυτό η κάποια απόδειξη ότι είναι θεωρητικά δυνατό.

Και αν δινόταν όμως εξακολουθεί να υπάρχει το γεωμετρικό πρόβλημα της εύρεσης του RN. Αν και με το μάτι (και στις 2 διαστάσεις) είναι εύκολο να βρεθεί ένα παραλληλόγραμμο που να περιέχει το R υπολογιστικά αυτό έχει δυσκολίες. Ειδικά όταν οι διαστάσεις είναι περισσότερες των 3 όπου οι έννοιες διπλανά regions γίνονται δυσνόητες και ο αλγόριθμος πρέπει να στηρίζεται καθαρά σε άλγεβρα και όχι γεωμετρία.

Το πρόβλημα είναι η κάλυψη κάποιου χώρου με ορισμένα δοθέντα γεωμετρικά χωρία και σε brute force approach έχει εκθετικό χρόνο εκτέλεσης. Μια πιο γρήγορη λύση είναι η τήρηση κάποιας λίστας ιστορικού διασπάσεων ώστε κάθε κόμβος να μπορεί να επανασυνδεθεί με τους παλαιότερους από όπου προέκυψε, σε ένα είδος undo. (Με την υλοποίηση του εσωτερικού των κόμβων με kd δένδρα αντί λίστες από regions αυτό το ιστορικό υπάρχει ήδη). Αυτή η λύση μπορεί να μην είναι βέλτιστη

καθώς μπορεί να οδηγήσει σε γρήγορη επανασύνδεση όλων των subregions κάτι που ουσιαστικά σημαίνει και πλήρες reorganization του υποδένδρου(κάτι που δεν είναι επιθυμητό, αν ήταν να επιτρέπεται κάθε φορά full reorganization θα μπορούσε να χρησιμοποιηθεί και το στατικό kd tree)

4.2.4 Range Query

Οι αναζητήσεις στο kdb tree γίνονται μέσω range queries που όπως αναφέρθηκε (3.2.1) μπορούν να έχουν τις ειδικές μορφές των partial queries ή point queries. Για την αναζήτηση δίνεται ως όρισμα ένα region RQ του χώρου και επιστρέφονται όλα τα σημεία που έχουν εισηχθεί στην δομή και που βρίσκονται γεωμετρικά στο εσωτερικό του RQ.

Το RQ δεν είναι ακριβώς region με την έννοια αυτών που διαμέριζαν τον χώρο στους κόμβους. Παραμένει ν διάστατο παραλληλόγραμμο με πιθανόν απείρου μήκους ακμές, αλλά περιέχει και το δεξί και το αριστερό boundary σε κάθε διάσταση αντί για μόνο το αριστερό. Δηλαδή τα διαστήματα κάθε διάστασης που δίνουν με καρτεσιανό γινόμενο το RQ είναι της μορφής [,] (κλειστό κλειστό)

Αυτό συμβαίνει ώστε να μπορούν να οριστούν τα point queries. Θα ήταν άτοπο το δεξί και το αριστερό boundary να ήταν ίσα και το ένα να περιέχεται στο region ενώ το άλλο όχι. Ο αλγόριθμος δεν έχει κάποια δυσκολία στην κατανόηση ή την υλοποίηση εκτός ίσως από το ενδιάμεσο βήμα του collision detection σε ν διαστάσεις. Δεν περιγράφεται αναλυτικά στο paper. Χωρίζεται εδώ σε 2 υπορουτίνες

SUBROUTINE REGIONPAGEQUERY(RP,RQ)
Για κάθε subregion sri στην subregionlist του RP το οποίο έχει κοινά σημεία στον χώρο με το RQ κάλεσε
REGIONPAGEQUERY(RP2,RQ)αν το sri δείχνει σε region page RP2
ή
POINTPAGEQUERY(PP,RQ)αν το sri δείχνει σε point page PP
END

SUBROUTINE POINTPAGEQUERY(PP,RQ)
Τοποθέτησε στην global λίστα αποτελεσμάτων όσα σημεία του pointlist του PP βρίσκονται εντός του RQ
END

SUBROUTINE QUERY(RQ)
Κάλεσε το
POINTPAGEQUERY(ROOT,RQ) αν ή ρίζα ROOT είναι point page
ή
REGIONPAGEQUERY(ROOT,RQ) αν ή ρίζα ROOT είναι region page
END

Για να ξεκινήσει η διαδικασία αναζήτησης επιλέγεται το RQ και καλείται το Query(RQ).

Η περιγραφή εδώ είναι λίγο διαφορετική από την υλοποίηση σε c# όπου αντί για global λίστα αποτελεσμάτων τα points αποτελούν μια λίστα επιστρεφόμενη τιμή των συναρτήσεων που υλοποιούν τις παραπάνω υπορουτίνες.

Αυτό που γίνεται είναι ένα Depth First Search like traversal των κόμβων του δένδρου που τέμνονται στον χώρο με το RQ. Όταν φτάσει ο έλεγχος στα point pages όλα τα σημεία εντός του RQ τοποθετούνται σε μια λίστα αποτελεσμάτων.

Να σημειωθεί ότι όπως είναι φανερό υπάρχει backtracking και δεν ακολουθείται μια μοναδική διαδρομή από την ρίζα ως τα φύλα (πχ στα point queries).

Στην περίπτωση ενός point query σε κάθε επίπεδο κόμβων του δένδρου υπάρχει ένα και μόνο ένα καταχωρημένο subregion που να περιέχει το σημείο P που αναζητείται. Αυτή άλλωστε είναι και η βασική αρχή διαμερισμού του χώρου στο kdb tree. Έτσι δεν χρειάζεται back tracking για τα point queries που είναι σημαντικό γιατί αυτά είναι συχνά (το πρώτο βήμα των αλγορίθμων εισαγωγής και διαγραφής περιέχει ένα point query).

4.2.4.1 Collision detection σε ν διαστάσεις

Η εύρεση του κάθε πιθανού subregion που μπορεί να έχει κοινά σημεία με το δοθέν RQ είναι ένα πρόβλημα που ονομάζεται και collision detection και όταν οι διαστάσεις είναι ν ο αλγόριθμος που το λύνει δεν είναι trivial αλγόριθμος.

Διευκολύνει πάντως την επίλυση το γεγονός ότι τα regions είναι παραλληλόγραμμα καθώς με μαθηματική επαγωγή αποδεικνύεται ότι αν 2 ν διάστατα παραλληλόγραμμα με πλευρές τους κάθετες στους βασικούς άξονες, έχουν κοινό σημείο, θα πρέπει να έχουν κοινό σημείο και σε όλες τις προβολές τους στους βασικούς άξονες.

Αν δεν ίσχυε αυτό πχ αν τα παραλληλόγραμμα μπορούσαν να έχουν περιστραφεί από κάποιον τυχαίο άξονα αναγκαστικά θα έπρεπε να ελεγχθεί αν τέμνονται μια προς μια η ακμές τους, και σε ν διαστάσεις υπάρχουνε 2^ν κορυφές και ανάλογα μεγάλο πλήθος ακμών οπότε ο αλγόριθμος θα ήτανε δύσκολος και αργός.

Αν τα παραλληλόγραμμα έχουν ν διάστατες πλευρές κάθετες στους βασικούς άξονες όμως, τότε υπάρχει πιο απλός τρόπος υπολογισμού που βασίζεται στην παρατήρηση

για τις προβολές.

Έστω τα R1 και R2 και R παραλληλόγραμμα ν διαστάσεων με πλευρές κάθετες στους βασικούς άξονες. Για οποιαδήποτε από αυτά αν υπάρχει εντός τους σημείο P με συντεταγμένη στην πρώτη διάσταση α1 τότε για κάθε συνδυασμό δυνατών τιμών συντεταγμένων στις άλλες διαστάσεις θα υπάρχει και σημείο P' όπου στις άλλες διαστάσεις οι τιμές έχουν αυτόν τον συνδυασμό και στην πρώτη την τιμή α1.

Ομοίως ισχύει και για κάθε συνδυασμό τιμών α1,α2 στις πρώτες δύο διαστάσεις, α1,α2,α3 στις πρώτες 3 κτλ. Αυτό συμβαίνει γιατί το παραλληλόγραμμο εκτείνεται σε όλες τις διαστάσεις με βάση τα διαστήματα, πιο συγκεκριμένα γιατί είναι το καρτεσιανό γινόμενο των διαστημάτων που το ορίζουν σε κάθε διάσταση και κάθε υποπλειάδα μπορεί να συμπληρωθεί κατάλληλα ώστε να προκύψει πλειάδα του παραλληλογράμμου.

Θα γίνει μια προσπάθεια με βάση τα παραπάνω να δειχθεί ότι τα R1 και R2 έχουν κοινό σημείο αν και μόνο αν τα διαστήματα R1Iδ και R2Iδ που περιγράφουν τα όρια τους σε κάθε διάσταση δ, έχουν για κάθε δ και κοινό σημείο. (τα διαστήματα αυτά είναι οι προβολές των Ri στους άξονες)

Αν ισχύει ότι τα R1I1 και R2I1 έχουν κοινό σημείο αυτό δεν σημαίνει ότι τα R1 και R2 έχουν και αυτά. Σε δύο διαστάσεις για παράδειγμα μπορεί το R1 να βρίσκεται πιο ψηλά από το R2 και παρόλο που και τα δύο έχουνε κοινά σημεία στην προβολή τους στον άξονα χ να μην τέμνονται λόγω του ύψους τους στον y.

Όμως επειδή κάθε παραλληλόγραμμο αποτελεί το καρτεσιανό γινόμενο I1 x I2 x ... Iν των Iδ για κάθε σημείο K1 κοινό στα R1I1 και R2I1 κάθε δυνατή τιμή Δ1 Δ2 των R1I2 και R2I2 μπορεί να συνδυαστεί με το K1 δίνοντας τις δυάδες (K1,Δ1) και (K1,Δ2) που αποτελούν υποπλειάδες πλειάδων του R1 και του R2. Αν τώρα τα R1I2

και R2I2 έχουνε και αυτά κοινό σημείο τότε θέτοντας Δ1=Δ2=K2 για αυτό σχηματίζεται μια υποπλειάδα (K1,K2) που υπάρχει και στο R1 και στο R2.

Συνεχίζοντας με επαγωγή μέχρι τις ν διαστάσεις προκύπτει μια πλειάδα (K1,K2,...,Kν) η οποία τότε αποτελεί σημείο και του R1 και του R2. Άρα αποδείχτηκε ότι αν τα R1Iδ και R2Iδ έχουνε κοινό σημείο για κάθε δ τότε και τα R1 και R2 έχουνε κοινό σημείο.

Αντίστροφα αν τα R1 και R2 έχουνε κοινό σημείο τότε παίρνοντας τις συντεταγμένες του προκύπτει για κάθε διάσταση και ένα κοινό σημείο των Iδ, κάτι που ολοκληρώνει την απόδειξη. Με βάση τα παραπάνω ο αλγόριθμος collision detection σε ν διαστάσεις ανάγεται σε λογικό AND ν collision detections σε 1 διάσταση (πάντα για αυτού του συγκεκριμένου τύπου τα παραλληλόγραμμα, με πλευρές κάθετες στους βασικούς άξονες)

(Πηγές κεφαλαίου 4 [Robinson 81])

5. ΣΧΕΔΙΑΣΜΟΣ

Η διπλωματική εργασία περιελάμβανε την ανάπτυξη κώδικα που να υλοποιεί το kdb tree όπως αυτό παρουσιάστηκε από τον Robinson στο [Robinson 81]. Εκεί δινόταν ήδη σε ψευδοκώδικα τα γενικά βήματα των αλγόριθμων των βασικών λειτουργιών εισαγωγής διαγραφής διάσπασης και αναζήτησης.

Ορισμένα βήματα ωστόσο δεν ήταν στοιχειώδη**(1)** (πχ στην διαγραφή, εύρεση ενός συνόλου regions που η ένωση τους να είναι και πάλι region). Άλλα άφηναν περισσότερες από μια επιλογές όσον αφορά το πως θα υλοποιηθούν **(2)**(πχ η διάσπαση των region pages)

Ο σχεδιασμός λογισμικού υλοποίησης μιας αυτού του τύπου δομής δεδομένων έχει ορισμένα ιδιαίτερα χαρακτηριστικά . Η υλοποίηση imperative αλγορίθμων που ενεργούν πάνω σε δενδρική δομή δεν είναι μια διαδικασία που προσεγγίζεται ικανοποιητικά από τις ευρέως διαδεδομένες αντικειμενοστραφείς μεθόδους σχεδιασμού. Το object oriented model δεν μπορεί να βοηθήσει ιδιαίτερα γιατί ο κώδικας που ζητείται είναι από την φύση του διατακτικός.

Το OOM ταιριάζει για εφαρμογές όπου υπάρχει ανάγκη συλλογής απαιτήσεων από μη ειδικούς, επαναχρησιμοποίησης και ολοκλήρωσης δομημένης πληροφορίας και διαδικασιών, επεκτασιμότητας για τις περιπτώσεις μελλοντικών αλλαγών, και ροής που βασίζεται στην ανταλλαγή μηνυμάτων που προκύπτει από την μοντελοποίηση οντοτήτων και διαδικασιών του πραγματικού κόσμου.

Αντίθετα στην υλοποίηση του kdb tree οι απαιτήσεις είναι γνωστές (δίνονται τα βασικά βήματα κάθε αλγορίθμου), τα τμήματα της δομής έχουν συγκεκριμένους ρόλους και δεν πρόκειται να ολοκληρωθούν σαν ανεξάρτητες μονάδες με εξωτερικά αντικείμενα, ενώ η ροή βασίζεται στην τήρηση μαθηματικών ιδιοτήτων και όχι σε μοντελοποίηση επικοινωνίας οντοτήτων του πραγματικού κόσμου.

Σχεδιαστικά θα περίμενε κανείς ότι με βάση τα παραπάνω η ανάπτυξη θα ήταν εύκολη, ειδικά όταν δίνεται ήδη μεγάλο μέρος των αλγορίθμων. Μιας και οι απαιτήσεις είναι λίγο πολύ γνωστές σε τι ακριβώς χρησιμεύει ο σχεδιασμός;

Κατ αρχάς σημειώνεται πως οι αλγόριθμοι του paper παρουσιάζουν τα χαρακτηριστικά (1) και (2) που σημαίνει πως σε μεγάλο βαθμό καλείται ο προγραμματιστής να τους συμπληρώσει**(3)**, κάτι που δεν απαιτεί τόσο γνώσεις σχεδιασμού λογισμικού όσο γνώσεις στην κατασκευή αλγορίθμων και στα μαθηματικά (ειδικότερα πολλά από τα σημεία που δεν δίνει τόση προσοχή το paper έχουν να κάνουν με υπολογιστική γεωμετρία)

Από εκεί και πέρα αν και δεν χρειάζεται ανάλυση απαιτήσεων και σχεδιασμός πάνω σε αυτές, η αποσφαλμάτωση γίνεται πιο δύσκολη, για λόγους που αναλύονται παρακάτω, και ο σχεδιασμός της αποκτά μεγάλη σημασία.

Ο τελικός κώδικας με βάση τους δοθέντες αλγορίθμους και μια καθαρά διατακτική υλοποίηση θα είναι πολύπλοκος και εκτενής. Θα χαρακτηρίζεται από πλήθος βρόγχων και διακλαδώσεων if (βρόγχοι επιλογής subregion, βρόγχοι αλλαγής συντεταγμένης, βρόγχοι εγγραφής αποτελεσμάτων, έλεγχος τύπου page region ή point, έλεγχος ένταξης σημείου σε region κτλ)

Η επιπλέον δυσκολία στην αποσφαλμάτωση προκύπτει από το ότι ενώ είναι γνωστό τι κάνει το κάθε βήμα, η υλοποίηση είναι τόσο μεγάλη σε έκταση και πολυπλοκότητα γραμμών κώδικα, ώστε αναπόφευκτα να "ξεφεύγουν" τυχαία σφάλματα απροσεξίας από τον προγραμματιστή τα οποία όμως εμποδίζουν την σωστή λειτουργία ολόκληρου του προγράμματος και οι επιπτώσεις τους δεν είναι τοπικά περιορισμένες. Παράδειγμα τέτοιου λάθους είναι η χρήση του πίνακα "max" αντί του πίνακα "min" σε κάποιο σημείο του κώδικα. Αυτό δεν θα δώσει κάποιο άμεσο compile η run time error και θα είναι πολύ δύσκολο να εντοπιστεί σε αδόμητο κώδικα που αποτελείται από ένα πλήθος εντολών επανάληψης και διακλάδωσης.

Όταν ο προγραμματιστής προσπαθήσει να εντοπίσει σφάλματα έρχεται αντιμέτωπος με τα προβλήματα που παρουσιάζονται στην ενότητα 5.5 (Debugging) Επίσης τα λάθη απροσεξίας ανακατεύονται με τα λογικά σφάλματα που γίνονται κατά την υλοποίηση των διαδικασιών τις δομής για τις οποίες ισχύει το (3).Ο έλεγχος της ορθότητας των υλοποιήσεων αυτών γίνεται πιο δύσκολος αφού τα τυχαία σφάλματα εμφανίζονται στην θέση των λογικών μπερδεύοντας τον προγραμματιστή.

Τα παραπάνω μπορούν να αυξήσουν κατά πολύ την χρονική διάρκεια της αποσφαλμάτωσης . Μάλιστα εφόσον οι αλγόριθμοι ήδη δίνονται, η αποσφαλμάτωση αποτελεί μάλλον το δυσκολότερο και πιο χρονοβόρο στάδιο της ανάπτυξης.

Πρέπει επομένως να ληφθούν βήματα για την διευκόλυνση αυτής της διαδικασίας, Υπάρχουν 2 κύριες κατευθύνσεις:

(α) Απλοποίηση του κώδικα με χρήση frameworks και άλλων τεχνολογιών λογισμικού

-Χρήση των μηχανισμών πολυμορφισμού των ΟΟ γλωσσών
-Εκτέλεση των ενεργειών τύπου data selection and handling μέσω LINQ
-Χρήση της XSLT για ενέργειες τύπου μετασχηματισμού δεδομένων
-Διαχωρισμός των “crosscutting concerns” μέσω AOP

(β) Σχεδιασμός στρατηγικής ελέγχων

-Design by contract

Στο (α) γίνεται προσπάθεια να γραφεί το πρόγραμμα με λιγότερες γραμμές κώδικα, όχι μόνο γιατί αυτό επιφέρει ένα productivity gain (δεν είναι αμελητέο αλλά από την άλλη ούτε και ιδιαίτερα μεγάλο), αλλά και γιατί οι επιπλέον γραμμές δίνουν περισσότερες ευκαιρίες για τυχαία σφάλματα. Ταυτόχρονα μειώνεται η τυχόν επανάληψη των ίδιων κομματιών κώδικα σε πολλά σημεία κάτι που κάνει πιο

γρήγορη την τροποποίηση του, και έτσι διευκολύνει την διόρθωση λογικών σφαλμάτων

Το κύριο κέρδος από τις λιγότερες γραμμές επομένως δεν είναι το άμεσο κέρδος της λιγότερης πληκτρολόγησης, αλλά το έμμεσο κέρδος από το ότι ο ο κώδικας γίνεται πιο manageable και τα λάθη είναι λιγότερα και διορθώνονται πιο εύκολα.

Στο (β) γίνεται προσπάθεια να οργανωθεί η διαδικασία ελέγχου εντοπισμού και χαρακτηρισμού/προσδιορισμού των λαθών.

Τέλος υπάρχουνε σημεία για τα οποία ακολουθείται αναγκαστικά σχεδιασμός που δεν συμβαδίζει με τα παραπάνω , αυτά αναφέρονται στην ενότητα 5.2

5.1 Πολυμορφισμός

Ο ψευδοκώδικας που υπάρχει στο paper βασίζεται σε μια αρχιτεκτονική όπου κάποια κεντρική ρουτίνα ελέγχου τρέχει τους αλγορίθμους καλώντας υπορουτίνες, και φορτώνοντας τις δομές δεδομένων (στην περίπτωση αυτή τους κόμβους). Ομοίως και πολλές από τις υπορουτίνες ακολουθούν την ίδια διαδικασία με την κεντρική.

Αυτή η φόρτωση δεδομένων και επιλογή αλγορίθμου έχει την μορφή

Φόρτωσε τον κόμβο που δείχνει ο τρέχων δέκτης
IF είναι point page
THEN [...]
ELSE IF είναι region page
THEN [...]

Σε ένα τέτοιο σημείο φορτώνεται ο επόμενος κόμβος και η ροή ελέγχου χωρίζεται σε δύο υπορροές ανάλογα με τον τύπο κόμβου που φορτώθηκε, region page ή point page.

Θα ήταν μεγάλο σχεδιαστικό λάθος ο κώδικας ανάμεσα στις αγκύλες να μην τοποθετηθεί σε κάποια υπορουτίνα αλλά να γραφεί απευθείας σε εκείνη την θέση. Τότε το πρόγραμμα γρήγορα θα αποκτούσε μορφή spaggheti code. Οι εντολές if των αλγορίθμων θα εμφωλεύονταν μέσα στις εντολές if που τους επιλέγουν. Τα σημεία που θα ήταν κοινά σε όλες τις διακλαδώσεις θα έπρεπε να αντιγραφούν με copy paste ή αλλιώς η ροή να μπαίνει και να βγαίνει από την εμβέλεια εντολών if με περίπλοκο τρόπο. Γενικά το manageability του κώδικα μοιάζει να μειώνεται σχεδόν συνδυαστικά καθώς επιπλέον εντολές διακλαδώσεις προστίθενται σε μια ρουτίνα..

Ακόμα καλύτερα από την τμηματοποίηση σε υπορουτίνες μπορεί να γίνει χρήση των μηχανισμών πολυμορφισμού των ΟΟ γλωσσών. Εκεί αντί για έλεγχος τύπων με if χρησιμοποιείτε το dynamic dispatch. Αυτός είναι ο μηχανισμός όπου καλώντας μια συνάρτηση σε αναφορά υπερκλάσης, κατά την εκτέλεση η συνάρτηση που καλείται εξαρτάται από το ποια υποκλάση δείχνει εκείνη την ώρα η αναφορά.

Με τον πολυμορφισμό, αντί για ελέγχους τύπων γίνονται κλήσεις συναρτήσεων στην υπερκλάση Node, συναρτήσεις που υλοποιούνται στις υποκλάσεις RegionNode και PointNode. Ενώ οι κοινές λειτουργίες τοποθετούνται σε βοηθητικές συναρτήσεις της κλάσης Node.

Ο κώδικας γίνεται πιο κατανοητός γιατί αποκτά μορφή αλληλουχίας βημάτων αντί πλήθους εμφωλευμένων και επαναλαμβανόμενων διακλαδώσεων Για να λειτουργήσουν τα παραπάνω θα πρέπει το πρόγραμμα να περάσει σε ΟΟ μοντέλο και οι δομές δεδομένων να γίνουν κλάσεις. Στην εργασία προέκυψαν κλάσεις όπως Node, RegionNode, PointNode, Region, Point.

Ο διαχωρισμός του κώδικα και η μείωση των διακλαδώσεων έχει σημασία στην διαδικασία του debugging. Σε ένα πλήθος διακλαδώσεων μέσα σε μια κεντρική ρουτίνα ελέγχου είναι πιο δύσκολο να εντοπισθεί το λάθος και να εφαρμοστούν τεχνικές design by contract, από ότι όταν γίνεται χρήση ξεχωριστών συναρτήσεων και πολυμορφισμού. 5.5.2)

5.2 Χρήση Στοίβας

Με βάση το OOM θα ήταν λογικό να υπάρχει κάποια συσχέτιση ανάμεσα σε κάθε κόμβο υιό Κ με τον κόμβο γονέα Γ.

Σε επίπεδο υλοποίησης αυτή θα εκφραζόταν με την ύπαρξη στον Κ ενός δείκτη/αναφοράς προς τον Γ (υπάρχουνε βέβαια και πιο αφηρημένοι τρόποι να εκφραστεί η συσχέτιση). Μέσω του δείκτη αυτής της συσχέτισης ο Κ θα έστελνε μηνύματα στον Γ (πχ για την προσθήκη κάποιου region).

Αυτό όμως δεν μπορεί να γίνει επειδή ο αλγόριθμος του kdb προσπαθεί να ελαχιστοποιήσει τις ενέργειες I/O κατά την χρήση δευτερεύουσας μνήμης.

Όταν ένας εσωτερικός κόμβος RP διασπάται, ένα υποσύνολο SR από τα subregions στην subregionlist του RP δεν διασπώνται με forced split. (όλα όσα δεν τέμνονται από το υπερεπίπεδο διαχωρισμού) Με την διάσπαση του RP τα μέλη του συνόλου SR αλλάζουν γονέα. Επειδή και το κάθε μέλος του SR αντιστοιχεί σε ένα page δίσκου, το page αυτό θα πρέπει να φορτωθεί και να εγγραφεί σε αυτό νέα τιμή του δείκτη προς τον γονέα αφού ο παλιός έχει διασπαστεί. Αυτό σημαίνει μια επιπλέον ανάγνωση και εγγραφή I/O για κάθε μέλος του SR.

Αυτές οι ενέργειες I/O είναι άσκοπες καθώς αν δεν κρατούνταν για κάθε κόμβο δείκτης προς τον γονέα τότε θα μπορούσαν οι καταχωρίσεις του SR να αντιγραφούν στα δύο νέα region pages που προκύπτουν από την διάσπαση του RP χωρίς να χρειάζεται να μεταβληθούν οι σελίδες των SR.

Στο πρόγραμμα που έγραψα δεν υπήρχε απαίτηση να γίνει ούτε έγινε τελικά κάποια εγγραφή δεδομένων στον δίσκο. Θεωρήθηκε ότι αυτό που έχει σημασία είναι να αποδείξει το πρόγραμμα την δυνατότητα υλοποίησης της δομής, και να μπορεί να καταγράφει κάθε υποτιθέμενη ενέργεια I/O (έστω και αν αυτή δεν γίνεται πραγματικά) ώστε να συλλεχθούν στατιστικές μετρήσεις.

Κάθε φορά που το πρόγραμμα θα εκτελούσε κανονικά I/O, παραλείπει την εκτέλεση, και αντ αυτού σημειώνει το γεγονός αυτό σε κάποιο log.

Αν και δεν εκτελούνται πραγματικά I/O το πρόγραμμα, δεν έγινε χρήση δείκτη προς τον γονέα σε κάθε κόμβο (κάτι που θα απλοποιούσε και θα έκανε πιο object oriented τον κώδικα) για να μπορεί η τροποποίηση ώστε να είναι δυνατή η χρήση I/O να γίνει δυνατή χωρίς μεγάλες αλλαγές.

Κάθε κόμβος K μπορεί να γνωρίζει ποιος είναι ο γονέας του όταν δεν κρατάει πληροφορία για χάρη στην χρήση ενός global Stack. Σε αυτό αφαιρούνται και τοποθετούνται κόμβοι καθώς αντίστοιχα ανεβαίνει ή κατεβαίνει ο αλγόριθμος στο δένδρο.

Η χρήση στοίβας κάνει τον κώδικα πιο imperative με την σειρά των εντολών να αποκτά πιο μεγάλη σημασία. Κάθε κλήση συνάρτησης πρέπει να γίνεται εν μέσω εντολών για την διατήρηση της στοίβας σε μορφή Push(); CallFunction(); Pop();

Αυτό δεν είναι modular (δομημένο) και μπορεί κάποιος να ξεχάσει κάποιο push ή pop προκαλώντας σοβαρά λάθη. Ειδικά αν στον πηγαίο κώδικα γίνονται πολλές τέτοιες κλήσεις, και αν οι τροποποιήσεις του κώδικα είναι συχνές, τότε είναι ιδιαίτερα εύκολο να συμβεί κάτι λάθος, να μην τοποθετηθούν οι εντολές με την σειρά ή να παραλειφθεί κάποια. Για να μειωθεί αυτή η δυσκολία χρησιμοποιήθηκε μια δομή βασισμένη στα delegates και παρόμοια με τις τεχνικές του AOP

5.3 Χειρισμός δεδομένων

Στα περισσότερα βήματα των αλγορίθμων υπάρχουν μεν αρκετές μαθηματικές πράξεις και υπολογισμοί, αλλά σε μεγάλο βαθμό κυριαρχούν εντολές όπου από μια συλλογή δεδομένων επιλέγονται μόνο όσα δεδομένα υπηρετούν κάθε φορά κάποια κριτήρια. (Αν προσέξει κανείς τα πιο πολλά βήματα περιέχουν φράσεις όπως "για κάθε ", "για όσα regions ισχύει" κτλ) Αυτό λέγεται και data handling code.

Η μεγάλη συχνότητα σε εντολές αυτού του τύπου επηρέασε την επιλογή της γλώσσας προγραμματισμού που χρησιμοποίησε η υλοποίηση. Προτιμήθηκε η γλώσσα C#, από για παράδειγμα την Java, γιατί έχει ειδικές προγραμματιστικές δομές (LINQ) που διευκολύνουν αυτού του είδους την επεξεργασία. Το LINQ μπορεί να μειώσει το πλήθος εντολών στις υλοποιήσεις του ιδίου αλγορίθμου ανάμεσα πχ στην Java και στην C#.

Το πλήθος εντολών έχει σημασία όχι μόνο για το productivity του προγραμματιστή αλλά και ειδικότερα γιατί σε ένα πρόγραμμα που το debugging είναι δύσκολο, κάθε επιπλέον εντολή προσθέτει και ένα σημείο όπου μπορεί να γίνει κάποιο hard to find σφάλμα κώδικα.

Επίσης όταν μια υλοποίηση Α κάνει το ίδιο με την Β σε λιγότερες γραμμές κώδικα τότε διορθώσεις λαθών ή άλλες τροποποιήσεις θα χρειαστούν περισσότερο χρόνο στην Β από ότι στην Α, γιατί στην Β επηρεάζονται περισσότερες γραμμές κώδικα σε κάθε τροποποίηση του προγράμματος. Στην Java ο κώδικας data handling χαρακτηρίζεται από συνεχή επανάληψη του ακόλουθου κομματιού ψευδοκώδικα:

```
ItemClass a;
Collection<ItemClass> collection;
ResultCollection<ItemClassMemberType> resultcollection;

for(i in collection)
{
if i.satisfiesConditions()
resultcollection.add(i.somemember)
}
resultcollection.sort(new SpecialComparerObject());

class specialComparerObbject implements ComparerInterface
{
[...]
public int Compare(Object arg1, Object arg2){
...
ItemClassMemberType i1=(ItemClassMemberType) arg1;
ItemClassMemberType i1=(ItemClassMemberType) arg1;
}
}
```

Ο κώδικας φαίνεται απλός αλλά συνδυαστικά μπορεί να γίνει και δύσκολος στην διαχείριση. Η ταξινόμηση του αποτελέσματος έχει ως ενδιάμεσο βήμα την χρήση κάποιου αντικειμένου Α που υλοποιεί ένα interface σύγκρισης ComparerInterface. Στο ComparerInterface ορίζεται μια ειδική βοηθητική μέθοδος σύγκρισης Compare που χρησιμοποιεί εσωτερικά η μέθοδος ταξινόμησης (sort). Η sort δέχεται ως όρισμα μια αναφορά τύπου ComparerInterface σε αντικείμενο Α (εδώ το new SpecialComparerObject()) που υλοποιεί το ComparerInterface. Στην συνέχεια για να συγκριθούν τα στοιχεία η sort καλεί την Compare του CI όπως την έχει υλοποιήσει το Α, περνώντας κάθε φορά τα δύο επόμενα στοιχεία που είναι να συγκρίνει ως όρισμα. Η υλοποίηση της Compare στο Α ή θα κάνει cast από ορίσματα object όπως στο παράδειγμα ή θα είναι generic συνάρτηση.

Ο κώδικας αυτός είναι μακρύς, boilerplate, και πρέπει να επαναλαμβάνεται συχνά. Επίσης περιπλέκεται παραπάνω από το static typing Δεν γίνεται κάποιο compile time inference των τύπων που χρησιμοποιούνται, και έτσι ο προγραμματιστής πρέπει να υπολογίσει από μόνος του το τι τύπο θα δώσει στην κάθε μεταβλητή. Αν και τα ονόματα που δόθηκαν στο παράδειγμα είναι απλά ενδεικτικά και όχι αυτά ακριβώς που ορίζει η Java, φαίνεται πως δεν είναι αποδοτικό να αναφέρεται κάθε φορά αν γίνεται χρήση ComparerInterface ή SpecialComparerObject ή των εκάστοτε ItemClass και ItemClassMemberType κτλ.

Όταν πρέπει να επιλεγούν δεδομένα που δεν αντιστοιχούν σε κάποιο ορισθέν structure(πχ από μια λίστα με αντικείμενα πελάτης να επιλεχθεί για τον κάθε πελάτη μόνο το όνομα και η ηλικία και όχι η διεύθυνση) ο κώδικας είναι ακόμα πιο μακροσκελής. γιατί θα πρέπει η να ορισθούν ειδικά structures για να κρατάνε τα δεδομένα επιστροφής.

Στην c# αντίθετα το παραπάνω γίνεται με μια μόνο sql like εντολή

```
from i in collection where i.SatisfiesConditions() select
i.somemember orderby i.somemember.somefunction()
ascending;
```

5.3.1 LINQ

Η τεχνολογία που επιτρέπει τέτοιες εντολές ονομάζεται Language Integrated Query (LINQ). Εκτός από την C# μπορεί να χρησιμοποιηθεί με παραπλήσιο τρόπο και στις άλλες γλώσσες που υποστηρίζει το περιβάλλον .NET

Το LINQ αποτελεί μια προσπάθεια να αναπτυχθεί ένα general purpose data query api (γενικής χρήσης API χειρισμού δεδομένων) που να επιτρέπει το data handling δεδομένων από πηγές όπως βάσεις δεδομένων αρχεία xml και αντικείμενα στην

κεντρική μνήμη. Σε κάθε περίπτωση το βασικό συντακτικό διατηρείται σταθερό και έχει μορφή πολύ όμοια αλλά όχι ταυτόσημη με αυτήν της sql. Το συντακτικό είναι επίσης απλούστερο του συντακτικού για τέτοιες εντολές που έχουν οι συνήθεις ΟΟ γλώσσες όπως η Java. Αποτελεί κατά κάποιο τρόπο ένα DSL (Domain Specific Language). Δεν γίνεται αποστολή μηνυμάτων σε αντικείμενα με τον τελεστή . (dot) αλλά οι εντολές παρατίθενται όπως στην SQL.

Οι εντολές αυτού του dsl αναλύονται από τον compiler και μετατρέπονται εν τέλει στον πραγματικό κώδικα του query. Η ανάλυση από τον compiler έχει το πρόσθετο πλεονέκτημα του static checking που ανακαλύπτει νωρίς τυχόν λάθη. Ο κώδικας LINQ τελικά μετατρέπεται σε για παράδειγμα εντολές πάνω σε αντικείμενα στην μνήμη ή εντολές sql που τρέχουν σε ένα DBMS κτλ.

Τα κύρια τεχνικά χαρακτηριστικά πίσω από αυτήν την τεχνολογία, τα οποία επιτρέπουν την συγγραφή σύντομου κώδικα και την αφαιρετικότητα ως προς τα σύνολα δεδομένων που υποστηρίζονται, είναι τα εξής

Type Inference
Lamda Expressions
Anonymous Types

Type Inference

Το inference τύπων επιτρέπει στον compiler να αποδίδει τύπους σε μεταβλητές με βάση την τιμή αρχικοποίησης τους. Ο τύπος μετά την αρχικοποίηση είναι μόνιμος και δεν γίνεται η μεταβλητή να τον αλλάζει κατά την εκτέλεση του προγράμματος όπως στις scripting και δυναμικές γλώσσες. Μειώνεται όμως το χρονικό κόστος σε υπολογισμό και πληκτρολόγηση των τύπων καθώς και αυξάνεται η σαφήνεια του κώδικα.

Όταν χρησιμοποιούνται ανώνυμοι τύποι, που είναι βασικοί για την λειτουργία του LINQ, ο προγραμματιστής δεν μπορεί ούτως η άλλος να τους κατονομάσει σε μια μεταβλητή. Έτσι το type inference όχι απλώς είναι μια διευκόλυνση αλλά είναι αναγκαίο.

```
var personsNotInSeattle = from person in persons
                          where person.Address !=
                          "Seattle"
                          orderby person.FirstName
                          select new
                          {person.FirstName,
                          person.Address};
    lstResults.Items.Clear();
    foreach (var person in personsNotInSeattle)
    {
          lstResults.Items.Add(person.FirstName
            + " --- "
            + person.Address );
    }
```

Σε αυτό το κομμάτι κώδικα δεν ορίζεται ποιος είναι ο τύπος της μεταβλητής personsNotInSeattle αλλά με τον τελεστή var ζητείται από τον compiler να τον βρει με inference.

Anononymous types

Στο συγκεκριμένο παράδειγμα γίνεται και χρήση ανώνυμων τύπων. Στην γραμμή

```
select new {person.FirstName, person.Address};
```

επιλέγονται όπως και στο select της sql τα επιστρεφόμενα δεδομένα, μόνο που εδώ το container persons από όπου λαμβάνονται δεν είναι πίνακας sql αλλά συλλογή C#

αντικειμένων. Επειδή το select πρέπει να επιστρέψει αντικείμενα C# γεννάται το ερώτημα τι τύπου αντικείμενο θα πρέπει να επιστρέψει αν στο select επιλέχθηκαν μόνο ένα υποσύνολο των πληροφοριών σε κάθε αντικείμενο person του persons ή αν οι πληροφορίες υπολογίζονται με κάποια κλήση μεθόδου μέσα στο select και δεν είναι προαποθηκευμένες στο person.

Προφανώς θα ήταν λάθος να επιστραφεί ολόκληρο το αντικείμενο person μαζί με τα όποια επιπλέον πεδία πλην των Firstname, Address. Αρά θα πρέπει να επιστραφεί κάποιο ειδικό struct (δομή/αντικείμενο) που ορίστηκε έτσι ώστε να κρατάει τα αποτελέσματα του συγκεκριμένου query. Χωρίς την χρήση ανώνυμων τύπων αυτό θα απαιτούσε τον ορισμό στον πηγαίο κώδικα ενός νέου τύπου για το εν λόγω struct. Κάτι τέτοιο θα ήταν μη αποδοτικό και θα απαιτούσε πλήθος γραμμών περιττού “boilerplate” κώδικα, αφού για κάθε διαφορετικό select θα έπρεπε να ορίζεται και ένας νέος τύπος.

Το LINQ για αυτό τον σκοπό επιτρέπει την επιστροφή αντικειμένων ανώνυμων τύπων οι οποίοι δεν ορίζονται στον πηγαίο κώδικα αλλά κατασκευάζονται στο background από το linq με βάση το query. Οι ανώνυμοι τύποι δεν έχουν καν δικό τους όνομα όσον αφορά τον προγραμματιστή, το LINQ και ο compiler χρησιμοποιούν ένα δικό τους εσωτερικό όνομα. Η σύνταξη new {person.FirstName, person.Address} ουσιαστικά ζητάει από το LINQ να δημιουργήσει έναν ανώνυμο τύπο με δύο πεδία. Ο τύπος τους προκύπτει με type inference των τιμών που τα αρχικοποιούν, και ονόματα, αν δεν δοθούν νέα, προκύπτουν από τα ονόματα των πεδίων που χρησιμοποιούνται στην αρχικοποίηση.

Η μεταβλητή personsNotInSeattle μέσω type inference γίνεται τύπου container ακριβώς του ανώνυμου τύπου που δημιούργησε το linq πχ IEnumerable<GeneratedType1>. Τελικά ο προγραμματιστής δεν ασχολείται καθόλου με το να δημιουργήσει ειδικές για την περίπτωση δομές αποτελεσμάτων ή να βρει τον τύπο που πρέπει να έχει το αποτέλεσμα του query και άλλες μεταβλητές μέσα σε αυτό.

Lambda functions

Τα lambda functions αποτελούν μια μορφή συντακτικού που επιτρέπει τον σύντομο και inline ορισμό συναρτήσεων. Στην ουσία αποτελούν αυτό που ονομάζεται syntactic sugar του μηχανισμού ανώνυμων nested συναρτήσεων της C#. Οι nested συναρτήσεις είναι συναρτήσεις που ορίζονται στο εσωτερικό μεθόδων, και επιτρέπεται να μην τις δοθεί όνομα από τον προγραμματιστή οπότε είναι και ανώνυμες.

Η κλήση μιας nested συνάρτησης IF ορισμένης στο εσωτερικό μιας μεθόδου Α γίνεται για παράδειγμα όταν η Α καλεί την συνάρτηση Β δίνοντας της ένα όρισμα τύπου delegate (Τα delegates μπορεί κανείς να τα δει σαν type safe εκδοχές των function pointers της C, δηλαδή είναι δείκτες σε συναρτήσεις που έχουν συγκεκριμένη signature) που δείχνει στην IF. Η Β τότε μέσω του δείκτη μπορεί να καλεί την IF..

Όταν εκτελεστεί η IF έχει την ίδια εμβέλεια με αυτήν του σημείου ορισμού της στην Α. Δηλαδή οι τοπικές μεταβλητές της Α μπορούν να χρησιμοποιηθούν στον κώδικα της IF, σαν ο κώδικας να ήταν της Α και όχι της IF. Για αυτό το λόγο οι τοπικές μεταβλητές της Α δεν χάνονται μετά από την ολοκλήρωση της. Αντίθετα διατηρούνται στην μνήμη και μετά το πέρας της Α, εφόσον όμως ο garbage collector βρίσκει αναφορές κάποιας nested συνάρτησης της Α σε δείκτες delegate (ο garbage collector συμμετέχει σε αυτήν την διαδικασία γιατί κάθε delegate υλοποιείται σαν αντικείμενο και όχι σαν 32bit διεύθυνση μνήμης όπως στην C)

Για παράδειγμα

```
class C
{public Delegate void d();[..] public void aMethod();[...]}

A(){
int n = 0;
Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };} (1)

C c=new C(); c.d=d; return;}
```

Παρόλο που η A έχει ολοκληρωθεί οι μέθοδοι της C όπως η aMethod θα μπορούν να καλούν την ανώνυμη nested συνάρτηση του σημείου (1), ενώ η μεταβλητή n θα είναι η ίδια σε κάθε κλήση και η τιμή της θα είναι αυξημένη κατά 1. Όταν η συνάρτηση πάψει να είναι προσβάσιμη στον γράφο του garbage collector, τότε και αυτή και η μεταβλητή n θα διαγραφούν.

Στο παρασκήνιο μια εντολή LINQ όπως η

```
from c in conatainer where c.someint>5 select
c.Compute();
```

Αντιστοιχεί, αν αφαιρεθεί το syntactic sugar, σε μια αλληλουχία κλησης μεθόδων

```
container.Where(c=>c.someint>5).Select(c=>c.Compute());
```

Τα ορίσματα των συναρτήσεων Where και Select είναι lambda functions και τύπου delegate. Η where η select και οι άλλες συναρτήσεις του LINQ είναι generic. Δέχονται αναλόγως με τον τύπο container CT όπου χρησιμοποιούνται και ένα όρισμα delegate με παραμέτρους και επιστρεφόμενες τιμές τύπου σχετικού με τα στοιχεία του CT.

Τα ορίσματα που δίνονται στις Where, Select κτλ πρέπει να είναι συναρτήσεις. Δεν είναι εύχρηστο όμως κάθε φορά να ορίζεται και νέα συνάρτηση για να δοθεί σαν

ορίσμα της εντολής LINQ και μόνο. Για αυτό ο μηχανισμός των lambda functions επιτρέπει την σύντομη δημιουργία ανώνυμων συναρτήσεων μέσα σε πολύ λίγες γραμμές και με την χρήση type inference ώστε να μην είναι αναγκαία ούτε η πληκτρολόγηση τύπων ορισμάτων και επιστροφής. Σημειώνεται πως δεν έχει σημασία το ότι οι συναρτήσεις δεν έχουν ξεχωριστό όνομα αφού χρησιμοποιούνται μόνο στο συγκεκριμένο σημείο.

Η έκφραση

```
c=>c.someint>5
```

Αντιστοιχεί στην συνάρτηση

```
bool Anonymous(CsType c) { return c>5; }
```

Αριστερά τον τελεστή => γίνεται type inference των ορισμάτων. Δεξιά ορίζεται το σώμα της συνάρτησης και γίνεται type inference της επιστρεφόμενης τιμής (από τις τιμές των εντολών return)

Έτσι οι συναρτήσεις μπορούν να γραφούν πολύ γρήγορα και να τοποθετηθούν κατευθείαν ως ορίσματα. Συναρτήσεις χωρίς ορίσματα δίνονται ως ()=>body, ενώ μπορεί να παρακάμψει κανείς το type inference π.χ. με το εξής συντακτικό
(int a, int b, c)=>{[...]}

Με συντακτική ζάχαρη αφαιρείται και η χρήση των τελεστών => και dot, και έτσι προκύπτει η εντολή LINQ Παρόλα αυτά ο τρόπος συγγραφής ερωτημάτων LINQ με αλληλουχία κλήσεων και χρήση lambda functions χρησιμοποιείται συχνά.

Ένα επιπλέον χαρακτηριστικό του LINQ είναι τα extension methods. Αυτά είναι ένας μηχανισμός μέσω του οποίου επιτρέπεται η απευθείας με τον τελεστή dot κλήση των συναρτήσεων LINQ, σε αντικείμενα τύπων container, ακόμα και αν δεν

έχουν ορίσει πουθενά τις συγκεκριμένες μεθόδους, όπως πχ αντικείμενα των τύπων array ή List<T>

Οι πιο σημαντικές μέθοδοι που προσθέτει το LINQ σε container τύπους είναι οι

Aggregate, All, Any, Average, Distinct, Except, First, Intersect, Last, Max, Min, Union, Skip, Take, Where, Select, OrderBy, GroupBy

Οι Aggregate, Sum, Min, Max, Average λειτουργούν όπως οι αντιστοιχεί τελεστές συνάθροισης της SQL.

Οι Where, Select, OrderBy, Distinct λειτουργούν όπως τα αντίστοιχα τμήματα των εντολών SQL

Οι First και Last επιλέγουν το πρώτο και το τελευταίο στοιχείο που πληρεί κάποια κριτήρια

Οι All, Any, Union, Intersect, Except υποστηρίζουν set like operations σε αντικείμενα container της C#.

Η OrderBy κάνει sorting και η GroupBy κάνει aggregation αντίστοιχα όπως στην SQL.

Όλες αυτές οι μέθοδοι μπορούν να δέχονται ένα όρισμα lambda function. Το lambda function παίρνει διαδοχικά τα στοιχεία του container και τα επεξεργάζεται με σχετικό με το είδος της μεθόδου τρόπο.

Ο κώδικας που προκύπτει τελικά μπορεί να είναι πολύ σύντομος. Για παράδειγμα

```
bool a =
container1.Any(c1=>container2.All(c2=>c2.SomeTest(c1)));
```

Αυτή η εντολή ελέγχει αν υπάρχει έστω και ένα στοιχείο c1 του container1 για το οποίο να ισχύει ότι για όλα τα στοιχεία c2 του container2 η κλήση c2.SomeTest(c2) δίνει true. Ο κώδικας που θα έκανε το ίδιο πράγμα χωρίς LINQ θα ήτανε πιο εκτενής και επιρρεπής σε λάθη.

5.4 Aspects

Το Aspect Oriented Programming είναι τεχνική αντιμετώπισης των λεγόμενων cross cutting concerns. Αυτός ο όρος αποδίδεται σε επιπλέον λειτουργίες που πρέπει να εκτελούν σύνολα από τις υπομονάδες του προγράμματος, και που για λόγους modularization δεν είναι σωστό να τοποθετηθούν μέσα στον κώδικα των υπομονάδων. Συνήθως τα crosscutting concerns έχουν να κάνουν με λειτουργίες όπως το logging, τα transactions κ.α. Για παράδειγμα αν όλες οι συναρτήσεις μιας εφαρμογής πρέπει να καταγράφουν σε log την εκκίνηση τους, η μόνη δυνατή με βάση τις προγραμματιστικές δομές της C# υλοποίηση είναι να τοποθετηθεί σε κάθε μια συνάρτηση ο κώδικας εκτύπωσης στο log. Έτσι ο ίδιος κώδικας επαναλαμβάνεται σε πολλά σημεία κάτι που δεν αποτελεί δομημένο σχεδιασμό. (Σημείωση ότι κάτι αντίστοιχο για αντικείμενα είναι πιο απλό καθώς αυτά δεν έχουν ροή εκτέλεσης και οι καταγραφές μπορούν να γίνουν όλες μαζί από μια κεντρική ροή ελέγχου, ο μηχανισμός των annotations εξυπηρετεί αυτόν τον σκοπό [Troelsen 2007])

Λόγω της πολύπλοκης ιεραρχικής δομής του kdb tree είναι δύσκολο μετά το πέρας μιας λειτουργίας insert/query/delete, να προσδιοριστεί τι αλλαγές προκάλεσε αυτή στην δομή. Οι αλλαγές μπορεί να είναι τόσο μεγάλες (πχ στην διάσπαση στην ρίζα) ώστε η σύγκριση παλιάς και νέας κατάστασης είναι αλγοριθμικά δύσκολη. Οπότε πρέπει να γίνεται καταγραφή των αλλαγών επί τόπου ενώ γίνονται, δηλαδή με την χρήση ενός log.

Στον κώδικα μιας λειτουργίας για να γίνει πιο εύκολη η αποσφαλμάτωση πρέπει να υπάρχουν παράλληλα με τις εντολές του αλγορίθμου της, και κάποιοι έλεγχοι που εκτελούνται πριν και μετά από αυτήν (design by contract). Άλλοι έλεγχοι συμμετέχουν μαζί σε πολλές λειτουργίες και άλλοι όχι. Η ομαδοποίηση τους σε συναρτήσεις υστερεί σε ευελιξία καθώς δεν πρέπει να ομαδοποιούνται μαζί έλεγχοι που δεν μπορούν για κάποια λειτουργία να εφαρμοστούν και οι δύο. Έτσι αναγκαστικά οι έλεγχοι τοποθετούνται οι ίδιοι στο εσωτερικό της συνάρτησης Σ μιας λειτουργίας Λ που ελέγχουν. Αυτό όμως κάνει τον κώδικα δυσνόητο, και μειώνει το cohesion της Σ αφού τώρα κάνει δύο πράγματα μαζί, υλοποιεί και ελέγχει την Λ. (Μάλιστα αν συμπεριλάβει κανείς και το logging που αναφέρθηκε παραπάνω η κάθε συνάρτηση κάνει 3 πράγματα.)

Το AOP Aspect Oriented Programming αντιμετωπίζει τα δύο παραπάνω πρόβλημα σχετικά με το cohesion και τον μη δομημένο σχεδιασμό, επειδή και τα δύο αποτελούν είδος cross cutting concern. Η τεχνική του AOP βασίζεται στον καθορισμό σημείων εισόδου εξόδου σε κάθε συνάρτηση και την αντιστοίχηση σε αυτά κομματιών κώδικα. Η αντιστοίχηση γίνεται από ένα κεντρικό σημείο, εξωτερικά του κώδικα των συναρτήσεων, μέσω κάποιου framework ή Domain Specific Language.

Δεν χρειάζεται π.χ. ένα κομμάτι κώδικα logging να αντιγραφεί σε όλες τις συναρτήσεις, αλλά επιλέγεται σε ένα κεντρικό σημείο ποιες θα είναι αυτές που θα το χρησιμοποιήσουν (το ένα και μοναδικό αντίγραφο του). Ο κώδικας AOP είτε κατονομάζει τις επιλεγμένες συναρτήσεις ή ακόμα και τις εντοπίζει μέσω κάποιου string pattern.

Με την χρήση AOP ο κώδικας είναι πιο καθαρός και δεν θα υπάρχουνε περιττές εξαρτήσεις. Αλλιώς οι συναρτήσεις υλοποίησης του kdb tree αναγκαστικά περιέχουν μέσα τους ελέγχους Preconditions και Postconditions καθώς και διάφορες εντολές logging, κάτι που τις κάνει να είναι πιο δυσνόητες και όχι καλά modularized για εξωτερική χρήση. Δυστυχώς στο πρόγραμμα της εργασίας δεν χρησιμοποίησα AOP

γιατί η υποστήριξη της τεχνικής στην C# δεν είναι ικανοποιητική.

5.5 Debugging

Όπως αναφέρθηκε και στην αρχή της ενότητας ενότητας 5, σε μεγάλο βαθμό η δυσκολία στην ανάπτυξη της εφαρμογής βρίσκεται στο debugging. Τα λάθη σε ένα τόσο μεγάλο και αλγοριθμικό πρόγραμμα είναι σχεδόν αναπόφευκτα και δεν οφείλονται τόσο στην ανικανότητα του προγραμματιστή (σε μεγάλο βαθμό οι αλγόριθμοι δίνονται ήδη στο paper) αλλά στο ότι σε ένα πλήθος εντολών όπου η μια εύκολα επηρεάζει την άλλη, το παραμικρό τυχαίο λάθος οδηγεί σε κατάρρευση της δομής και δυσκολία εύρεσης του σημείου που έγινε αυτή.

Ο πιο απλός και γνωστός τρόπος αποσφαλμάτωσης είναι η παρατήρηση της κατάστασης του προγράμματος (κατάσταση = τιμές μεταβλητών, stack κλήσεων) την στιγμή ενός runtime error, και ο έλεγχος της ορθότητας της ροής εκτέλεσης εντολών μέχρι το σημείο εκείνο.

Αυτή η διαδικασία δεν είναι εύκολη στην συγκεκριμένη εφαρμογή γιατί η κατάσταση της δομής είναι ιδιαίτερα πολύπλοκη**(3)**. Δεν είναι εύκολο να εντοπιστεί τι είναι λάθος σε ένα δένδρο από κόμβους παραλληλόγραμμα ν διαστάσεων. Ένας άνθρωπος δυσκολεύεται π.χ. να υπολογίσει αν οι θυγατρικοί κόμβοι ενός κόμβου αποτελούν διαμέριση του γονέα τους.

Επίσης η ροή εκτέλεσης εντολών περιλαμβάνει αναδρομικές κλήσεις, χρήση στοίβας, διακλαδώσεις βάση μαθηματικών συνθηκών κτλ. Έτσι είναι δύσκολο να προβλεφθεί η κανονική ροή και να εντοπισθεί το σημείο που το πρόγραμμα παρέκκλινε από αυτήν.

Επομένως όταν συμβεί ένα run time error ο προγραμματιστής παρατηρώντας την κατάσταση του προγράμματος δεν μπορεί να καταλάβει τι ακριβώς έχει γίνει λάθος, και ποιοσημείο του κώδικα είναι υπεύθυνο γα αυτό.

Για παράδειγμα αν προκληθεί ένα NullReferenceException, ο προγραμματιστής ξέρει μεν ότι έχει γίνει προσπάθεια χρήσης null αναφοράς, αλλά δεν γνωρίζει

-Ποιο λάθος στην εφαρμογή των αλγορίθμων έχει γίνει **(1)**
-Σε ποιο σημείο του πηγαίου κώδικα έγινε **(2)**

Το (1) έχει να κάνει με το ότι το NullReferenceException είναι ένα γενικό είδος σφάλματος το οποίο μπορεί να προκληθεί από πολλούς τύπους λάθος εφαρμογής των αλγορίθμων. Για παράδειγμα τόσο η παράλειψη εισαγωγής κάποιου region node στον γονέα του, όσο και η λάθος υλοποίηση του υπολογισμού αν ένα σημείο ανήκει σε κάποιο region, μπορεί να προκαλούν ακριβώς τον ίδιο τρόπο διακοπής του προγράμματος, NullReferenceException σε μια γραμμή Γ1 που λόγω του (2) βρίσκεται μακριά από τα σημεία του πηγαίου κώδικα όπου έγινε το λάθος.

Λόγω του (3) θα είναι δύσκολη η κατανόηση του σφάλματος μέσω μελέτης της κατάσταση του προγράμματος την ώρα του NullReferenceException από τον προγραμματιστή. Πρέπει επομένως τα λάθη να προσδιορίζονται με καλύτερο τρόπο, να μην δίνονται γενικά exceptions όπως το NullReferenceException αλλά εξειδικευμένοι έλεγχοι του προγράμματος να δίνουν (στο παράδειγμα αυτό) πχ RegionsNotIncreasedByOneDuringSplitError ή NoRegionContainsPointError.

Το (2) είναι συναφές του (1) και έχει να κάνει με το ότι ένα λάθος του πηγαίου κώδικα SCE δεν προκαλεί runtime error κατ ανάγκη κοντά (σε απόσταση γραμμών κώδικα και κλήσεων μεθόδων, δηλαδή εκτελεσθέντων εντολών) στο SCE.

Λόγω της περίπλοκης ροής εκτελέσεων συναρτήσεων που προβλέπουν οι αλγόριθμοι του kdb tree, ένα λάθος στο σημείο Α του πηγαίου κώδικα θα μπορούσε να προκαλέσει το runtime error σε ένα σημείο Β πολύ μακριά του Α σε τοπικότητα πηγαίου κώδικα και σε χρονική σειρά στην ροή εκτέλεσης.

Για να αντιμετωπιστούν τα (1) και (2) οι έλεγχοι που ο προγραμματιστής θα έκανε "χειρωνακτικά" θα πρέπει να γίνουν με αυτόματο τρόπο από συναρτήσεις του προγράμματος ειδικές για αυτόν τον σκοπό. Για παράδειγμα ένα κομμάτι κώδικα να ελέγχει σε κάθε διάσπαση αν τα subregions ενός κόμβου Κ δίνουν όντως διαμέριση του. Πρέπει μάλιστα οι έλεγχοι να προσδιορίζουν ποιου τύπου ακριβώς λάθος έχει γίνει (1) και ποια υπομονάδα του προγράμματος ήτανε υπεύθυνη(2). Έχοντας καλύτερη πληροφορία για το είδος και την τοποθεσία του σφάλματος ο προγραμματιστής θα το εντοπίσει και θα το διορθώσει πιο εύκολα.

Οι συγκεκριμένοι έλεγχοι που χρειάζεται να γίνουν προκύπτουν από τα invariants της δομής. Δηλαδή σε κάποιο βαθμό ο προγραμματιστής δεν χρειάζεται να τους προσδιορίσει αλλά είναι ήδη γνωστοί. Τελικά η δομή πρέπει όχι μόνο να υλοποιεί το kdb tree αλλά και να ελέγχει τον εαυτό της καθώς το κάνει, με σκοπό διευκολύνει και να καθοδηγήσει τον προγραμματιστή κατά την διόρθωση λαθών.

5.5.1 Υλοποίηση ελέγχων

Τα invariants εξ ορισμού προβλέπουν πως η διαδικασία του ελέγχου γίνεται μόνο μετά το πέρας ολόκληρης της λειτουργίας, και όχι κατά τμηματικά την διάρκεια της. Ο έλεγχος δεν μπορεί να υλοποιηθεί έτσι. Αυτό συμβαίνει γιατί το λάθος μπορεί να έχει αλλάξει τόσο πολύ την κατάσταση της δομής που ο προσδιορισμός του με όλες αυτές τις αλλαγές θα είναι δύσκολος ακόμα και με χρήση ειδικού κώδικα. Επίσης γιατί η κάθε λειτουργία κάνει χρήση μεγάλου πλήθους υποβημάτων και ο έλεγχος μόνο των invariants στο τέλος ή στην αρχή της θα δυσκόλευε τον εντοπισμό του συγκεκριμένου σημείου όπου συνέβη το λάθος. Για παράδειγμα η εισαγωγή εκτελείται μέσω της αναδρομικής κλήσης κάποιας συνάρτησης σε πλήθος κόμβων του δένδρου. Ο έλεγχος μόνο στην αρχή και το τέλος του συνόλου όλων αυτών των κλήσεων δεν δίνει καλή πληροφορία για το τι έγινε στον κάθε ένα ξεχωριστά. Οπότε αυτό που πρέπει να γίνει είναι η δημιουργία νέων invariants, βασισμένων σε αυτά των λειτουργιών, των οποίων το πεδίο ελέγχου να μην είναι αυτό όλης της λειτουργίας άλλα να ελέγχουν ξεχωριστά κάθε συνάρτηση-υπορουτίνα αυτής. Πιο

μετά εξηγείται πως αυτό γίνεται μέσω του design by contract.

* * *

Μια αρχιτεκτονική οργάνωσης του ελέγχου για λάθη είναι η δημιουργία tests (unit testing) και η εκτέλεση τους κάθε φορά που αλλάζει ο πηγαίος κώδικας, ώστε να εντοπίζεται αυτόματα το λάθος. Γενικά όμως αν και δοκίμασα να χρησιμοποιήσω μερικά unit tests, από ένα σημείο και πέρα η συγγραφή τους ήτανε δύσκολη γιατί έλεγχαν την δομή με τρόπο εκ των υστέρω και γιατί υπήρχε η απαίτηση κάθε φορά να τροφοδοτούνται με το κατάλληλο για τον εκάστοτε έλεγχο σύνολο δεδομένων ελέγχου (test data). Τα test data υποτίθεται αντιπροσωπεύουν μια περίπτωση χρήσης που αναπαράγει ένα πιθανό σφάλμα. Ο σχεδιασμός τους όμως γινότανε πολύ δύσκολος γιατί τα δεδομένα του kdb, καθώς είχανε ιεραρχικότητα, και χρειαζόταν να τηρούν συγκεκριμένες και δύσκολες τον υπολογισμό μαθηματικές ιδιότητες ώστε να μπορούν να προκαλέσουν επιθυμητά σφάλματα. Οπότε αυτή η αρχιτεκτονική ελέγχου δεν χρησιμοποιήθηκε όσο το Design by Contract.

* * *

Σε κάποια σημεία οι απαιτούμενοι έλεγχοι ήτανε δύσκολο να υλοποιηθούν γιατί στην ουσία απαιτούσαν εξελιγμένους αλγορίθμους υπολογιστικής γεωμετρίας. Για παράδειγμα ο έλεγχος αν τα subregions sr ενός region R αποτελούν διαμέριση του. Με το μάτι και με γραφική παράσταση ο έλεγχος είναι εύκολος και από τον προγραμματιστή. Αλλά με λίστες που περιγράφουν σε text την δομή και σε περισσότερες των τριών διαστάσεις είναι σχεδόν αδύνατο η διαδικασία ελέγχου να γίνει από άνθρωπο και όχι από τον υπολογιστή.

Τότε όμως ο αλγόριθμος υπολογιστικής γεωμετρίας που θα κάνει αυτήν την δουλειά είναι σχετικά περίπλοκος γιατί η ανάπτυξη του χρειάζεται μαθηματική κατανόηση της γεωμετρίας του προβλήματος. και στην εργασία ζητήθηκε να μην ασχοληθώ καν με τέτοια θέματα. Από την άλλη υπάρχουν και λιγότερο έξυπνοι και πιο brute force

αλγόριθμος, αλλά αυτοί έχουν απαγορευτικά μεγάλο χρόνο εκτέλεσης.

Επειδή όμως έπρεπε να γίνουν οι έλεγχοι των διαμερίσεων στα πλαίσια του debugging, στην εργασία έγινε χρήση μιας τεχνικής τύπου "monte carlo". Αυτή δεν δίνει σίγουρα τα σωστά αποτελέσματα γιατί είναι στοχαστική, συγκεκριμένα υπήρχε ο κίνδυνος να μην αντιλαμβανόταν κάποιο μικρό κενό ή overlap στην διαμέριση. Με μεγάλη πιθανότητα όμως αν δεν βρει κάποιο λάθος τότε όντως δεν υπήρχε λάθος.

Για τον έλεγχο της διαμέρισης επιλεγόταν με ομοιόμορφη κατανομή πλήθος σημείων (της τάξης των εκατομμυρίων) στον χώρο και ελεγχόταν σε πόσα subregions άνηκε το κάθε ένα. Αν έστω και ένα άνηκε σε περισσότερα του ενός subregions (overlap), η λιγότερα του ενός (κενά στην διαμέριση) τότε εντοπιζόταν σφάλμα στην διαμέριση.

Σημειώνεται ότι όταν κάποιο σφάλμα εντοπιζόταν αυτό σήμαινε και το ότι υπήρχε. Η πιθανότητα που κάνει στοχαστική την διαδικασία είναι να μην βρεθεί κάποιο υπάρχων σφάλμα (false negative) και όχι να βρεθεί ενώ δεν υπάρχει (false positive).

5.5.2 Design by contract

Το design by contract είναι μια αρχιτεκτονική που προσπαθεί να οργανώσει με δομημένο και ιεραρχικό τρόπο την ανάθεση στις υπομονάδες του προγράμματος της υπευθυνότητας για την πρόκληση σφαλμάτων και της εκτέλεσης των ελέγχων για αυτά. Ένα δομημένο πρόγραμμα προσπαθεί να μοιράσει σε υπομονάδες το σύνολο των ενεργειών που πρέπει να εκτελεστούν. Κατά παρόμοιο τρόπο ένα δομημένο και σχεδιασμένο με design by contract πρόγραμμα προσπαθεί να μοιράσει επιπλέον και την υπευθυνότητα για τα λάθη.

Για παράδειγμα έστω οι συναρτήσεις Α και Β όπου η Α καλεί την Β για να εκτελέσει ένα μέρος της επεξεργασίας. Τόσο η Α όσο και η Β μπορούν να έχουν σφάλματα

κώδικα. Όμως ένα σφάλμα πηγαίου κώδικα στην συνάρτηση Β, μπορεί να προκαλέσει runtime error όχι μόνο κατά την εκτέλεση της Β αλλά και κατά την εκτέλεση της Α. Αυτό θα συμβεί αν η Β επιστρέψει στην Α λανθασμένα δεδομένα.

Ομοίως ένα λάθος στον πηγαίο κώδικα της Α μπορεί να προκαλέσει σφάλμα χρόνου εκτέλεσης ενώ εκτελείται η Β, αν η Α δώσει στην Β λανθασμένα ορίσματα.

Η αποσφαλμάτωση δεν γίνεται επομένως να βασιστεί στο που έγινε το runtime error ειδικά αν η αλυσίδα κλήσεων είναι ακόμα μεγαλύτερη και για παράδειγμα αν η Μ καλεί την Α ενώ η Β την Γ. (Τέτοιες αλυσίδες είναι συχνές στην υλοποίηση του kdb tree).

Ταυτόχρονα με την εντοπισμό της υπομονάδας που ευθύνεται για το λάθος πρέπει να γίνει οργανωμένα και η εκτέλεση των ελέγχων που το προσδιορίζουν και το χαρακτηρίζουν, καθώς ακόμα και αν ο προγραμματιστής ξέρει που έγινε το λάθος, η κατάσταση του προγράμματος εκείνη την ώρα μπορεί να είναι τόσο πολύπλοκη που να μην είναι εύκολο παρατηρώντας την να βρει τι φταίει.

Συνοπτικά το design by contract οργανώνει την διαδικασία ως εξής. Κάθε συνάρτηση (πιο αφηρημένα υπομονάδα) θεωρείται υπεύθυνη για τους εξής 3 τομείς. Διατήρηση κάποιου αναλλοίωτου στις δομές που χρησιμοποιεί, παράδοση σωστών ορισμάτων στα μηνύματα κλήσης των άλλων συναρτήσεων που καλεί και επιστροφή σωστών αποτελεσμάτων όταν καλείται. Χωρίς να μπει η περιγραφή σε λεπτομέρειες εντοπίστηκε από τους εμπνευστές του DbC ότι ο καλύτερος τρόπος να οργανωθεί η παραπάνω διαδικασία είναι η κάθε συνάρτηση να ελέγχει πριν την εκτέλεση της κατά σειρά τα invariants και τα preconditions , και μετά το πέρας της κατά σειρά τα postconditions και ξανά τα invariants.

Ο έλεγχος γίνεται με κώδικα που όταν εντοπίζει κάποιο λάθος σταματάει την εκτέλεση και ενημερώνει με λεπτομερή περιγραφή του σφάλματος. Ο κώδικας επομένως τοποθετείται στην αρχή και στο τέλος κάθε συνάρτησης (όπως

αναφέρθηκε προηγουμένως αυτό θα ήταν καλύτερο να γίνει με AOP) και προκαλεί exception όταν ο έλεγχος αποτυγχάνει. Με αυτήν την λογική όταν η Α καλεί την Β είναι σίγουρη ότι τα αποτελέσματα της Β είναι σωστά, γιατί η Β τα ελέγχει. Έτσι αν γίνει κάποιο λάθος η Β θα το εντοπίσει και θα σταματήσει την ροή πριν αυτή γυρίσει στην Α. Ταυτόχρονα θα προσδιορίσει με κάποιο τρόπο το σφάλμα ώστε να γίνει αντιληπτό από τον προγραμματιστή.

Αν αντίθετα όλοι οι έλεγχοι γινόταν από την Α που καλούσε την Β, η Β την Γ, η Γ την Δ κ.ο.κ η Α θα έβρισκε μεν κάτι λάθος αλλά δεν θα μπορούσε να προσδιορίσει σε ποια από τις άλλες συναρτήσεις της αλυσίδας έγινε.

Αφού η Β αναλαμβάνει τους δικούς της ελέγχους δεν χρειάζεται να επαναλαμβάνονται αυτοί ξανά στην Α, και αυτό έχει ένα μεγάλο όφελος καθώς η επανάληψη του ίδιου κώδικα χαλάει το modularization και προκαλεί τα συναφή προβλήματα. Το τελευταίο μπορεί να πάει ένα βήμα πιο πέρα όταν οι έλεγχοι κληρονομούνται με βάση την ιεραρχική οργάνωση των modules. Για παράδειγμα αν η Α είναι virtual συνάρτηση υπερκλάσης και οι Α1, Α2 εξειδικεύσεις της σε υποκλάσεις, τότε δεν χρειάζεται οι Α1 και Α2 να κάνουνε οι ίδιες με κώδικα τους ελέγχους που είναι κοινοί, αρκεί να τους κληρονομήσουν με κάποιο τρόπο από την Α. Η όταν το αντικείμενο Ο διαθέτει τις μεθόδους Α Β Γ που όλες ελέγχουν το invariant του Ο, τότε δεν χρειάζεται ο έλεγχος του invariant να επαναλαμβάνεται σαν κώδικας στις Α Β Γ, αρκεί να ορίζεται μια φορά σε επίπεδο κλάσης αντικειμένου και οι Α Β Γ να τον τρέχουν αυτόματα.

Για να γίνει σωστή εφαρμογή των παραπάνω χρειάζεται κάποια γλώσσα προγραμματισμού που να τα υποστηρίζει με κατάλληλες δομές. Τέτοιες γλώσσες είναι η Eiffel και η D. Στην C# επειδή ανάλογες δομές δεν υπάρχουν, οι έλεγχοι επαναλαμβάνονται κάθε φορά στην αρχή και το τέλος κάθε συνάρτησης και έχουνε την μορφή assertions. Παρουσίαση του πως γίνεται αυτό δίνεται στην ενότητα 6.

5.6 Αναπαράσταση και μετασχηματισμός της δομής

5.6.1 XML

Στην εργασία δεν ζητήθηκε να γίνει αποθήκευση δεδομένων στον δίσκο ή σε DBMS. Επομένως δεν ήταν απαραίτητο να γίνει κάποια αποθήκευση του kdb tree σε μορφή διαφορετική από αυτή του δένδρου αντικειμένων C# της κύριας μνήμης.

Παρ όλα αυτά χρησιμοποιήθηκε η xml για την ενδιάμεση αποθήκευση ενός αντιγράφου της δομής με σκοπό την εφαρμογή μετασχηματισμών σε XSLT.

Ορισμένες από τις λειτουργίες που έπρεπε να υλοποιηθούν ήτανε σχετικά δύσκολες και θα έπαιρναν περισσότερο χρόνο ανάπτυξης αν γινόταν χρήση της C#. Τέτοιες ήτανε για παράδειγμα η οπτικοποίηση της ιεραρχίας των κόμβων της δομής, (η οποία θα ήτανε σημαντική για το debugging), και η οποία για αντίστοιχες δενδρικές δομές που έχουν μορφή κειμένου xml γίνεται αυτόματα κατά το άνοιγμα του κειμένου από κάποιον browser ή κάποιο πρόγραμμα σαν το XMLViewer.

Θα ήτανε χάσιμο χρόνου να αναπτυχθεί απ την αρχή κώδικας που να υλοποιεί τις λειτουργίες αναπαράστασης δενδρικής δομής των browsers. Πιο αποδοτικό θα ήταν η δομή να μετατραπεί απ την μορφή των αντικειμένων Κεντρικής Μνήμης στην μορφή του xml κειμένου, ώστε οι browsers να μπορούν να χρησιμοποιήσουν το αρχείο και να κάνουν αυτοί την αναπαράσταση.

Η δενδρική δομή των δεδομένων έκανε αναγκαία την συχνή χρήση αναδρομής καθώς οι περισσότερες συναρτήσεις. κάνουν Depth First Search διασχίζοντας στο δένδρο με αναδρομικές κλήσεις στον εαυτό τους (και χρήση πολυμορφικότητας για να διαφοροποιηθούν ανάλογα με τον τύπο κόμβου region ή page.) Για την γραφική αναπαράσταση ενός δισδιάστατου kdb tree θα χρειαζόταν να αναπτυχθούν μερικές τέτοιες συναρτήσεις οι οποίες δεν θα εισήγαγαν ή αναζητούσαν σημεία αλλά με βάση τα περιεχόμενα κάθε κόμβου θα σχεδίαζαν στην οθόνη γραμμές και πολύγωνα.

Οι συναρτήσεις αυτές θα ενεργούσαν σε όλη την δομή χωρίς να την μεταβάλλουν. Είχαν δηλαδή χαρακτήρα μετασχηματισμού. Για την xml υπάρχει ειδική γλώσσα μετασχηματισμού, η XSLT, και θεωρήθηκε ότι λειτουργίες σαν αυτές θα μπορούσαν να υλοποιηθούν πιο γρήγορα και με πιο κατανοητό τρόπο αν γινόταν μετασχηματισμός με XSLT ενός XML αντιγράφου του kdb tree, αντί να υλοποιηθούν από την αρχή συναρτήσεις C#. Το αποτέλεσμα του μετασχηματισμού θα ήτανε ένα κείμενο xml που θα μπορούσε να χρησιμοποιηθεί από κάποια βιβλιοθήκη η πρόγραμμα σχεδιασμού γραφικών.

Το γενικότερο θέμα ήταν το πως θα χειριζόταν γενικά την δομή κάποιο εξωτερικό εργαλείο η πρόγραμμα. Προφανώς θα χρειαζότανε μια αναπαράσταση της ως όρισμα κατά την κλήση του προγράμματος. Οι δύο προηγούμενες παρατηρήσεις της ενότητας (προβολή σε browser, σχεδιασμός με graphics library) αποτελούν υποπεριπτώσεις αυτής της γενικής ανάγκης.

Ο καλύτερος τρόπος για την αποθήκευση μιας δομής σαν του kdb tree θεωρήθηκε ότι είναι η XML. Η XML είναι ήδη δενδρική και είναι ένα ανοικτό στάνταρ που επιτρέπει σε πολλά άλλα εργαλεία να ανοίγουν xml αρχεία (browsers, wpf, το πρόγραμμα XMLViewer με το οποίο μπορούσε κανείς να δει με ιεραρχικό τρόπο τα ογκώδη περιεχόμενα της δομής). Επίσης μέσω των τεχνολογιών xpath και xslt επιτρέπει την functional επεξεργασία μιας δομής (κάτι που φάνηκε χρήσιμο στην γραφική απεικόνιση).

Αν δεν γινόταν χρήση xml θα χρειαζόταν να αναπτυχθεί από την αρχή κώδικας αποθήκευσης ανάγνωσης και παρουσίασης των δεδομένων, ενώ δεν θα ήτανε δυνατή η επεξεργασία και ο μετασχηματισμός τους με τις Xpath, XSLT κτλ ή η εύκολη εμφάνιση τους στο XML Viewer. Αυτά τα εργαλεία υποστηρίζουν την xml ακριβώς γιατί είναι ανοικτό standar, ενώ οποιοσδήποτε άλλος ad hoc τρόπος για την αποθήκευση ενός δένδρου δεν θα ήταν και άρα δεν θα υποστηριζόταν. Έτσι αναγκαστικά θα έπρεπε να κατασκευαστούν πλήθος νέων συναρτήσεων για να επιτευχθεί ανάλογη λειτουργικότητα και αυτό θα καθυστερούσε και πολύ την

εκκίνηση της αποσφαλμάτωσης της δομής μέχρι να ολοκληρωθεί αυτή η ανάπτυξη (ένα είδος bottleneck)

5.6.2 Γραφική αναπαράσταση

Αφού δομή είχε αποθηκευτεί σε αντίγραφο XML ο κώδικας της γραφικής αναπαράστασης έπρεπε να μεταφράσει με κάποιο τρόπο την ιεραρχία κόμβων του XML αρχείου στην σχεδίαση γραμμών πολυγώνων και σημείων στην οθόνη.

Αυτό θα ήτανε σχετικά δύσκολο αν δεν υπήρχανε τεχνολογίες γραφικών που βασίζονται και οι ίδιες σε xml. Τέτοιες είναι για παράδειγμα η SVG (Scalable Vector Graphics) και η XAML/WPF. Σε αυτές ο ορισμός του τι θα ζωγραφίσει το πρόγραμμα δίνεται με έναν ιεραρχικό τρόπο. Δεν χρειάζεται ο προγραμματιστής να κατασκευάσει δύσκολο κώδικα που σκανάρει το δένδρο και καλεί εντολές κατά την επεξεργασία των κόμβων.

Για παράδειγμα το ακόλουθο κομμάτι xaml

```
<Canvas>
<Rectangle Canvas.Left=50 Canvas.Top=100 height=”100” width=”100” color=”blue” stroke=”black”/>
<Rectangle Canvas.Left=600 Canvas.Top=100 height=”100” width=”100” color=”blue” stroke=”black”/>
</Canvas>
```

Εδώ δίνεται δήλωση του τι θέλει ο προγραμματιστής να σχεδιάσει και όχι το πως θα γίνει αυτό. Οι εντολές εδώ σημαίνουν, “δημιούργησε έναν πεδίο σχεδιασμού (canvas), και στις θέσεις (50,100) και (600,100) αυτού ζωγράφισε δύο παραλληλόγραμμα με μήκη ακμών 100”. Δεν δίνονται χαμηλού επιπέδου εντολές όπως “σχεδίασε αυτήν την γραμμή ανάμεσα σε αυτά τα σημεία.

Επειδή και τα δύο αρχεία, xml του kdb tree και xaml της γραφικής παράστασης, αποτελούνε έγγραφα xml, είναι δυνατόν να γίνει χρήση κάποιου μετασχηματισμού με xslt ώστε από το αρχείο xml να δημιουργηθεί το xaml της παράστασης χωρίς να χρειαστεί καθόλου επιπλέον κώδικας. Αρκεί τα regions του xml να μετατραπούν σε Rectangles του wpf.

Στην ουσία τα μόνα tags που χρειάστηκε να γνωρίζει κανείς για την εργασία αυτή ήταν τα Canvas και Rectangle.

Αρχικά ο σκοπός ήτανε να χρησιμοποιηθεί η SVG που είναι παρόμοια και όχι η XAML/ WPF αλλά η Microsoft στην ουσία δεν υποστήριζε το πρότυπο SVG στο .NET καθώς αποτελεί ανταγωνιστική της XAML/WPF τεχνολογία.

5.6.2.1 XAML

XAML σημαίνει eXtensible Application Markup Language κάτι που υποδηλώνει ότι η διαφορά της XAML με την XML βρίσκεται στο ότι η πρώτη εξειδικεύεται στην υποστήριξη εφαρμογών.

Η XAML είναι μια γλώσσα αρχικοποίησης αντικείμενων Object Oriented γλωσσών. Βασίζεται στον τρόπο με τον οποίο η XML αναπαριστά δεδομένα. Ένα κείμενο XAML αποτελεί και κείμενο XML με την διαφορά ότι το πρότυπο της XAML προσθέτει επιπλέον σημασιολογία σε ορισμένες συντακτικές μορφές, τέτοιες ώστε η αρχικοποίηση των δομών μιας object oriented εφαρμογής να είναι πιο εύκολη.

Για παράδειγμα η σύνταξη Canvas.Top="300", στην κανονική XML.θα είχε την σημασιολογία “το attribute με όνομα Canvas.Top έχει την τιμή 300” (το όνομα του attribute θα περιείχε και την τελεία). Στην xaml ο τελεστής dot δεν είναι απλά μέρος αλφαριθμητικού αλλά χρησιμοποιείται σε μια σύνταξη που παραπέμπει σε αντικειμενοστραφή προγραμματισμό. Για παράδειγμα το κομμάτι κώδικα xaml στην

αρχή της 5.6.2 μεταφράζεται από το runtime component σε ένα σύνολο από εντολές:

Κατασκεύασε ένα αντικείμενο Canvas
Κατασκεύασε δύο αντικείμενα Rectangle
Θέσε τις τιμές των properties τους με βάση τα attributes στο xaml αρχείο
Συνέδεσε τα αντικείμενα Rectangle με το Canvas καθώς αυτά βρίσκονται εσωτερικά του στο xaml αρχείο

Δηλαδή κάθε xml element αντιστοιχεί στην κατασκευή ενός αντικείμενο. Οι αναθέσεις στα attributes ενός element αντιστοιχούν σε καθορισμό των properties του αντικειμένου του element..

Κατά την περιγραφή με xml μιας ιεραρχίας αντικειμένων C# παρουσιάζεται το πρόβλημα ότι κάποια αντικείμενα δεν μπορούν να περιγραφούν με ένα αλφαριθμητικό string. Όταν κάποιος τύπος (κλάση) αντικειμένων C1 διαθέτει έναν constructor με μοναδικό όρισμα ένα string, τότε στο xaml αρχείο με την σύνταξη

```
<AnotherClass name="anotherobject" C1Object="value">
```

Κατασκευάζεται ένα αντικείμενο της κλάσης AnotherClass, ανατίθεται σε μια μεταβλητή με όνομα anotherobject (το attribute name έχει αυτήν την επιπλέον ειδική σημασιολογία του ορισμού μιας named μεταβλητής), και στο τέλος στο property του C1Object ανατίθεται ένα αντικείμενο τύπου C1 που κατασκευάζεται με κλήση του constructor της C1 με το όρισμα "value".

Αν όμως η C1 δεν είχε τέτοιον constructor (η κάποιον μετατροπέα βλ. βιβλιογραφία) τότε η παραπάνω σύνταξη δεν θα μπορούσε να λειτουργήσει. Το κύριο επιπλέον σε σχέση με την XML στοιχείο της XAML είναι ότι σε αυτές τις περιπτώσεις επιτρέπει τον ορισμό του xml attribute με διαφορετικό τρόπο. Δηλαδή.

```
<AnotherClass name=”anotherobject”>
     <AnotherClass.C1Object>
          <C1FieldClass att1=”string2”/>
          <C1FieldClass2 att2=”string2”/>
     </AnotherClass.C1Object>
</AnotherClass>
```

Η παραπάνω δήλωση δεν σημαίνει δημιούργησε ένα αντικείμενο τύπου “ AnotherClass.C1Object” και θέσε το στο εσωτερικό του αντικειμένου anotherobject. Επειδή γίνεται χρήση του τελεστή dot το AnotherClass.C1Object έχει την σημασία “το property του AnotherClass που έχει το όνομα C1Object”. Στην συνέχεια μέσα στο element του AnotherClass.C1Object ορίζεται το C1Object με την κανονική σύνταξη XAML ορισμού αντικειμένων και όχι μέσα από ένα string. Εν τέλει τα εσωτερικά της C1 αντικείμενα C1FieldClass1 και C1FieldClass2 ορίζονται με strings (ώστε να μην είναι ατέρμων αναδρομή ο ορισμός)

Το κλείσιμο ενός element B μέσα στο element A, όταν δεν γίνεται χρήση της property like σύνταξης με το dot operator, σημαίνει ότι κάποιο πεδίο του αντικειμένου του A δείχνει στο αντικείμενο του B (το πεδίο καθορίζεται implicitly ανάλογα με τον τύπο του element, πχ αν το A έχει μια λίστα Childs η εμφώλευση του B το τοποθετεί στην Childs, αν το A είναι πίνακας, το B θα μπεί στην επόμενη διαθέσιμη θέση κτλ. Οι ορισμοί όπως το σε τι αντιστοιχεί η εμφώλευση xml elements αλλάζουν ανάλογα με την εκάστοτε υλοποίηση)

Υπάρχουν και άλλες λεπτομέρειες για το XAML το θέμα πάντως είναι ότι μπορεί να δημιουργήσει αντικείμενα WPF και από εκεί και πέρα οι υπόλοιπες δεν έχουν τόση σημασία για αυτήν την εφαρμογή.

Ο λόγος για τον οποίο γίνεται η χρήση του XAML στο WPF είναι το ότι διευκολύνει τον ορισμό και την αρχικοποίηση των στοιχείων ενός GUI. Μιας και το WPF είναι αντικειμενοστραφές ο ορισμός του GUI θα μπορούσε να γίνει σε κώδικα με κλήση

κατάλληλων constructors για κάθε γραφικό στοιχείο του περιβάλλοντος. Το θέμα όμως είναι ότι αυτόν τον ορισμό του τι θα υπάρχει στο GUI είναι καλύτερο να τον κάνουν γραφίστες και όχι προγραμματιστές. Οι γραφίστες δεν έχουν γνώσεις προγραμματισμό σε C# για να κάνουν την αρχικοποίηση των αντικείμενων.
Κατά συνέπεια ο γραφικός σχεδιασμός του GUI και ο προγραμματισμός του δεν μπορούν να γίνουν ανεξάρτητα και αυτό επιβραδύνει την όλη διαδικασία. Ταυτόχρονα η ανάμειξη του business tier κώδικα με τον κώδικα δημιουργίας του GUI δημιουργεί περιττές εξαρτήσεις και μειώνει την ευελιξία και την ευκολία αλλαγών. Το παραπάνω αποτελεί το λεγόμενο πρόβλημα του seperation of concerns ανάμεσα στον κώδικα για την παρουσίαση και τον υπόλοιπο κώδικα της εφαρμογής.

Μέσω του XAML τα σχεδιαστικά εργαλεία που χρησιμοποιούν οι γραφίστες συνθέτουν αυτόματα ένα XAML αρχείο που αρχικοποιεί τα αντικείμενα του interface. Το αρχείο αυτό χρησιμοποείται από τον υπόλοιπο κώδικα για να γίνει στο run time η κατασκευή του GUI. Με τα εργαλεία σχεδιασμού ορίζονται η διάταξη η μορφή και τα ονόματα των gui elements. Τότε μέσω του αρχείου xaml τα gui elements γίνονται απ ευθείας διαθέσιμα στον κώδικα σαν μεταβλητές αναφορές με αναγνωριστικό το όνομα τους. Για παράδειγμα αν ο σχεδιαστής GUI τοποθετήσει ένα Button με όνομα bt1, τότε αυτό θα είναι διαθέσιμο στον κώδικα σαν μεταβλητή Button bt1;

Πάντως το XAML δεν είναι τεχνολογία γραφικών. Χρησιμοποιείται κυρίως σε συνδυασμό με το WPF, όμως οι δυνατότητες του στην αρχικοποίηση αντικειμένων μπορούν να βρουν χρήση και σε άλλες εφαρμογές πχ στο ορισμό Workflows (στο Windows Workflow Ffoundation), σε Dependency Injection σε Application Servers κ.α.

5.6.2.2 WPF

Το WPF (Windows Presentation Function) είναι τεχνολογία της Microsoft για την δημιουργία GUI σε Windows. Αντικαθιστά τα παλαιότερα συστήματα γραφικών του λειτουργικού και δεν χρησιμοποιεί bitmaps αλλά μόνο διανυσματικά γραφικά. Είναι αντικειμενοστραφής και λειτουργεί με την χρήση αντικειμένων που αρχικοποιούνται σε κώδικα η μέσω xaml.

Δεν χρειάζεται να να αναφερθούν για αυτήν περισσότερα. Η κατασκευή του GUI βασίστηκε στην λογική της μετατροπής του αρχείου xml που αναπαριστά την δομή σε ένα αρχείο xaml που περιγράφει αντικείμενα του WPF. Στην συνέχεια το WPF runtime διαβάζει το xaml αρχείο και ζωγραφίζει στην οθόνη την δομή με βάση αυτό.

5.6.3 XSLT/XPATH

Η μετατροπή της δομής από αντικείμενα C# σε κείμενο xml έγινε με συναρτήσεις ToXML() που κατασκευάζουν σαν string το κείμενο. Θα μπορούσε να γίνει και με ειδικές για αυτόν τον σκοπό βιβλιοθήκες xml serialization.

Ο μετασχηματισμός από αυτό το xml κείμενο σε κάποιας άλλης μορφής xml κειμένου, πχ xaml/ wpf θα χρειαζόταν boilerplate (κατά κάποο τρόπο “γραφειοκρατικό”) και σχετικά δύσκολο κώδικα που θα έπρεπε να διαβάσει και να κάνει parse το input xml κείμενο, να το μετατρέψει σε αντικείμενα C#, και διασχίζοντας το δένδρο που θα προέκυπτε να ξαναεκτυπώσει το ouput xml document με τις σχετικές κατάλληλες αλλαγές..

Αυτό το είδος επεξεργασίας όμως μπορεί να κάνει καλύτερα η xslt που είναι μια γλώσσα μετασχηματισμού xml κειμένων. Με αυτήν δεν χρειάζεται ο χρήστης να σχεδιάσει αναδρομικές συναρτήσεις και περίπλοκες διαδικασίες output καθώς η γλώσσα είναι βασισμένη στο template matching και χειρίζεται την αναδρομή από μόνη της

Όταν η επεξεργασία έχει τα χαρακτηριστικά του μετασχηματισμού περιγράφεται καλά με το functional και το pattern matching μοντέλο προγραμματισμού. Θεωρητικά ως μετασχηματισμός μπορούν να περιγραφούν όλων τον μορφών οι διαδικασίες επεξεργασίας. Πρακτικά όμως μετασχηματισμός θεωρείται κάποια διαδικασία που δεν τροποποιεί την ίδια την δομή αλλά για κάθε στοιχείο της δίνει και κάποιο output. Δρα δηλαδή σαν συνάρτηση που αντιστοιχεί το πεδίο ορισμού στο πεδίο τιμών. Η διαφορά ανάμεσα στις δύο περιπτώσεις είναι ότι στην δεύτερη δεν χρειάζονται αναθέσεις μεταβλητών (αναθέσεις με την έννοια απόδοσης νέας τιμής στην μεταβλητή μετά την αρχικοποίηση της). Η δομή δεν αλλάζει κατάσταση μέσω τροποποίησης των θέσεων μνήμης που δείχνουν οι μεταβλητές της, αλλά εφαρμόζονται σε αυτήν συναρτήσεις που μετασχηματίζουν σταθερά ορίσματα στο αποτέλεσμα της εξόδου. Αυτό ονομάζεται side effects free execution. Λόγω της απουσίας "κατάστασης" δεν έχει σημασία η σειρά που εφαρμόζονται οι συναρτήσεις και έτσι μπορούν να εκτελεστούν παράλληλα.(Για παράδειγμα στην XSLT τα templates μπορούν να ελέγχονται και να εκτελούνται παράλληλα) Η XSLT είναι σχεδιασμένη για να υποστηρίζει τέτοιου είδους επεξεργασία.

Αντίθετα άλλων τύπων λειτουργίες δεν είναι κατάλληλες για υλοποίηση σε XSLT γιατί στηρίζονται στην αλλαγή κατάστασης κάποιας δομής από ακολουθιακό αλγόριθμο. Αυτό εκφράζεται καλύτερα με την τροποποίηση μεταβλητών. Για παράδειγμα λειτουργίες όπως το insert του kdb tree είναι περίπλοκες στην υλοποίηση και αργές σε χρόνο εκτέλεσης αν χρησιμοποιηθεί η XSLT. Εκτός αυτού, από πλευράς σχεδιασμού λογισμικού η XSLT έχει το επιπρόσθετο μειονέκτημα ότι δεν διαθέτει καλούς μηχανισμούς debugging που δυσκολεύει την ανάπτυξη ιδιαίτερα πολύπλοκων λειτουργιών.

Οπότε στην εργασία έγινε πρώτα η ανάπτυξη σε C# και στην συνέχεια εντοπίστηκαν οι περιπτώσεις λειτουργιών που μπορούσαν να περιγραφούν ως μετασχηματισμοί ώστε εκεί να γίνει η χρήση XSLT. Αν και η xslt είναι πιο αργή από την C# (όταν δεν γίνεται παράλληλη επεξεργασία) θεωρήθηκε ότι ο σκοπός δεν ήτανε η ταχύτητα σε

χρόνο εκτέλεσης αλλά η ταχύτητα ανάπτυξης, και για συγκεκριμένες λειτουργίες η XSLT υπερτερούσε σε αυτόν τον τομέα.

Η XSLT βασίζεται στο pattern matching των xml elements του input xml document, σε templates, τα οποία εκτυπώνουν xml στο αρχείο εξόδου, εκκινούν ξανά αναδρομικά το pattern matching κτλ.

.

Xml element θεωρείται το περιεχόμενο ανάμεσα σε ένα starting και ένα ending xml tag. Ένα xml element μπορεί να περιέχει άλλα xml elements αν αυτά είναι ανάμεσα στα tags του. Μπορεί κανείς να σκεφτεί τα templates της xslt ως συναρτήσεις και τα patterns ως κριτήρια για το σε ποια xml elements θα εφαρμοστεί το κάθε template

Τα templates στην xslt ορίζονται με την ακόλουθη σύνταξη

```
<xsl:template match="apattern">
[..]
</xsl:template>
```

Το πρόγραμμα XSLT είναι και αυτό xml document (μπορούν προγράμματα xslt να μετατρέπουν άλλα προγράμματα xslt). Το κάθε template αποτελεί και ένα xml element του namespace της xslt, το οποίο εδώ έχει το alias xsl. Στην θέση του appattern τοποθετείται μια έκφραση XPATH, Η XPATH είναι κάτι αντίστοιχο της SQL αλλά για xml δεδομένα. Επιτρέπει την δημιουργία queries που επιλέγουν xml στοιχεία.

Το XPath pattern της ιδιότητας match ενός template λειτουργεί ως εξής. Στο template T δίνονται ως όρισμα ένα σύνολο ΧΕ από xml elements (μέσω της κλήσης <apply-templates>). Το T ελέγχει αν τα ΧΕ μπορούν να περιγραφούν από το match pattern του, αν δηλαδή το xpath query του match pattern του T μπορεί να τα επιλέξει.. Η επεξεργασία μπορεί να προχωρήσει μόνο για όσα στοιχεία ισχύει αυτό.

Αν για κάποιο στοιχείο ισχύει ότι περισσότερα του ενός templates μπορούν να το επεξεργαστούν επιλέγεται μόνο αυτό με το πιο "εξειδικευμένο" match pattern (περισσότερα παρακάτω) Δηλαδή το κάθε στοιχείο επεξεργάζεται μόνο από ένα template**(1)**.

Πιο συγκεκριμένα αυτή η διαδικασία ξεκινάει με κάποια κλήση της xslt εντολής.

```
<xsl:apply-templates select="pattern"/>
```

Η κλήση αυτή γίνεται μέσα σε κάποιο άλλο template. Όπως τα templates είναι το XSLT αντίστοιχο του ορισμού συναρτήσεων, η εντολή apply-templates είναι το αντίστοιχο της κλήσης συναρτήσεων. Η διαφορά όμως είναι το ότι στην apply-templates δεν γίνεται επιλογή συγκεκριμένης συνάρτησης/template που θα εκτελεστεί, αλλά γίνεται επιλογή στοιχείων xml προς εκτέλεση.

Για κάθε ένα από αυτά τα στοιχεία επιλέγεται και ένα μόνο template από όλα τα διαθέσιμα με βάση το (1). Αν δεν υπάρχει κανένα template που το match pattern του να ταιριάζει με το στοιχείο αυτό δεν επεξεργάζεται. Έτσι η apply-templates δεν καλεί μια συγκεκριμένη συνάρτηση/template αλλά επιλέγει μια ξεχωριστή για κάθε στοιχείο xml το οποίο επιλέγεται για επεξεργασία.

Τα στοιχεία xml που επιλέγονται ορίζονται στο select pattern της εντολής apply-templates. Αυτό είναι και πάλι xpath query.

Μια λεπτομέρεια είναι η διατήρηση ενός "current" στοιχείου xml. Είναι κάτι ανάλογο της διατήρησης ενός current καταλόγου κατά την χρήση του δένδρου καταλόγων ενός λειτουργικού συστήματος. Όταν ένα template κάνει match και αρχίσει την επεξεργασία κάποιου στοιχείου E, τότε θέτει το E ως current στοιχείο. Το current element αλλάζει το νόημα των XPath queries κατά τον ίδιο τρόπο που το current directory του unix αλλάζει το νόημα των relative location specifiers.

Μια απλουστευμένη γενική περιγραφή της μορφής ενός τμήματος xpath query είναι

```
axis::ElementName[condition1][condition2]...[conditionn]/
```

Το τμήμα αυτό παρατίθεται συνεχόμενα n φορές. Στην αρχή της παράθεσης αυτών των τμημάτων υπάρχει είτε το σύμβολο / είτε δεν υπάρχει επιπλέον σύμβολο οπότε υπονοείται το σύμβολο . (dot)

Το / στην αρχή του Query έχει την σημασία της ρίζας του xml αρχείου, για την ακρίβεια η ρίζα του xml αρχείου είναι το ένα και μοναδικό παιδί του /. Αλλιώς χωρίζει τα τμήματα του Query μεταξύ τους.

Το . (dot) συμβολίζει τον τρέχοντα κόμβο, (current element), και επομένως αποκτά σημασία ανάλογα με το σε ποιο template ή xpath query γίνεται η χρήση του.

Το axis:: δηλώνει τον τρόπο επιλογής κόμβων με βάση το αποτέλεσμα του τμήματος του query που βρίσκεται αριστερά του axis:: . Δηλαδή, για παράδειγμα αν το αριστερό τμήμα είναι η ρίζα το axis:: μπορεί να πάρει τις εξής σημασίες

/child::Υπόλ.Query
Βρες τα παιδιά της ρίζας για τα οποία ισχύει το υπόλοιπο Query

/descendant::Υπόλ.Query
Βρες τους απογόνους της ρίζας για τους οποίους ισχύει το υπόλοιπο Query

/ancestor::Υπόλ.Query
Βρες τους προγόνους της ρίζας. για τους οποίους ισχύει το υπόλοιπο Query

Το τελευταίο προφανώς είναι άτοπο, όμως η σύνταξη επιτρέπεται και όταν δεν υπάρχει το / στην αρχή.

./ancestor::Υπόλ.Query
Βρες τους προγόνους του τρέχοντα κόμβου. για τους οποίους ισχύει το υπόλοιπο Query.

Προηγ.Query/ancestor::Υπόλ.Query
Για κάθε έναν από τους κόμβους του αποτελέσματος του Προηγ.Query βρες τους προγόνους τους για τους οποίους ισχύει το υπόλοιπο Query και στο τέλος συγκέντρωσε τα αποτελέσματα.

Το // αποτελεί συντόμευση του /descendant:: ενώ το / σκέτο και όταν δεν είναι στην αρχή του query αποτελεί συντόμευση του /child::

Το ElementName είναι ένα όνομα στοιχείου xml.

(Σημείωση ότι μπορεί στην θέση του ElementName να χρησιμοποιηθούν και μερικοί συμβολισμοί και wildcards που ταιριάζουν και σε περιεχόμενα του xml document που δεν είναι στοιχεία. Για παράδειγμα @att είναι η attribute att, Text() είναι το κείμενο ενός στοιχείου, Node() είναι ο οποιοσδήποτε κόμβος, * σαν το Node() με κάποιες διαφορές, Comment() ένα σχόλιο xml κτλ. Για αυτές τις λεπτομέρειες κοιτάξτε στην βιβλιογραφία.)

Όπως φαίνεται στο τελευταίο παράδειγμα, το δεξί κάποιου anchor τμήμα ενός query εφαρμόζεται ξεχωριστά "για κάθε" στοιχείο του αποτελέσματος του αριστερού μέρους. Στο τέλος τα αποτελέσματα συγκεντρώνονται. Πιθανόν αυτό να είναι κάπως unintuitive καθώς ο προγραμματιστής ίσως έχει στο μυαλό την σημασιολογία "για όσα". Δηλαδή το δεξί τμήμα να κάνει ένα αντίστοιχο του sql-where και να περιορίζει τα στοιχεία του αριστερού. Αυτό στην xpath γίνεται με άλλον τρόπο και συγκεκριμένα με την σύνταξη

[condition1][condition2]...[conditionn]

Εδώ δίνονται ν συνθήκες που εφαρμόζονται ακολουθιακά στο σύνολο κόμβων του αριστερού της σύνταξης τμήματος, κόβοντας όσα στοιχεία δεν τις πληρούν. Οι συνθήκες μπορεί να είναι διάφορων εξειδικευμένων τύπων. Ένας τύπος συνθήκης είναι τα νέα αναδρομικά xpath queries που δέχονται ως τρέχοντα κόμβο αυτόν που ελέγχεται, και που τον κόβουν αν η εκτέλεση τους δώσει το κενό σύνολο. Για παράδειγμα

/Book/Chapter[Paragraph/Random]

Το παραπάνω query πρώτα βρίσκει το σύνολο S από κόμβους που είναι στοιχεία Chapter τα οποία είναι παιδιά στοιχείου Book το οποίο είναι η ρίζα του xml document. Στην συνέχεια εφαρμόζει την συνθήκη θέτοντας ως current node το εξεταζόμενο στοιχείο του S. Αφού το query εσωτερικά της συνθήκης δεν αρχίζει με / υπονοείται το ./ . Άρα η σημασία του είναι βρες στοιχεία Random που είναι παιδιά ενός στοιχείου Paragraph που να είναι παιδί του εξεταζόμενου κόμβου. Αν αυτό το query δεν βρει στοιχεία τότε το εξεταζόμενο στοιχείο αφαιρείται από το S. Όταν εξεταστούν όλα τα αρχικά στοιχεία του S επιστρέφεται ότι έχει μείνει σε αυτό από την παραπάνω διαδικασία.

Άλλα παραδείγματα queries είναι τα εξής

/Book/Chapter[3]
Βρες το 3ο στοιχείο από τα στοιχεία Chapter που είναι παιδιά της Book ρίζας του xml document. Η σειρά των στοιχείων είναι η σειρά εμφάνισης των starting tags στους στο xml document (προκύπτει από το document order). Έτσι όλα τα στοιχεία του κειμένου είναι σε μια διάταξη. Κατά την τοποθέτηση στοιχείων σε μια συλλογή, αυτά ταξινομούνται με βάση αυτή τη διάταξη και έτσι μπορεί κανείς να κάνει λόγο για το ν-οστό στοιχείο της συλλογής.

/Book/Chapter[@pages<50][Random][3]

Από τα Chapters που είναι παιδιά ενός Book που είναι η ρίζα, όσα έχουνε λιγότερες από πενήντα σελίδες και μόνο αν έχουν παιδί ένα στοιχείο Random, και από αυτά μόνο το τρίτο σε σειρά.
(Εδώ φαίνεται η αναφορά σε attributes με την χρήση του @.)

* * *

Στην εργασία τα μόνα queries που χρειάστηκαν ήτανε τα

//Region[@level=”value”]
Βρες τα regions σε συγκεκριμένο επίπεδο του δένδρου.

//Point
Βρες όλα τα στοιχεία Point όπου και να είναι στο δένδρο

Τα queries αυτά καλούνται με apply-templates μέσα σε ένα template με match pattern το /. Επομένως για current node είχανε πάλι και αυτά το /.

6. ΥΛΟΠΟΙΗΣΗ

Η γλώσσα υλοποίησης του προγράμματος ήταν η C# γιατί συγκέντρωνε πολλές από τις τεχνολογίες που αναφέρθηκαν στην ενότητα 5 και που θα βοηθούσαν την διαδικασία ανάπτυξης

6.1 Κλάσεις

Οι βασικές κλάσεις στο πρόγραμμα είναι οι Node, RegionNode, PointNode, Region και Point. Επιπλέον η κλάση KDB_Tree λειτουργεί ως ή Main Class όπου αρχικοποιείται η δομή και οργανώνονται και εκτελούνται οι δοκιμές σε αυτήν.

Οι RegionNode και PointNode είναι υποκλάσεις της Node.

Η RegionNode αντιστοιχεί στα region pages της δομής, η PointNode στα point pages, και η Node είναι μια αφαίρεση που οργανώνει όλες τις λειτουργίες που γίνονται και στους δύο τύπους pages

.Η Point αναπαριστά ένα ν διάστατο σημείο.

Η Region αναπαριστά 2 πράγματα (κάτι που σχεδιαστικά είναι ίσως λάθος άλλα θεωρήθηκε πιο εύκολος και φυσικός ο σχεδιασμός με αυτόν τον τρόπο) πρώτα ένα ν διάστατο παραλληλόγραμμο και δεύτερον, καθώς κρατάει και δείκτη σε κάποιο Node, αντιπροσωπεύει και μια καταχώριση σε subregionlist.

Η RegionNode χρησιμοποιείται και στον καθορισμό των region queries. Τα regions των region queries έχουνε μια διαφορά με αυτά των subregionlists στο ότι περιλαμβάνουν και τα "δεξιά" όρια τους. Η κλάση Region μπορεί να παραστήσει και τους δύο τύπους regions.

6.2 Data members

Παρακάτω δίνονται αποσπάσματα κώδικα C# που δείχνουν την δομή που έχουν τα αντικείμενα των κλάσεων της εφαρμογής. Δείχνουν δηλαδή πως υλοποιείται στο πρόγραμμα το εσωτερικό των κόμβων που ως τώρα είχε περιγραφεί θεωρητικά.

```
abstract class Node
  {
    public static Node root;
    public static Stack<RegionNode> parentStack;
    [...]
    public static int SIZE = 4;
    public static int SIZEP = 4;
    public int splitd { get; set; }
    [...]
  }
```

Η κλάση Node είναι abstract και επομένως δεν γίνεται απευθείας χρήση αντικειμένων Node αλλά μόνων αντικειμένων RegionNode και PointNode. Δεν έχει μέσα δομή από data members παρά μόνο κάποια static fields(στην ουσία global μεταβλητές) και κάποια βοηθητικά αντικείμενα που θεωρούνται στο πρόγραμμα globals.

Η σταθερά SIZE είναι το μέγιστο όριο subregions σε region page πριν γίνει overflow. Αντίστοιχα η SIZEP είναι το μέγιστο όριο points σε point page.

Η property splitd που κληρονομείται στις υποκλάσεις δείχνει την επόμενη κατά σειρά διάσταση που πρέπει να διασπαστεί ο κόμβος. Οι διαστάσεις παριστάνονται με αριθμούς int 0 εως n. Το πεδίο splitd μεταβάλλεται κατά την διάσπαση κόμβων με απόδοση της επόμενης τιμής διάστασης στους νέους κόμβους.

Το splitd μπορεί να αγνοηθεί αν ο αλγόριθμος διάσπασης διαπιστώσει ότι πρέπει να γίνει σε άλλη διάσταση το split.

Η μεταβλητή root αναπαριστά την ρίζα του δένδρου. Τέλος η parentStack είναι η στοίβα από όπου ο κάθε κόμβος βρίσκει ποιος είναι ο γονέας του και η οποία μεταβάλλεται κατάλληλα καθώς οι συναρτήσεις κάνουν Depth First Search like διάσχιση του δένδρου

```
public  class Point
  {
    public float[] x=new float[Region.DC];
    [...]
  }
```

Η κλάση point υλοποιείται με ένα πίνακα float x μεγέθους ίσου με το πλήθος διαστάσεων.

Το Region.DC είναι μια static μεταβλητή της κλάσης Region που μετρά το συνολικό πλήθος διαστάσεων.

Στην εφαρμογή η υλοποίηση γίνεται με την χρήση floats αλλά και άλλοι τύποι θα μπορούσαν να χρησιμοποιηθούν. Αν η εφαρμογή ήταν για ευρεία χρήση και όχι μόνο για δοκιμές θα έπρεπε η κλάση Point, όπως και οι υπόλοιπες, να είναι generic και με στατικό όρισμα να επιλέγεται ο τύπος των πεδίων.

```
class Region
  {
    public static int DC=2;
    public static float MIN = 0;
    public static float MAX = 500;

    public float[] min = new float[DC];
    public float[] max = new float[DC];
    public Node n { get ;set;   }
    [...]
   }
```

Η κλάση Region έχει static fields (στην ουσία globals) που δείχνουν το πλήθος των διαστάσεων (DC) και τα όρια του χώρου σε κάθε μια (MIN, MAX).

Τα διαστήματα του παραλληλογράμμου σε κάθε διάσταση περιγράφονται με την χρήση δύο πινάκων float όπου ο min έχει για κάθε μια από τις διαστάσεις το ελάχιστο όριο και ο max το μέγιστο.

Η property n δείχνει τον κόμβο που αντιστοιχεί το παραλληλόγραμμο όταν αυτό είναι μέρος καταχώρισης στην subegionlist ενός region page.

```
class PointNode:Node
  {
    [...]
    public List<Point> points { get; set; }
    [...]
  }
```

Τα δεδομένα ενός PointPage είναι μια List από points..

```
class RegionNode : Node
  {
    [...]
    public List<Region> subr {[...] }
    [...]
  }
```

Η RegionNode περιέχει μια List από Regions. Επίσης περιέχει μερικές μεταβλητές flags σχετικές με το debugging που δεν εξηγείται εδώ τι κάνουν.

Έχουν γίνει μερικές όχι και τόσο καλές επιλογές στον σχεδιασμό που όμως προτιμήθηκαν γιατί αλλιώς το πρόγραμμα θα γινόταν δυσνόητο.

Πρώτα από όλα η χρήση static fields για την αποθήκευση των παραμέτρων του KDBTree όπως τα fields Node.SIZE ή Region.DC, Region.MAX δεν είναι καλή ιδέα αν το πρόγραμμα είναι φτιαγμένο για γενική χρήση.

Λειτουργούν ως global fields, κάτι που αποτρέπει την δημιουργία πολλών τύπων kdbtrees (διαφορετικών διαστάσεων και χώρων) στο ίδιο πρόγραμμα.

Κανονικά κάθε κατασκευή Region Node κτλ θα έπρεπε να συνοδεύεται από προσδιορισμό του πλήθος των διαστάσεων κτλ μέσω κάποιας δομής που να περιέχει όλες τις παραμέτρους (KDBTreeParameters), και για να μην γίνεται συνεχής επανάληψη της μεταφοράς αυτής της δομής κανονικά τα Regions θα έπρεπε να κατασκευάζονται με factories. (RegionFactory.CreateRegion() αντί για new Region(), όπου η κλάση RegionFactory θα διέθετε κατάλληλο πεδίο προσδιορισμού των kdb tree παραμέτρων)

Ως έχει οι παράμετροι αποθηκεύονται σε static fields που ρυθμίζονται με μια static συνάρτηση KDBTree.Init() που περιγράφεται πιο μετά. Έτσι δεν μπορούν να δημιουργηθούν πολλά objects KDBTree ταυτόχρονα με διαφορετικό πχ τύπο διαστάσεων καθώς αυτός είναι global. Κρίθηκε όμως ότι η σωστή από αυτήν την πλευρά αρχιτεκτονική θα έκανε πολύ πολύπλοκη την κατανόηση του τι κάνει το πρόγραμμα για κάποιον που δεν έχει ξοδέψει χρόνο να καταλάβει την λειτουργία του. Οπότε για εκπαιδευτικούς σκοπούς προτιμήθηκε η χρήση των globals.

6.3 Μέθοδοι

Στο επόμενο επίπεδο λεπτομέρειας παρουσιάζονται οι συναρτήσεις των κλάσεων

Στην κλάση Node υπάρχουν κυρίως ορισμοί abstract μεθόδων που υλοποιούνται στις κλάσεις RegionNode και PointNode, ορισμοί μεθόδων που είναι κοινές και στις δύο περιπτώσεις. και βοηθητικές μέθοδοι.

```
abstract class Node
{
[...]
public static void Up(StackDelegate sd)[...]
public static void Down(RegionNode w,StackDelegate sd)[...]
public void split(RegionNode parent)[...]
public abstract void split(int d, float v, RegionNode parent)[...]
protected virtual int ChooseDimension() [...]
public abstract float ChooseValue()[...]
public abstract List<Point> Query(Region r)[...]
public abstract void Insert(Point p)[...]
 public abstract string ToXml()[...]
[...]



public Region Combine(IEnumerable<Region> regs)[...]
public abstract UtilizationResults Utilization()[...]
public abstract void CountNodesPerLevel()[...]
}
```

Οι συναρτήσεις Up και Down χρησιμοποιούνται για τον έλεγχο της στοίβας και επεξηγούνται παρακάτω.

Οι συναρτήσεις για την διάσπαση είναι οι

1) split(RegionNode parent) και
2) split(int d, float v, RegionNode parent)

Η 1 χρησιμοποιείται για split που δεν είναι forced. Συγκεκριμένα υλοποιείται στην υπερκλάση Node και η δουλειά της είναι να καλεί την ChooseDimension(), η οποία αποφασίζει σε ποια διάσταση θα γίνει το split, και την ChooseValue(), που προσδιορίζει ποιο υπερεπίπεδο κάθετο σε αυτήν την διάσταση θα χρησιμοποιηθεί.

Στη συνέχεια παραδίδει αυτές τις αποφάσεις στην δεύτερη overloaded μέθοδο split. Η 2 με βάση τις αποφάσεις που λαμβάνει από την 1 εκτελεί την υπόλοιπη διάσπαση και κυρίως το data handling (αντικαταστάσεις, αντιγραφές σημείων και regions, εκκίνηση του forced split, δημιουργία νέων κόμβων κτλ.) Το forced split γίνεται με την 2 καθώς σε αυτό δεν γίνεται η επιλογή υπερεπιπέδου που εκτελεί η 1.

Οι ChooseValue(), ChooseDimension() και η split(int d, float v, RegionNode parent) υλοποιούνται στις υποκλάσεις RegionNode και PointNode καθώς οι λειτουργίες τους εξαρτώνται από τον τύπο page και δεν είναι κοινές σε region και point pages.

Η συνάρτηση Query(Region) ψάχνει όλα τα σημεία μέσα στο δοθέν όρισμα της που είναι το παραλληλόγραμμο του range query.

Η Insert εισάγει σημείο στην δομή.

Λόγω του ότι αυτές οι συναρτήσεις λειτουργούν αναδρομικά, στο πρόγραμμα η κλήση τους γίνεται πάντα μόνο στον κόμβο ρίζα του kdb tree. Αν κληθούν σε κάποιο άλλο Node πλην της ρίζας θα εκτελεστούν μόνο στο υποδένδρο αυτού του Node και τα αποτελέσματα θα είναι λάθος.

Η ToXml() κατασκευάζει αναδρομικά μια string περιγραφή της δομής σε XML και πρέπει και αυτή να καλείται μόνο στο root του kdb tree.

Οι υπόλοιπες 3 συναρτήσεις είναι βοηθητικές κι χρησιμοποιούνται για το debugging και για την διενέργεια συλλογής στατιστικών. Η Utilization() μετρά το utilization των κόμβων. Ή Combine() συνδυάζει με πρόχειρο αλγόριθμο τις περιοχές των ορισμάτων της ώστε να ελεγχθούν αν είναι ίσες με κάποιο άλλο Region. Η CountNodesPerLevel() συλλέγει τα στατιστικά για το πόσα nodes υπάρχουν στο κάθε επίπεδο του kdb tree.

Η κλάση PointNode δεν έχει επιπλέον συναρτήσεις πέρα από όσες abstract της Node υλοποιεί. Η RegionNode διαθέτει μερικές επιπλέον.

```
class RegionNode : Node
  {
        public void Substitute(Node rem, Region ileft, Region iright, int dimension)
        public void Remove(Node node)
        public Region getRegion(Node node)
        public void Insert(Region r)
        private Cut ChooseCut()
  }
```

Η Insert(Region) και η Remove(Region) χρησιμοποιούνται για την εισαγωγή και διαγραφή καταχωρήσεων στην subregionlist του κόμβου. Οι δύο αυτές συναρτήσεις δεν χρησιμοποιούνται απευθείας καθώς η εισαγωγή και η διαγραφή regions γίνονται πάντα μαζί, μετά από κάποια διάσπαση, και πάντα σαν ομάδα από μια διαγραφή και δύο εισαγωγές.(μόνο αν το πρόγραμμα υλοποιούσε και διαγραφές σημείων θα υπήρχαν και άλλα patterns εκτέλεσης αυτών των συναρτήσεων)

Η Substitute() εκτελεί την ομάδα εισαγωγών και διαγραφών regions που χρειάζονται κατά την διάσπαση και διενεργεί επιπλέον ελέγχους debugging. Οι έλεγχοι αυτοί δεν μπορούσαν να τοποθετηθούν ξεχωριστά στις Insert() και Remove() καθώς έχουνε να κάνουνε με το όλο transaction των 2 inserts και ενός remove της διάσπασης.

Η getRegion() χρησιμοποιείται για την εύρεση από κάθε κόμβο της region στην οποία είναι καταχωρημένος τον γονέα του. Δεν συνδυάζονται τα subregions του κόμβου για να προκύψει η region γιατί αυτό θα ήτανε και επικίνδυνο και θα δυσκόλευε την αποσφαλμάτωση. Αν η δομή είχε μέσα παραβιάσεις του invariants, ένας πρόχειρος συνδυασμός των subregions δεν θα τις αντιλαμβανόταν, και έτσι θα συνεχιζόταν η λανθασμένη εισαγωγή σημείων χωρίς να γίνει αντιληπτό νωρίς -fail

fast- το σφάλμα. Στο debugging πάντως γίνεται έλεγχος αν το Combine(subregions) είναι το ίδιο με το getRegion(this) ώστε να εντοπιστούν γρήγορα ορισμένα σφάλματα στην διαμέριση του χώρου).

Η τυπική κλήση της getRegion είναι της μορφής parentStack.peek().getRegion(this) η οποία βρίσκει στην στοίβα parentStack τον γονέα Γ του κόμβου Κ, και ζητάει από τον Γ δώσει στον Κ το region όπου τον έχει καταχωρημένο.

Τέλος η ChooseCut() είναι private συνάρτηση της RegioNode η οποία προϋπολογίζει τις τιμές των ChooseDimenion() Και ChooseValue() με έναν πιο πολύπλοκο αλγόριθμο.Ενδιάμεσα η ChooseCut() χρησιμοποιεί την Cubicality, που υπολογίζει έναν μέσο όρο του πόσο cubical είναι τα regions που προκύπτουν από κάποια προτεινόμενη διάσπαση, την EqualUtilization που υπολογίζει κατά πόσο ένα υπερεπίπεδο χωρίζει με ίση κατανομή το σύνολο των καταχωρήσεων του subregionlist, και την Splits που υπολογίζει πόσα forced splits στα παιδιά του region node προκαλεί ένα υπερεπίπεδο. Έτσι υπάρχει μια μέθοδος για κάθε ένα από τα κριτήρια της ώστε με κατάλληλη στάθμιση των αποτελεσμάτων να επιλεγεί το υπερεπίπεδο διαχωρισμού.

```
class Region
 {
public void split(int d, float val, out Region leftr, out Region rightr)
public bool ContainsPoint(Point p)
public bool Contains( int d, v)
public bool DividedBy(Cut c)
public override bool Equals(object obj)
public override bool Same(object obj)
public Cut[] getCuts()
public bool Intersects(Region r)
 public bool AxisIntersect(int i, Region r)
}
```

Στην κλάση Region υπάρχουνε οι εξής μέθοδοι. Η split χωρίζει regions, δηλαδή εκτελεί τον γεωμετρικό διαχωρισμό παραλληλεπιπέδων του kdb tree. Η εκτέλεση της split για κάποιο region r γενικά προηγείται της διάσπασης με split του κόμβου που δείχνει το r δηλαδή του r.n

Η ContainsPoint βρίσκει αν ένα σημείο βρίσκεται γεωμετρικά εντός του παραλληλεπιπέδου και χρησιμοποιείται ως ενδιάμεσο βήμα των αλγορίθμων Insert και Query.

Η ContainsPointQ είναι παραλλαγή της ContainsPoint() που χρησιμοποιείται στο Query. Η διαφορά της είναι ότι συμπεριλαμβάνει και τα σημεία στο δεξί boundary του region (καθώς τα regions των queries διαφέρουν σε αυτό ακριβώς από αυτά των subregionlists 4.1.3)

Η DividedBy() επιστρέφει τιμή αληθείας για το αν ένα Cut (που είναι struct που αντιπροσωπεύει έναν συνδυασμό διάστασης και θέσης σε αυτήν, δηλαδή ένα υπερεπίπεδο) τέμνει στο εσωτερικό (και όχι στο boundary) το region. Χρησιμοποιείται για να αποφασιστεί σε ποια nodes θα γίνει forced split.

Η Contains είναι όμοια της DividedBy() με την διαφορά ότι έχει διαφορετική συμπεριφορά στα όρια των regions ώστε να χαρακτηρίσει ένα region με τις τιμές -1,0,1 ανάλογα αν βρίσκεται αριστερά δεξιά ή ούτε αριστερά ούτε δεξιά του υπερεπιπέδου των ορισμάτων της. Χρησιμοποιείται για να μοιράσει τα nodes δεξιά και αριστερά κατά την διάσπαση.

Η getCuts() επιστρέφει όλα τα υπερεπίπεδα που τέμνουν τα όρια του region σε μορφή λίστας από δομές Cut. Αυτά χρησιμεύουν στην επιλογή υπερεπιπέδου διάσπασης από την συνάρτηση RegionNode.ChooseCut() η οποία ταξινομεί τα cuts με κάποια κριτήρια καταλληλότητας και επιλέγει το βέλτιστο.

Η Equals() και Same() ελέγχουν αν δύο regions δείχνουν την ίδια περιοχή στον χώρο με την Equals να ελέγχει επιπλέον αν και ο δείκτης n προς τον κόμβου τους δείχνει στον ίδιο κόμβο

Η Intersects() κάνει ν διάστατο collision detection ανάμεσα σε δύο regions, με την AxisIntersects() να είναι ένα ενδιάμεσο βήμα της για τον υπολογισμό collisions σε προβολές πάνω σε κάποιον άξονα (4.2.4.1)

Η Cubicality χρησιμοποιείται από την αντίστοιχη μέθοδο της RegionNode.

```
public  class Point

{
public string ToXml()
public override bool Equals(System.Object obj)
}
```

Η Point διαθέτει συναρτήσεις για xml αναπαράσταση και σύγκριση Points. Η σύγκριση γίνεται με έλεγχο των στοιχείων των πινάκων x τους ένα προς ένα

6.4 Περιγραφή κώδικα

6.4.1 Insert

Η Insert εισάγει σημεία. Η κλήση γίνεται στο root και συμμετέχουν πολυμορφικά η Insert της RegionNode και η Insert της PointNode

```
public override void Insert(Point p)
    {
      Region r;
      var count = subr.Where(sss => sss.ContainsPoint(p)).Count();
      if ((count > 1)||(count < 1)) throw new Exception();                    (1)
      r = subr.First(a => a.ContainsPoint(p));                                (2)

      Down(this, () => r.n.Insert(p));                                        (3)
      if ((subr.Count == SIZEP) ) { split(parentStack.Peek()); }              (4)
     }
```

Στην insert της RegionNode πρώτα γίνεται έλεγχος αποσφαλμάτωσης για το αν περισσότερα η λιγότερα του ενός subregions του κόμβου περιέχουν το σημείο. Αν συμβαίνει κάτι τέτοιο η διαμέριση είναι λανθασμένη και πρέπει να γίνει ειδοποίηση για το λάθος πριν αυτό προκαλέσει αλλαγές που θα κάνουν ακόμα δυσκολότερη την αποσφαλμάτωση (1). Στο (2) αφού εξασφαλίστηκε ότι ένα και μόνο ένα subregion περιέχει το point, αυτό επιλέγεται με μια εντολή Linq (η First επιστρέφει το πρώτο στοιχείο μιας συλλογής που ικανοποιεί την συνθήκη που δέχεται όρισμα με lambda function)

Στο (4) γίνεται ο έλεγχος για υπερχείλιση και η εκκίνηση της διάσπασης. Στο (3) η σύνταξη είναι λίγο περίεργη αλλά αυτό που γίνεται είναι να καλείται η insert στον κόμβο του region που επιλέχτηκε στο (2). Λόγω πολυμορφισμού η συνάρτηση Insert που καλείται είναι είτε της RegionNode είτε της PointNode (Δεν χρειάζονται δηλαδή έλεγχοι if για το τι τύπος είναι ο κόμβος)

Το Down(this, () => r.n.Insert(p)) περνάει στην μέθοδο Down της Node ένα lambda function που κάνει την κλήση. Ο λόγος για αυτό το ενδιάμεσο βήμα είναι το ότι η Down ρυθμίζει την στοίβα αυτόματα και έτσι μειώνεται ο κίνδυνος να γίνει κάποιο λάθος τον χειρισμό της

```
public static void Up(StackDelegate sd)
    {
      RegionNode w=parentStack.Pop();
      sd.Invoke();
      parentStack.Push(w);
    }

public static void Down(RegionNode w,StackDelegate sd)
    {
      par.pagesread++;
      parentStack.Push(w);
      sd.Invoke();
      parentStack.Pop();
    }
```

Αναλόγως υπάρχει και η Up. Η Down πρώτα τοποθετεί τον τρέχων κόμβο στην στοίβα ώστε ο υιός κόμβος στον οποίο γίνεται η κλήση να βρει τον γονέα του με parentStack.Peek(), μετά κάνει την κλήση και στο τέλος κατά την αποπεράτωση αυτής ξαναγυρνάει την στοίβα στην παλαιά κατάσταση.

Η Up αφαιρεί τον γονέα του τρέχοντα κόμβου από την στοίβα ώστε στην κορυφή της να βρίσκεται ο προηγούμενος γονέας. Έτσι μπορεί να δώσει με ασφάλεια την ροή ένα επίπεδο ψηλότερα (πχ κατά την κλήση της Substitute στις διασπάσεις)

Επιπλέον επειδή από την Down περνάνε όλες οι διασχίσεις προς τα κάτω του δένδρου, αυτή χρησιμεύει και ώστε να γίνει σε ένα κεντρικό σημείο καταμέτρηση στατιστικών όπως το πόσες σελίδες επισκέφτηκε ο αλγόριθμος.

```
public override void Insert(Point p)
      {
        if(points.Exists(a=>a.Equals(p)))return;                    (1)
        Program.log += "Inserted Point " + p + "\n";                (2)
        points.Add(p);                                              (3)
        if (points.Count > SIZE) {  split(parentStack.Peek());  }   (4)
        else par.pageswritten++;                                    (5)
      }
```

Όταν μετά από τα down η insert φτάσει στα φύλλα εκεί καλείται η PointNode.Insert() Αυτή στο (1) απαγορεύει τις διπλοκαταχωρήσεις όπως υπέθετε ο Robinson, στο (2) κάνει logging που χρησιμεύει στο graphical gui της εφαρμογής(ένα σημείο όπου χαλάει το cohesion και θα βοηθούσαν τεχνικές AOP.) Στο (3) απλά τοποθετείται το σημείο στο (4) γίνεται έλεγχος για split και στο (5) λαμβάνονται στατιστικά για το πλήθος σελίδων που θεωρητικά γράφτηκαν με I/O (πρακτικά δεν γίνεται χρήση blocks αλλά αντικειμένων της KM) Το par υπενθυμίζεται ότι είναι static μέλος της Node και είναι αυτό όπου καταγράφονται όλα τα στατιστικά. Επίσης το όρισμα της split δεν είναι ο κόμβος που πρόκειται να διασπαστεί αλλά ο γονέας αυτού του κόμβου.

6.4.2 Query

Η Query λειτουργεί αναλόγως με την Insert και καλείται από την ρίζα

```
public override List<Point> Query(Region r)
      {
        List<Point> result = new List<Point>();                          (1)
        subr.Where(s=>s.Intersects(r)).Each(ss=>result.AddRange(ss.n.Query(r)));   (2)
        return result;
      }
```

Στην Query των region nodes στο (1) ορίζεται η συλλογή όπου αποθηκεύονται τα

αποτελέσματα από την αναδρομική κλήση της Query σε υιούς του region node. Στο (2) επιλέγονται (Where) όσα regions έχουνε κοινό σημείο με αυτό του query και για το κάθε ένα (Each) καλείται η συνάρτηση αναδρομικά. Στο τέλος επιστρέφονται τα αποτελέσματα στο ανώτερο επίπεδο για συλλογή μέχρι που η ρίζα να επιστρέψει τα συνολικά αποτελέσματα. Οι κλήσεις των συναρτήσεων LINQ όπως η Where επιστρέφουν μετά από κάποιο transform στοιχεία του τελικού collection και μπορούνε για αυτό να μπούνε σε ένα chain εκτελέσεων όπως εδώ. Επίσης δεν γίνεται χρήση της στοίβας γιατί η διάσχιση στο δένδρο δεν πρόκειται να ανεβεί από υιό σε γονέα.

```
public override List<Point> Query(Region r)
    {
      List<Point> result = new List<Point>();
      result.AddRange(points.Where(p => r.ContainsPointQ(p)));
      return result;
    }
```

Στο Query ενός point node απλά συλλέγονται τα σημεία του point node που είναι μέσα στο όρισμα του region query. Γίνεται χρήση της ContainsPointQ και όχι της ContainsPoint.

```
public bool Intersects(Region r)
    {

      for (int i = 0; i < DC; i++)
      {
        if (!AxisIntersect(i, r))
          return false;
      }
      return true;
    }
```

Αυτή είναι η συνάρτηση που υλοποιεί το collision detection όπως αυτό είχε αναλυθεί μαθηματικά πιο πάνω. Παρατηρεί κανείς πως προσπαθεί να κάνει ένα λογικό and n συνθηκών και επειδή το n είναι δυναμικό, αυτό γίνεται με ένα for loop που προσπαθεί να ακυρώσει την υπόθεση ότι και η n συνθήκες ισχύουν.

Όπως περιγράφτηκε στον αλγόριθμο του colision detection η ν διάστατη περίπτωση ανάγεται σε ν ελέγχους της μονοδιάστατης οι οποίοι γίνονται από την AxisIntersect() η οποία είναι και trivial στην υλοποίηση. Σημειώνεται ότι γενικά ο for loop στο πλήθος των διαστάσεων της συνάρτησης αυτής επαναλαμβάνεται στις περισσότερες συναρτήσεις της Region καθώς είναι απαραίτητος αν η δομή θέλει να υποστηρίζει οποιοδήποτε πλήθος διαστάσεων.

Για παράδειγμα η ContainsPoint

```
public bool ContainsPoint(Point p)
    {
      for (int i = 0; i < DC; i++)                                    (1)
      {
        if (min[i] < max[i])                                          (2)
        {
         if (!(Contains(i, p.x[i]) == 0)) return false; }             (3)
         else if (min[i] == max[i]) { if (!(p.x[i] == min[i])) return false;   (4)
        }
     [..]
     return true;
    }
```

Στο (1) ξεκινάει τον βρόγχο των n διαστάσεων, στο (2) ελέγχει αν ισχύει ότι τα όρια είναι ίσα σε κάποια διάσταση (ώστε να αντιλαμβάνεται τα point queries και να μην παγιδεύεται από το μαθηματικό άτοπο που προκαλεί αυτή η ισότητα και αναλύθηκε σε άλλο μέρος της εργασίας). Στα (3) και (4) κάνει τον κατάλληλο κατά το (2) επιμέρους έλεγχο για την διάσταση που δείχνει
ο βρόγχος.

6.4.3 Split

Το κυριότερο μέρος της υλοποίησης ενός kdb tree είναι αυτό των μεθόδων για το split.

```
public override void split(int d, float v, RegionNode parent)
    {

      Program.log += "Split at level " + parentStack.Count+"\n";          (1)
      par.pageswritten += 2;                                              (2)
      Region left, right;

      parent.getRegion(this).split(d, v, out left, out right);            (4)

      var lp = from p in points where p.x[d] < v select p;                (5)
      var rp = from p in points where p.x[d] >= v select p;               (5)
      left.n = new PointNode(lp, splitd);                                 (6)
      right.n = new PointNode(rp, splitd);                                (6)

      Up(() => parent.Substitute(this, left, right, d));                  (7)
    }
```

Παραπάνω η μέθοδος split των PointNodes. Στο (1) γίνεται logging που χρησιμοποιείται από το GUI. Στο (2) γίνεται συλλογή στατιστικών για το πλήθος εγγραφών I/O. Η ιδέα είναι ότι κάθε forced split δίνει 2 εγγραφές, για τους 2 νέους κόμβους που δημιουργούνται, ενώ το split από υπερχείλιση δίνει μία εγγραφή καθώς γίνεται η υπόθεση ότι ο κόμβος θα μείνει στην KM μέχρι να αναπροσαρμοστεί όλο το υποδένδρο του οπότε και θα γίνει η μία εγγραφή του στο δίσκο..

Στο (4) μέσω της getRegion() λαμβάνεται από τον κόμβο γονέα (parent) η περιοχή region η οποία αντιστοιχεί στο pointnode. Η getRegion() έχει υλοποιηθεί κατάλληλα ώστε να επιστρέφει όλον τον χώρο αν το όρισμα της (το parent) είναι null που σημαίνει ότι το pointnode δεν έχει γονέα και είναι η ρίζα.

Στα (5) γίνεται με LINQ query η επιλογή από την λίστα σημείων του point node όσων βρίσκονται αριστερά και δεξιά του υπερεπιπέδου διαχωρισμού που δείχνουν τα d και v. Στα (6) γίνεται η δημιουργία των δύο νέων pointnodes που προκύπτουν από το split. Τα ορίσματα των constructors είναι οι λίστες που προέκυψαν στο (5).

Στο (7) εμφωλεύεται σε Up για ρύθμιση της στοίβας η εντολή που αντικαθιστά στον γονέα κόμβο το παλιό point node με τα δύο νέα. Το parent δεν γίνεται να είναι null γιατί κάθε κλήση αυτής της split με τα ορίσματα d και v σε pointnode έπεται της κλήσης της split με τα λιγότερα ορίσματα που υλοποιήθηκε στην υπερκλάση Node και η οποία αυτόματα προσθέτει έναν γονέα κόμβο σε περίπτωση που το split γίνεται στην ρίζα.

```
public void split(RegionNode parent)
    {

      if (parent == null)                                    (1)
      {
         root = parent = new RegionNode(-1);
      }
      split(ChooseDimension(), ChooseValue(), parent);       (2)
    }
```

Αυτή είναι η υλοποίηση της έκδοσης της split της Node που αποφασίζει την διάσταση d και την θέση του υπερεπιπέδου v πριν κάνει delegation της επεξεργασίας στην split των point pages ή των region pages. Αυτή η split αυτή καλείται πάντα μετά από κάποιο insert σημείου και όταν έχει διαπιστωθεί κάποιο overflow. Στο (1) δημιουργείται μια νέα ρίζα αν γίνεται split στην τωρινή και στο (2) καλείται κάποια overriden split αφού έχουν επιλεχθεί διάσταση και επίπεδο διαχωρισμού.

Η ChooseDimension() στα point pages έχει ως εξής

```
protected override int ChooseDimension()
        {
          int d = splitd;                                    (1)
          while (true)                                       (2)
            {
              var f=points[0].x[d];
              if (!points.All(p => p.x[d] == f)) return d;   (3)
              else d = (d + 1) % Region.DC;
            }
      }
```

Στο 1 αρχικά επιλέγεται η προεπιλογή διάστασης που είναι η τιμή του πεδίου splitd η οποία σε κάθε διάσπαση μεταβάλλεται κυκλικά από τον διασπώμενο στους δύο νέους κόμβους ώστε να διατηρηθούν κυβοειδή τα regions.

Υπάρχει όμως περίπτωση όλα τα σημεία να είναι ευθυγραμμισμένα ως προς αυτήν την διάσταση και να μην μπορεί να γίνει split που να μην δώσει κενά point pages. Για αυτό στο (2) ένας βρόγχος αλλάζει κυκλικά και με modulo n τρόπο την διάσταση μέχρι που στο (3) να βρεθεί κάποια για την οποία όλα τα σημεία να μην είναι ευθυγραμμισμένα.

Το (3) χρησιμοποιεί την LINQ συνάρτηση All που ελέγχει αν όλα τα στοιχεία μιας συλλογής επαληθεύουν την συνθήκη της lambda function που η All δέχεται ως όρισμα. Εδώ η συνθήκη ελέγχει αν όλα τα σημεία έχουνε την ίδια τιμή στην διάσταση d του βρόγχου στο (2)

η ChooseValue() έχει ως εξής

```
public override float ChooseValue()
      {
       int d = ChooseDimension();                (1)
      var a = points.OrderBy(p => p.x[d]);       (2)
       int mid = SIZEP / 2;                      (3)
       return a.ElementAt(mid).x[d];             (4)
     }
```

Στο (1) επιλέγεται ξανά διάσταση (κάτι που δεν έχει άλλο αποτέλεσμα από την κλήση ChooseDimension() στο Node.split() γιατί η ChooseDimension() είναι side effects free) στο (2) γίνεται ταξινόμηση των στοιχείων με LINQ ως προς την συντεταγμένη τους στην διάσταση που επιλέχθηκε, στο (3) εντοπίζεται η θέση του μέσου σημείου στην ταξινομημένη λίστα και στο (4) η τιμή αυτού επιστρέφεται ως θέση του υπερεπιπέδου διαχωρισμού στην διάσταση διάσπασης

Η εφαρμογή του design by contract για την split της κλάσης PointNode είναι ελλιπής καθώς θεωρήθηκε ότι η συνάρτηση αυτή δεν θα προκαλέσει πολλά προβλήματα όπως η split των region nodes. Θα έπρεπε να γίνονται περισσότεροι έλεγχοι για τα preconditions και τα postconditions

* * *

Το split της RegionNode ήτανε το σημείο της εφαρμογής που προκαλούσε τα περισσότερα προβλήματα σχετικά με την αποσφαλμάτωση. Για αυτό χρησιμοποιήθηκαν έλεγχο κατά το diesign by contract ώστε να γίνεται έλεγχος των invariants, preconditions και postconditions με σκοπό να προσδιορίζονται τα λάθη.

Η αρχή και το τέλος της συνάρτησης περιέχουν εντολές για αυτό το σκοπό. Αυτές έχουν την μορφή Debug.Assert(condition,description) όπου το condition είναι μια

συνθήκη ελέγχου και το description η περιγραφή του τι σφάλμα καταδεικνύει αυτή.

Αν η συνθήκη δεν ισχύει το πρόγραμμα τερματίζει με exception και παρουσιάζει στον χρήστη την περιγραφή του σφάλματος. Παραθέτω εδώ όλους τους ελέγχους χωρίς επιπλέον σχολιασμό γιατί οι περιγραφές των description εξηγούν τον κάθε ένα από μόνες τους.

Σημειώνεται πάντως ότι υπάρχουνε και μεταβλητές όπως οι oldcount, splitting που κρατάνε δεδομένα χρήσιμα για τους ελέγχους πχ το oldcount αποθηκεύει το πλήθος subregions κατά την εκκίνηση της συνάρτησης ώστε αυτό να συγκριθεί στο τέλος της με το νέο πλήθος subregions και να διαπιστωθεί αν έγινε σφάλμα, συγκεκριμένα εδώ αν η αύξηση του πλήθους δεν ήτανε μόνο κατά 1 subregion.

Από εκεί και πέρα στα (1) και (2) γίνεται logging και συλλογή στατιστικών ομοίως με τα splits των point nodes. Επίσης όπως στα point nodes στο (3) γίνεται split του region που αντιπροσωπεύει ο κόμβος,. Στο (4) επιλέγονται όσα subregions τέμνονται από το υπερεπίπεδο διάσπασης (και όχι στα boundaries τους) και στο (5) γίνεται σε αυτά forced splits.

Τα forced splits του (5) γεμίζουν την λίστα subregions του κόμβου με το πολύ δύο λιγότερο (4.2.1.6) από τα διπλάσια στοιχεία από όσα αρχικά είχε (το split γίνεται σε ένα boundary μεταξύ δυο subregions και το πολύ όλα τα υπόλοιπα να διασπαστούν, 4.2.1.6) και όλα αυτά τα νέα subregions θα χωριστούν σε δεξιά και αριστερά.

Επειδή είναι δύο λιγότερα από το διπλάσιο του ορίου υπερχείλισης η διάσπαση δεν θα δώσει overflown κόμβους (4.2.1.6). Στα (6) γίνεται με LINQ query η δημιουργία της δεξιάς και αριστερής λίστας subregion καλώντας τις κατάλληλες μεθόδους της Region για τον καθορισμό ποιο region είναι δεξιά ή αριστερά (σε αυτό το σημείο κανένα από τα subregions δεν τέμνεται στο εσωτερικό του από το επίπεδο διαχωρισμού γιατί έχουνε ολοκληρωθεί τα forced splits) Στα (7) δημιουργούνται οι νέοι κόμβοι με βάση τις λίστες του (6) και στο (8) ρυθμίζεται ο γονέας.

```
public override void split(int d, float v, RegionNode parent)
    {
      Debug.Assert(parent.getRegion(this).DividedBy(new Cut(d,v)),"The cutting line does not lie
        in the interior of   the region");
      Debug.Assert(parent != null, "The parent is null");
      Debug.Assert(parentStack.Peek()==parent, "Stack handled correctly");
      Debug.Assert(parent.subr.Count < SIZE||parent.splitting, "Non splitting Parent hat at lest one
        less region than the max no of regions");
      if (!(parentStack.Count == 2 && parent.subr.Count == 1))
      { Debug.Assert(parent.subr.Select(rrr => rrr.n).Contains(this), "The parent does not contain the
        node"); }
      var oldcount = parent.subr.Count;
      splitting = true;

      //τέλος ελέγχων εκκίνησης

      Program.log += "Split at level " + parentStack.Count + "\n";                    (1)
      par.pageswritten += 1;                                                          (2)

      Region left, right;
      parent.getRegion(this).split(d, v, out left, out right);                        (3)
      List<Region> leftl = new List<Region>();
      List<Region> rightl = new List<Region>();

      var rrtmp = (from r in subr where r.DividedBy(new Cut(d, v)) select r).ToList();  (4)
      foreach (var rr in rrtmp)                                                       (5)
      {             Down(this, () => rr.n.split(d, v, this));          }


      var lrs = (from reg in subr where reg.Contains(d, v) == -1 select reg).ToList();  (6)
      leftl.AddRange(lrs);                                                             (6)


      var rrs = (from reg in subr where reg.Contains(d, v) !=-1 select reg).ToList();   (6)
      rightl.AddRange(rrs);                                                            (6)
```

```
right.n = new RegionNode(rightl, splitd);                                    (7)
left.n = new RegionNode(leftl,  splitd);                                     (7)
Up(() => parent.Substitute(this,left,right,d));                              (8)

// έλεγχοι μετά την λήξη της συνάρτησης
RegionNode assertr = right.n as RegionNode;
RegionNode assertl = left.n as RegionNode;

Debug.Assert(!parent.subr.Select(rrrr=>rrrr.n).Contains(this),"The parent still holds a
  reference to this node");
Debug.Assert(parent.subr.Count == (oldcount + 1), "Size of parent's subr has not increased by
  exactly one");
Console.WriteLine(Program.i + " " + parentStack.Count + " " + parent.subr.Count);
Debug.Assert(parent.subr.Count<=SIZE+1||parent.splitting,"Non splitting Parent has more
  than one extra region than max “);
Debug.Assert(assertl!=null, "Failed to create valid nodes");
Debug.Assert(assertr != null, "Failed to create valid nodes");

Debug.Assert(assertl.subr.Count<=SIZE, "We created nodes that have more regions than
  allowed");
Debug.Assert(assertr.subr.Count <= SIZE, "We created nodes that have more regions than
  allowed");
}
```

Όπως αναφέρθηκε η διαδικασία αφαίρεσης παλαιού κόμβου και προσθήκης των 2 νέων που προέκυψαν από την διάσπαση είναι transaction like και γίνεται από την συνάρτηση Substitute που επιπλέον εκτελεί και άλλους ελέγχους πριν καλέσει της Insert(Region) και Remove(Region), ελέγχους που δεν θα μπορούσαν να τοποθετηθούν μόνο σε κάποια εκ των δύο γιατί αφορούν όλο το transaction.

```
public void Substitute(Node rem, Region ileft, Region iright, int dimension)
{
   Debug.Assert(rem != null, "Tried to remove a null");
   Debug.Assert(ileft != null, "Tried to insert a null left region");
   Debug.Assert(ileft != null, "Tried to insert a null right region");
   Region tmp2;
   if (!(parentStack.Count == 1 && subr.Count == 1))
   {
      var tmp1 = subr.Where(x => x.n == rem);
      Debug.Assert(tmp1.Count() == 1, "There is not a single node to be deleted");
      tmp2 = tmp1.ElementAt(0);
   }
   else
   {
      Debug.Assert(subr[0].Same(new Region()),"The new root node should have the default
     region");
      tmp2=subr[0];
   }

   var oldr = tmp2;
   var oldcount=subr.Count;
   Debug.Assert(tmp2.Same(Region.Combine(ileft,iright,dimension)), "Deleted region not union
     of inserted          regions");

  // τέλος ελέγχων έναρξης
   Remove(rem);
   Insert(ileft);
   Insert(iright);


  //έλεγχοι τερματισμού

   Debug.Assert(subr.Count == (oldcount + 1), "Size of subr didn't increase by one");
   Debug.Assert(!subr.Contains(oldr), "The deleted region hasn't been deleted");
   Debug.Assert(subr.Contains(ileft), "The left region was not inserted");
   Debug.Assert(subr.Contains(iright), "The right region was not inserted");
}
```

Μπορεί το πλήθος των ελέγχων να είναι μεγάλο σε σχέση με το σώμα της εφαρμογής όμως αυτό δεν αποτελεί πρόβλημα καθώς αυτοί μπορούν να απενεργοποιηθούν από τον compiler και να μην επηρεάσουν την ταχύτητα εκτέλεσης, ενώ από την άλλη η προσθήκη του επιπλέον ελέγχων δεν είναι δύσκολη προγραμματιστικά διαδικασία που θα επιβραδύνει την ανάπτυξη.

Στις διασπάσεις τον RegionNodes η ChooseDimension() και η ChooseValue() χρησιμοποιούν μια βοηθητική συνάρτηση την ChooseCut. Αυτή αναλαμβάνει να βαθμολογήσει τα boundaries των subregions με κριτήρια καταλληλότητας το πόσο καλή είναι η διάσπαση σε αυτά από άποψης τελικού utilization, το πλήθος διασπώμενων κόμβων, και της κυβοποίησης των regions.

```
private Cut ChooseCut()
    {
      int a=splitd;
      var cc = new List<Cut>();
      subr.Each(sr => cc.AddRange(sr.getCuts()));                    (1)
      var cc2 = cc.Distinct();                                       (2)
      var toremove = parentStack.Peek().getRegion(this).getCuts();   (3)
      cc2 = cc2.Except(toremove);                                    (3)
      var res = cc2.Select(cut => new {
        TCUT = cut,
        TTIMES = Splits(cut),
        TCUBE=Cubicality(cut),
        TUTIL=EqualUtilization(cut)}
        );                                                           (4)
      int ran = new Random().Next(1000);
      if (ran<200)                                                   (5)
      res.OrderBy(k => k.TTIMES);
      else if ((ran >= 200)&&(ran < 600))
      res.OrderBy(k => k.TUTIL);
      else res.OrderBy(k => k.TCUBE);
      return res.ElementAt(0).TCUT;                                  (6)
  }
```

Στο (1) δημιουργείται μια λίστα από υπερεπίπεδα πάνω στα boundaries από κάθε subregion. Τα υπερεπίπεδα αναπαριστώνται από την κλάση Cut. Η Cut έχει δύο πεδία ένα με την διάσταση και ένα άλλο με το offset του υπερεπιπέδουσε αυτήν.

Στο (2) αφαιρούνται από την λίστα οι διπλές καταχωρήσεις Στο (3) αφαιρούνται τα υπερεπίπεδα που ανήκουν και στο boundary του διασπώμενου region R (και που ήταν στην λίστα γιατί τα ακριανά subregions έχουνε κοινά boundaries με το υπέρ-region), καθώς πρέπει το split να γίνει στο εσωτερικό του R.

Στο (4) δημιουργείται με LINQ μια συλλογή από αντικείμενα ενός ανώνυμου τύπου ο οποίος κρατάει για κάθε Cut το πλήθος των split που μπορεί να προκαλέσει, και το utilization και cubicality που θα επιφέρει. Οι υπολογισμοί γίνονται με 3 βοηθητικές συναρτήσεις. Στο (5) η συλλογή ταξινομείται επιλέγοντας τυχαία κάποιο από τα κριτήρια. Ανάλογα πόση πιθανότητα δίνεται σε κάθε κριτήριο τόσο πιο πολύ θα επηρεάζει την τελική δομή. Στο (6) επιστρέφεται το πρώτο Cut της ταξινόμησης.

Αυτό το σημείο επιβραδύνει αρκετά το τελικό πρόγραμμα και αν είναι ανάγκη μπορούν να αφαιρεθούν κριτήρια ώστε να αυξηθεί η ταχύτητα.

Εδώ παρουσιάζονται οι βοηθητικές της ChooseCut() συναρτήσεις.

Η Splits() απλώς βρίσκει πόσα subregions τέμνει το Cut. Η Cubicality() λειτουργεί ως εξής. Πρώτα βρίσκει το μήκος ακμών για κάθε διάσταση, και από αυτά τα μήκη το ελάχιστο. Στην συνέχεια διαιρεί κάθε μήκος με το ελάχιστο και κοιτάζει κατά πόσο το τελικό αποτέλεσμα τείνει στο 1 για κάθε διάσταση. Συγκεκριμένα αφαιρεί μια μονάδα από τον μετασχηματισμένο πίνακα μηκών και παίρνει ένα μ.ο των τιμών που απομένουν. Όσο πιο μικρός είναι αυτός τόσο πιο cubical θεωρείται το region. Η μέτρηση είναι πρόχειρη και σε πολλές διαστάσεις θα δώσει λάθος αποτελέσματα αλλά σε λίγες διαστάσεις εκτιμήθηκε ότι δεν θα κάνει μεγάλο λάθος. Ομοίως

λειτουργεί και η EqualUtilization() που όμως δεν έχει πρόβλημα με τις πολλές διαστάσεις γιατί δεν κάνει χρήση μέσων όρων. Συγκεκριμένα αυτή διαιρεί το πλήθος subregions του κόμβου με τα περισσότερα subregions με το πλήθος αυτού με τα λιγότερα, και ελέγχει αν αυτό το πηλίκο τείνει προς το 1.

6.4.4 XML αναπαράσταση

Το δένδρο της δομής αποθηκεύεται σε xml μορφή με χρήση της συνάρτησης ToXml(). Αυτή όταν εκτελείται σε κάποιο κόμβο επιστρέφει ένα string που περιγράφει σε κείμενο xml το υποδένδρο του κόμβου.

Έτσι με βάση την αναδρομή καλώντας την στο Node.root επιστρέφεται ολόκληρη η δομή σε Xml. Εσωτερικά σε κάθε κόμβο, πριν καλέσει αναδρομικά την ToXml() των υιών, η συνάρτηση κατασκευάζει ένα string με βάση τις ιδιότητες του κόμβου.

Στο (1) δημιουργείται ένα string στο οποίο βήμα βήμα προστίθενται tags μέχρι να προκύψει η όλη xml αναπαράσταση. Στο (2) ξεκινάει ένα tag Region καθώς αυτή η ToXml αναπαριστά RegionNodes (στην ToXml() των points το πρώτο tag θα ήτανε <Point>). Τοποθετείται στο tag ένα attribute με το ύψος του στο δένδρο καθώς το αρχείο αυτό χρησιμοποιείται και για την γραφική αναπαράσταση της δομής η οποία θα χρειαστεί να επιλέγει κάθε φορά regions μόνο συγκεκριμένου ύψους.

Από εκεί και πέρα προστίθενται tags με βάση τις ιδιότητες του region node ενώ στο (3) καλείται αναδρομικά η ToXml όλων των υιών έναν προς ένα και τα αποτελέσματα προστίθενται στο a πριν κλείσει και επιστραφεί αυτό με το closing tag για το αρχικό tag <Region>

```
public override string ToXml()
    {
      String a = "";                                                    (1)
      a += "<Region Level=\"" + parentStack.Count + "\" >";             (2)
      a += "<MINX>" + Math.Round(Combine().min[0],0) + "</MINX>";
      a += "<MAXX>" + Math.Round(Combine().max[0],0) + "</MAXX>";
      a += "<MINY>" + Math.Round(Combine().min[1],0) + "</MINY>";
      a += "<MAXY>" + Math.Round(Combine().max[1],0) + "</MAXY>";
      a += "<HEIGHT>" + Math.Round((Combine().max[1] - Combine().min[1]),0) +
        "</HEIGHT>";
      a += "<WIDTH>" + Math.Round((Combine().max[0] - Combine().min[0]),0) + "</WIDTH>";
      }
      foreach (var r in subr) { Down(this,()=>a += r.n.ToXml()); }(3)
      a += "</Region>";
      return a;
    }
```

Παράδειγμα XML αρχείου που θα κατασκευαστεί με αυτόν τον τρόπο είναι το ακόλουθο

```
<?xml version='1.0'?>
<Data>
<Region Level="1">
<MINX>0</MINX>
<MAXX>500</MAXX>
<MINY>0</MINY>
<MAXY>500</MAXY>
<HEIGHT>500</HEIGHT>
<WIDTH>500</WIDTH>
<Region Level="2" >
<MINX>0</MINX>
<MAXX>293</MAXX>
<MINY>0</MINY>
<MAXY>500</MAXY>
```

<HEIGHT>500</HEIGHT>
<WIDTH>293</WIDTH>
<Region Level="3" >
<MINX>0</MINX>
<MAXX>293</MAXX>
<MINY>0</MINY>
<MAXY>370</MAXY>
<HEIGHT>370</HEIGHT>
<WIDTH>293</WIDTH>
<Points>
<Point>
<X>14,58917</X><Y>122,0153</Y>
</Point>
<Point>
<X>157,8356</X><Y>256,0248</Y>
</Point>
[...]
</Region>
<Region Level="3" >
<MINX>0</MINX>
[...]
</Region>
[…]
[…]
</Region>
</Region>
</Data>

6.4.4.1 Μετασχηματισμός με XSLT

Το xml αρχείο αυτό μπορεί να μετασχηματιστεί ώστε να δώσει το αρχείο xaml με το οποίο το WPF runtime θα ζωγραφίσει μια αναπαράσταση της δομής. Ο μετασχηματισμός θα μπορούσε θεωρητικά να γίνει και με χειρισμό του xml αρχείου σαν string χαρακτήρων όμως αυτό θα ήτανε ιδιαίτερα χρονοβόρο. Ο ενδεδειγμένος τρόπος μετασχηματισμού είναι μέσω της γλώσσας XSLT που είναι γλώσσα ειδική για μετασχηματισμούς xml αρχείων.

Η xslt είναι κάπως δύσχρηστη στον υπολογισμό νέων τιμών και δεν έχει καλό debugging. Για αυτόν τον λόγο η χρήση της θα πρέπει να γίνεται με τροφοδοσία όλων των δεδομένων προϋπολογισμένων στην C# και αποθηκευμένων στο xml αρχείο ώστε η XSLT απλώς να αναλαμβάνει να κάνει με template matching την αναδρομική επιλογή των κόμβων πάνω στο δένδρο και να εκτυπώσει τα δεδομένα που βρίσκει εκεί προετοιμασμένα.

Το παρακάτω είναι το αρχείο transform.xslt που μετασχηματίζει το xml representation της δομής σε xaml αρχείο με αντικείμενα WPF.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:s="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<xsl:output method="xml" />

<xsl:param name="lv"></xsl:param>                                    (1)

<xsl:template match="/">                                             (2)
<s:Canvas xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">    (3)
<xsl:apply-templates select="//Region[@Level=$lv]" />                (4)
<xsl:apply-templates select="//Point" />                             (5)
</s:Canvas>
 </xsl:template>
```

```
<xsl:template match="Region">                                                    (6)
<s:Rectangle s:Canvas.Left="{MINX}" s:Canvas.Top="{MINY}" Height="{HEIGHT}"
Width="{WIDTH}" Fill="sc#0,0.5,0,0.8" Stroke="sc#0.9,0,0,0.8" StrokeThickness="1" />
 </xsl:template>

<xsl:template match="Point">                                                     (7)
<s:Rectangle s:Canvas.Left="{X}" s:Canvas.Top="{Y}" Height="3" Width="3" Fill="Black"
Stroke="Black" StrokeThickness="1" />
 </xsl:template>
  </xsl:stylesheet>
```

Στο (1) ορίζονται παράμετροι του προγράμματος XSLT (κάτι που είναι ανάλογο με τον πίνακα argv μιας main συνάρτησης) Η συγκεκριμένη παράμετρος επιλέγει επίπεδο του δένδρου για το οποίο τα regions θα μετασχηματιστούν σε xaml ώστε να σχεδιαστούν. Ανάλογα με αυτήν την παράμετρο η γραφική απεικόνιση θα περιέχει μια λιγότερο η περισσότερο λεπτομερή διαμέριση του χώρου από το kdb tree.

Στο (2) βρίσκεται το root pattern που αποτελεί το σημείο εισόδου ενός προγράμματος XSLT. Εδώ τυπώνεται στην έξοδο η ρίζα του xaml δένδρου που είναι ένα στοιχείο Canvas, ένα WPF container (3).

Στην συνέχεια στα (4) (5) ζητείται η εφαρμογή templates πρώτα μόνον σε όλα τα xml elements τύπου Region (ώστε να σχεδιαστούν τα regions) και στην συνέχεια στα elements <Point> (για τα points). Στο (4) μέσω Xpath ζητείται συγκεκριμένα μόνο τα regions με attribute ύψους ίσο με αυτό που δόθηκε ως παράμετρος να επιλεγούν για εφαρμογή templates. Στο (6) βρίσκεται το template που ταιριάζει με όλα τα element region οπουδήποτε στο δέντρο, αλλά εδώ λόγω του (4) θα εφαρμοστεί σε regions μόνο συγκεκριμένου ύψους.

Αυτό εκτυπώνει ένα στοιχείο Rectangle της xaml με τις κατάλληλες προϋπολογισμένες και αποθηκευμένες στο xml αρχείο τιμές ύψους και θέσης. Η θέση ορίζεται σε σχέση με το εξωτερικό Canvas (που ως ρίζα περικλείει όλα τα στοιχεία του xaml αρχείου) και γίνεται χρήση του συντακτικού της xaml

(object.property) για τον καθορισμό της θέσης αυτής. Στο (7) εκτυπώνεται στην έξοδο για κάθε point και ένα Rectangle ύψους και πλάτους ενός pixel

Το αποτέλεσμα της εφαρμογής του μετασχηματισμού XSLT έχει μορφή παρόμοια με την επόμενη κα μπορεί να χρησιμοποιηθεί για τον σχεδιασμό της δομής στην οθόνη από το WPF.

<s:Canvas xmlns:s="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"><s:Rectangle Height="500" Width="500" Fill="sc#0,0.5,0,0.8" Stroke="Blue" /></s:Canvas>

Ο μετασχηματισμός εφαρμόζεται σε C# μέσω κώδικα όπως ο παρακάτω

```
string b2 = "<?xml version='1.0'?><Data>" + Node.root.ToXml() + "</Data>";          (1)
XPathDocument myXPathDoc = new XPathDocument(new StringReader(b2));                  (2)
XslTransform myXslTrans = new XslTransform();
myXslTrans.Load("transform.xsl");                                                    (3)

for(int i=0;i<CountLevels();i++)                                                     (6)
{
XsltArgumentList xslArg = new XsltArgumentList();
xslArg.AddParam("lv", "", ""+(i+1));                                                (4)
var k=new StringWriter(new StringBuilder());
myXslTrans.Transform(myXPathDoc,xslArg,k);                                           (5)
result.Add(k.ToString());
}
```

Στο (1) κατασκευάζεται ένα string με την xml αναπαράσταση του kdb tree. Στο (2) το string τοποθετείται σε ένα XPathDocument, που είναι μια αφαίρεση οποιοδήποτε κειμένου xml που πρόκειται να επεξεργαστεί μέσω XSLT/XPATH. Στο (3) δημιουργείται ένα αντικείμενο για τον μετασχηματισμό με βάση το αρχείου .xsl του. Στο (4) ορίζονται τα ορίσματα του μετασχηματισμού στο (5) αυτός εκτελείται με βάση τα ορίσματα και την xml αναπαράσταση επιστρέφοντας ένα stream το οποίο εν

τέλει μετατρέπεται σε string.

Ο βρόγχος στο (6) επαναλαμβάνει την διαδικασία για κάθε επίπεδο και τοποθετεί τα αποτελέσματα σε μια λίστα με string.

6.5 Εξωτερική χρήση και αρχικοποίηση

Η χρήση της δομής από το GUI γίνεται ως εξής.. Κάποιο event στο GUI εκτελεί την συνάρτηση

public static List<String> InsertService(int xx, int yy) της κλάσης KDB_Tree της κλάσης KDB_Tree. Αυτή τοποθετεί στο δένδρο το σημείο που επιλέχτηκε με το event και στην συνέχεια με μετασχηματισμό του δένδρου με XSLT επιστρέφει μια λίστα με xaml γραφικές απεικονίσεις στο GUI. Τότε το GUI μπορεί να σχεδιάσει όσες από αυτές θέλει.

Η κλάση KDB_Tree περιέχει μόνο static members και λειτουργεί ως σημείο εισόδου για όσες εφαρμογές θέλουν να χρησιμοποιήσουν το kdb tree. Η άλλη συνάρτηση της είναι η Init() που κάνει reset το δένδρο

```
public static void Init(int dimensions, int size, int sizep, float MIN, float MAX)
    {
      Node.SIZE = size;
      Node.SIZEP = sizep;
      Region.DC = dimensions;
      Region.MAX = MAX;
      Region.MIN = MIN;
      Node.par = new PagesAccessedResults();                    (1)
      Node.par.pagesread = 0;
      Node.par.pageswritten = 0;
      Node.parentStack = new Stack<RegionNode>();               (2)
      Node.parentStack.Push(null);
    }
```

Η Init() δέχεται ως παραμέτρους τα χαρακτηριστικά του δένδρου (πλήθος διαστάσεων κατώφλι υπερχείλισης) τα οποία θέτει στις static και global μεταβλητές που χρησιμοποιεί το υπόλοιπο πρόγραμμα όταν θέλει να τα μάθει. Στο (1) αρχικοποιεί την δομή καταγραφής στατιστικών και στο (2) την στοίβα, θέτοντας και ως πρώτο στοιχείο της το null, καθώς ο τρόπος που αντιλαμβάνονται οι κόμβοι ότι είναι οι ίδιοι η ρίζα είναι ελέγχοντας αν η κορυφή της στοίβας δείχνει στο null.

6.6 Έλεγχος

Η CheckLeaves κάνει έναν έλεγχο στην δομή με βάση ένα μεγάλο πλήθος στοιχείων προσπαθώντας με monte carlo τρόπο να εντοπίσει κενά στην διαμέριση του χώρου από το kdb tree.

Η CheckLeaves() λειτουργεί με ένα όρισμα το οποίο αποτελεί λίστα αποτελεσμάτων. Διενεργεί Depth First Search σε όλο το δένδρο προσθέτοντας σε αυτό το όρισμα μόνο τα φύλλα (τα point nodes) Στην συνέχεια ο ακόλουθος κώδικας μπορεί να διενεργήσει τον έλεγχο

```
var li=new List<Region>();                                    (1)
var x = Node.root as RegionNode;
x.CheckLeavesHelper(li);                                      (2)
var l2 = li.Distinct();
Random random = new Random();
for (int i = 0; i < 1000000; i++)                             (3)
{

   Point pp = new Point();
  for (int j = 0; j < Region.DC; j++)
     {pp.x[j] = ((float)random.NextDouble()) * (Region.MAX – Region.MIN)+Region.MIN;}
   if (l.Where(sss => sss.ContainsPoint(pp)).Count() != 1) throw new Exception();      (4)
}
```

Στο (1) δημιουργείται η λίστα των αποτελεσμάτων, στο (2) η CheckLeaves() τοποθετεί σε αυτήν όλα τα point nodes στο (3) για ένα εκατομμύριο τυχαία σημεία ελέγχεται μέσω του (4) αν η λίστα όλων τον point nodes διαθέτει λιγότερες της μίας ή περισσότερες της μίας περιοχές που να περιέχουν το σημείο, κάτι που θα καταδεικνύει ότι έχει γίνει λάθος διαμέριση.

7. ΠΑΡΟΥΣΙΑΣΗ ΠΡΟΓΡΑΜΜΑΤΟΣ ΓΡΑΦΙΚΗΣ ΑΝΑΠΑΡΑΣΤΑΣΗΣ

Στα πλαίσια της εργασίας κατασκευάστηκε πρόγραμμα το οποίο κάνει γραφική απεικόνιση της δομής ενός δισδιάστατου kdb tree καθώς εισάγονται σε αυτό σημεία.

Το πρόγραμμα μπορεί κανείς να το εκτελέσει κάνοντας unzip τον φάκελο Visualization.rar και τρέχοντας το αρχείο WpfApplication1.exe.Η ανάλυση της οθόνης πρέπει να είναι τουλάχιστον 1024*768 και στον υπολογιστή θα πρέπει να είναι εγκατεστημένο το .NET 3.5 runtime. (Σημείωση με τον αλγόριθμο του split που χρησιμοποιείται η απόκριση της εφαρμογής μπορεί να γίνει κάπως χαμηλή μετά την εισαγωγή περί των 100 σημείων. Με έναν πιο απλό αρχικό αλγόριθμο split το πρόβλημα δεν υπήρχε. Αν και ο νέος αλγόριθμος δεν έχει κάποιο πλεονέκτημα στο utilization διατηρήθηκε γιατί μπορεί να δώσει πιο cubical region. Πάντως στην εφαρμογή υπάρχει μεγάλο περιθώριο βελτιστοποίησης καθώς η ανάπτυξη έγινε με σκοπό την γρήγορη υλοποίηση και όχι τις επιδόσεις)

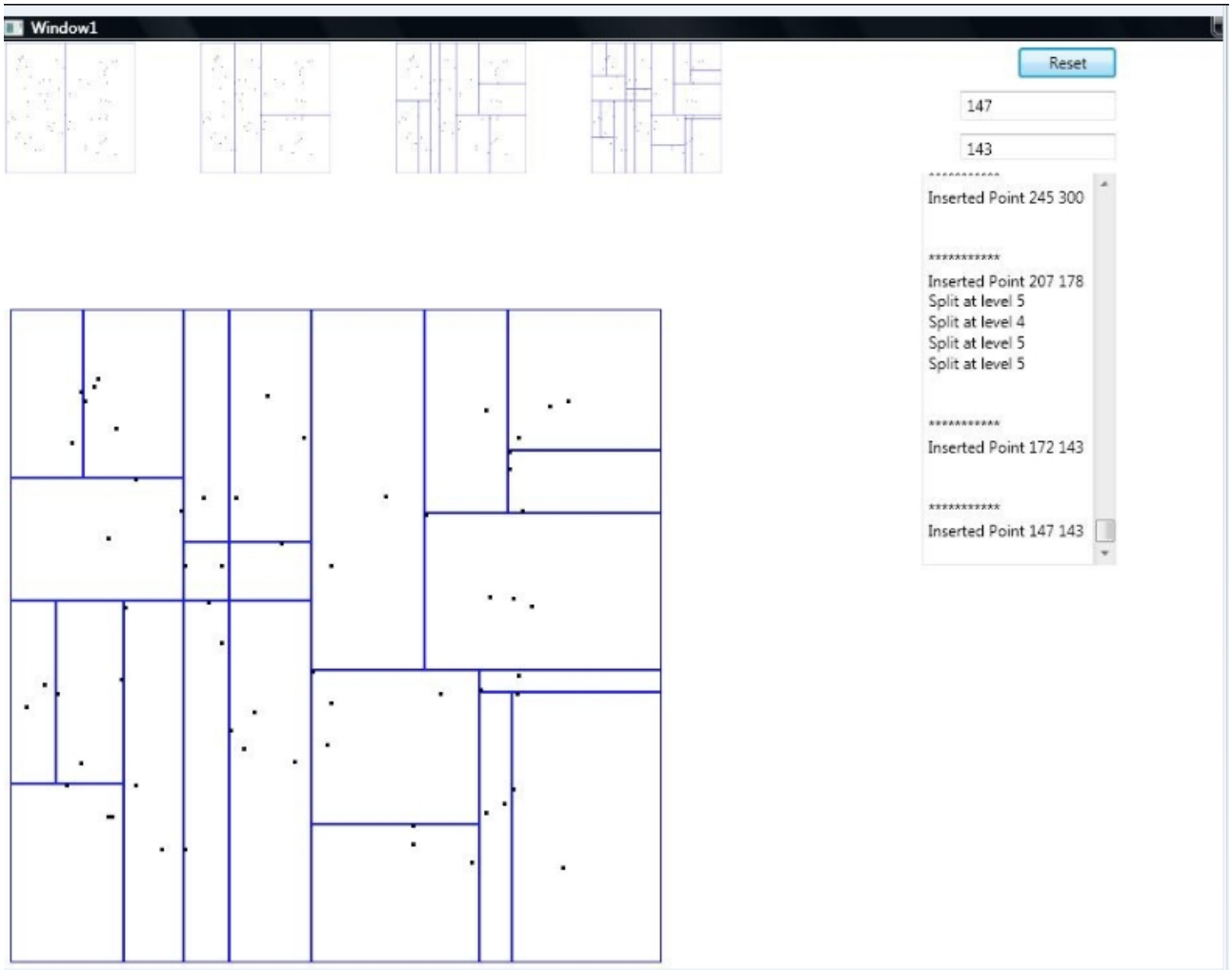
Το GUI περιλαμβάνει κάτω αριστερά ένα πεδίο εισαγωγής σημείων όπου ο χρήστης απλώς κάνει αριστερό κλικ στο σημείο που θέλει να γίνει η εισαγωγή. Μόνο σημεία με ακέραιους αιθμούς συντεταγμένων εισάγονται οπότε το πρόγραμμα αποτελεί μια πρόχειρη προσομοίωση και δεν λειτουργεί όπως ένα κανονικό kdb tree.Το πεδίο αυτό δείχνει μια απεικόνιση της διαμέρισης του χώρου στο κατώτερο επίπεδο του kdb tree (στα pointnodes)

Ταυτόχρονα στο πάνω μέρος της οθόνης υπάρχουνε real time previews που δείχνουν πως γίνεται η διαμέριση στα 4 ανώτερα επίπεδα του kdb tree. Για λόγους κατανόησης επιλέχθηκε πλήθος σημείων ανά point page και regions ανά region page τα 4 σημεία ή regions οπότε κατά την εισαγωγή ενός 5ου σημείου στο κάτω πεδίο θα γίνει μια διάσπαση σε real time, ενώ αναλόγως μετά από κάποιο αριθμό εισαγωγών γίνονται διασπάσεις και σε ανώτερα επίπεδα.

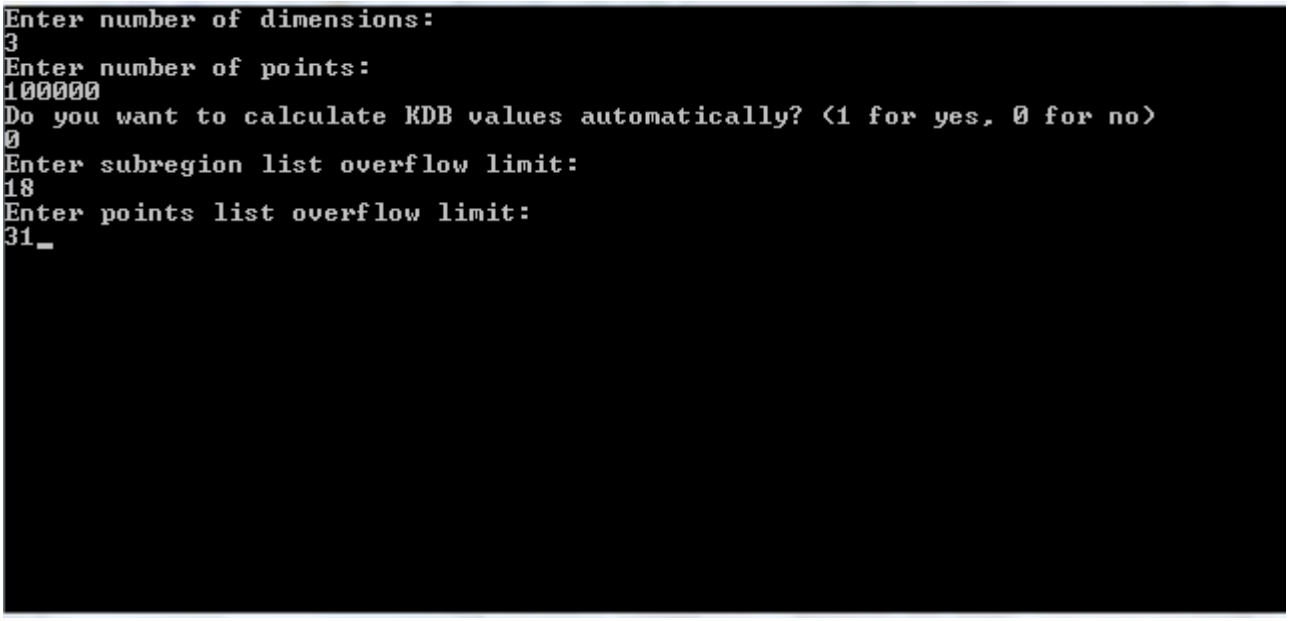
Δεξιά υπάρχει scroll box όπου δίνονται πληροφορίες για το πότε γίνεται το κάθε split, ενώ η διαδικασία επανεκκινείται με το πάτημα του κουμπιού reset. Σημείωση ότι μετά την εκκίνηση της εφαρμογής πρέπει αρχικά να πατηθεί το reset μια φορα πριν γίνει δυνατή η εισαγωγή σημείων.

Η δεύτερη εφαρμογή εκτελεί tests πάνω στην δομή. Η εφαρμογή εκτελείται κάνοντας unzip το Test.rar και τρέχοντας το αρχείο testing.exe. Τότε στην κονσόλα ο χρήστης καλείται να ορίσει το test. Η ερώτηση για το automatic calculation αφορά το αν θα γίνει αυτόματα η επιλογή του μεγέθους των subregionlists και pointlists με βάση το μέγεθος block των 4096 bytes. Μετά την ολοκλήρωση της εκτέλεσης τα αποτελέσματα βρίσκονται στον ίδιο φάκελο και στο αρχείο testresults.xml. Να σημειωθεί ότι για points περισσότερα των 10000-20000 η εφαρμογή γίνεται απότομα και εκθετικά πιο αργή κάτι που μάλλον δεν έχει σχέση μόνο με τον αλγόριθμο αλλά και με κάποιο πρόβλημα του .NET να χειριστεί μεγάλους πίνακες.

Επίσης παραδίδονται το αρχείο src.rar με τον πηγαίο κώικα συγκεντρωμένο, το αρχείο Library.rar που είναι το library της εφαρμογής μαζί με μια συνάρτηση Main που εκτελεί καποιους επιπλέον ελέγχους καθώς και οι φάκελοι των Visual Studio 2008 projects ConsoleApplication1 (το library), Testing (η εφαρμογή εκτέλεσης ελέγχων) και WpfApplication1 (το γραφικό περιβάλλον).



Screenshot από το πρόγραμμα γραφικής απεικόνισης KDB Tree 2 διαστάσεων



Screenshot από το πρόγραμμα κονσόλας για την εκτέλεση δοκιμών

8. ΔΟΚΙΜΕΣ

Οι μετρήσεις αποδοτικότητας που έγιναν πάνω στην δομή ήτανε αυτές που είχαν γίνει και στο αρχικό paper [Robinson 81].

Συγκεκριμένα μετρήθηκε το utilization ως μέσος όρος του ποσοστού των subregionlists και pointlists που περιείχε καταχωρήσεις, δηλαδή κατά κάποιο τρόπο το effective fan out της δομής..

Επίσης μετρήθηκαν το πλήθος pages σε κάθε επίπεδο του δένδρου καθώς και ο μέσος όρος αναγνώσεων και εγγραφών σελίδων κατά την εισαγωγή κάποιου σημείου. Η εισαγωγή μπορεί να προκαλέσει περισσότερες της μίας (στο τελικό point list) εγγραφές σελίδων εφόσον είναι δυνατόν να επιφέρει και διασπάσεις σελίδων .

Το πλήθος pages ανά επίπεδο λόγω των forced splits ίσως δεν αποτελεί καλή μετρική, με την έννοια ότι δεν είναι "ευσταθής". Αν σε δύο δομές kdb tree γίνει εισαγωγή 10000 σημείων τότε υπάρχει η περίπτωση η πρώτη να κάνει split στην ρίζα στο 9998 ενώ η δεύτερη να μπορεί να δεχτεί μερικές δεκάδες σημεία ακόμα πριν γίνει το split. Έτσι παρόλο που και οι δύο μπορεί να έχουνε την ίδια συμπεριφορά η μία στο τέλος του test θα φανεί να έχει διπλάσιο αριθμό από pages καθώς ένα split στην ρίζα μπορεί να σημαίνει και forced splits σε ολόκληρη την δομή, ειδικά σε πολλές διαστάσεις.

Όσον αφορά τα queries μετρήθηκε ένα στατιστικό που ο Robinson ονομάζει Query Efficiency και το οποίο ορίζεται για κάποιο query ως το πλήθος των εγγραφών που βρέθηκαν ως προς το πλήθος των συνολικών εγγραφών, επί το πλήθος των συνολικών σελίδων προς το σύνολο των σελίδων που αναγνώστηκαν κατά το query. Αυτό προσπαθεί να μετρήσει το κατά πόσο ο διαχωρισμός των εγγραφών σε pages συμφωνεί με την ομαδοποίηση που υποδηλώνει το καθε range query. Δηλαδή το μέτρο θα τείνει στο 1 όταν το ποσοστό σελίδων που χρησιμοποιεί το range query είναι όμοιο με το ποσοστό εγγραφών που περιλαμβάνει το κριτήριο αναζήτησης. Αν

περισσότερες σελίδες χρησιμοποιούνται από το αναμενόμενο με βάση το πλήθος εγγραφών που ταιριάζουν στα κριτήρια ποσοστό το στατιστικό τείνει στο 0

Όπως και στις δοκιμές του Robinson αρχικά έγινε εισαγωγή 10000 σημείων σε kdb trees με συγκεκριμένα χαρακτηριστικά. Τα σημεία είχαν τυχαία ομοιόμορφη κατανομή στον χώρο του καρτεσιανού γινομένου του διαστήματος[0-50000] για κάθε διάσταση.

Τα queries των δοκιμών προέκυψαν από τα queries του paper. Συγκεκριμένα ο Robinson χρησιμοποιεί κάποια ad hoc queries για να μετρήσει το στατιστικό query efficiency.

Στα δοκιμαστικά queries αυτής της εφαρμογής δεν έγινε χρήση αυτών ακριβώς των regions που πρότεινε ο Robinson.

Αντίθετα, με βάση τα region queries του paper, δημιουργήθηκαν δύο regions περίπου στο κέντρο του χώρου με πλευρές 0.2 το πρώτο και 0.3 του συνολικού χώρου το δεύτερο.

Δηλαδή ο αλγόριθμος κατασκευής υπολόγιζε το μήκος των ακμών του region query με βάση το μήκος του χώρου και στην συνέχεια τοποθετούσε το region σε σημείο τέτοιο ώστε να μην ξεπερνάει τα όρια του χώρου

Πχ region.max[i]=0.2*(Region.MAX- Region.MIN)+a[i]
όπου στον ψευδοκώδικα το a είναι τυχαίο σημείο του χώρου τέτοιο ώστε
region.max[i]<Region.MAX.

Για το κάθε ένα από αυτά τα δύο regions και για κάθε διάσταση έγινε επίσης και ένα partial query όπου η διάσταση αυτή μπορούσε να πάρει την οποιαδήποτε τιμή.

Το average query efficiency ήταν τελικά ο μέσος όρος από όλες αυτές μετρήσεις query efficiency για τα πρότυπα regions και για τα partial queries.

* * *

Αρχικά έγιναν μετρήσεις για τις "τυπικές" περιπτώσεις με βάση τις δοκιμές του Robinson.

Αποτελέσματα για 2 διαστάσεις 25 regions 42 points

<Tests regionsize="**25**" pointsize="**42**" dimensions="**2**">
<Level height="**0**">0</Level>
<Level height="**1**">1</Level>
<Level height="**2**">2</Level>
<Level height="**3**">27</Level>
<Level height="**4**">384</Level>
<Utilization>0,616957</Utilization>
<AveragePagesAccessed>
<Read>3,015</Read>
<Written>1,0847</Written>
</AveragePagesAccessed>
<AverageQueryEfficiency>0,3004983</AverageQueryEfficiency>
</Tests>

Οι τιμές του Robinson ήταν

Levels 1, 19, 360

Utilization 0.66

Pages Read 2.93

Pages Written 1.13

3 διαστάσεις 18 regions 31 points

```
<?xml version="1.0" ?>
<Tests regionsize="18" pointsize="31" dimensions="3">
 <Level height="0">0</Level>
 <Level height="1">1</Level>
 <Level height="2">7</Level>
 <Level height="3">68</Level>
 <Level height="4">634</Level>
 <Utilization>0,5094187</Utilization>
 <AveragePagesAccessed>
 <Read>3,6594</Read>
 <Written>1,1471</Written>
 </AveragePagesAccessed>
 <AverageQueryEfficiency>0,3535215</AverageQueryEfficiency>
 </Tests>
```

Οι τιμές του Robinson ήταν

Levels 1, 4, 45, 567

Utilization 0.54

Pages Read 3.53

Pages Written 1.16

Κατ αρχήν το kdb tree της εφαρμογής φαίνεται να κάνει πιο γρήγορα το split στην ρίζα από το αντίστοιχο του Robinson. Το πιο γρήγορο split στην ρίζα αυξάνει κατά πολύ τον αριθμό των pages (για ένα προσωρινό διάστημα). Ο αριθμός των pages δηλαδή κάνει απότομα άλματα κατά την εισαγωγή σημείων και οι συγκρίσεις μπορεί να μην είναι εύκολες καθώς εξαρτώνται από την χρονική στιγμή όπου γίνεται το test στο ενδιάμεσο της ακολουθίας εισαγωγών.

Για τον ίδιο λόγο καθώς αυξάνει απότομα το ύψος του δένδρου οι μετρήσεις Pages Read μπορεί να είναι παραπλανητικές. Να σημειωθεί ότι ο αλγόριθμος split της εργασίας προσπαθούσε να κρατάει τα regons κυβοειδή (γινόταν random επιλογή ανάμεσα στην βελτίωση του cubicality τα λίγα splits και το καλό utilization). Αυτό πιθανόν εξηγεί γιατί το split γίνεται νωρίτερα από ότι στον Robinson καθώς εκεί

πιθανόν ο αλγόριθμος να έχει βελτιστοποιηθεί ώστε να ελαχιστοποιεί τα splits.

Σε κάθε περίπτωση για τις δύο διαστάσεις μοιάζει να ισχύει η παρατήρηση του Robinson ότι ο μέσος αριθμός Reads είναι γύρω στο ύψος του δένδρου ενώ ο μέσος αριθμός Writes είναι γύρω στο 1.Αυτό από τις δοκιμές φάνηκε να ισχύει και σε περισσότερες διαστάσεις. Το ότι ο μέσος όρος των writes είναι ίσος με 1 εξηγείται από το ότι αν και χρειάζονται επιπλέον διασπάσεις σε κάποιες εισαγωγές σημείων, παρ όλα αυτά στις περισσότερες εισαγωγές γίνεται μόνο εισαγωγή του σημείου σε κάποιο point node χωρίς διάσπαση, και έτσι ο μέσος όρος τείνει στο 1. Ομοίως οι διασπάσεις είναι σχετικά σπάνιες και δεν επηρεάζουν το αναμενόμενο πλήθος reads που είναι ένα για κάθε επίπεδο του δένδρου.

Το κύριο μέτρο σύγκρισης είναι το utilization το οποίο είναι και στις δύο περιπτώσεις κοντά στο 0.60 και μοιάζει να χαμηλώνει στο 0.55 για 3 διαστάσεις στην υλοποίηση του Robinson και στο 0.52 σε αυτήν της εργασίας.. Αυτά τα utilization rates είναι αρκετά καλά και είναι κοντά στα worst case guarantees του b-tree, πάντως πρέπει να ληφθεί υπόψη ότι και στις δύο περιπτώσεις τα δεδομένα είναι ομοιόμορφα κατανεμημένα κάτι που μάλλον ευνοεί το υψηλό utilization.

Το γεγονός οτι η δομή που εξετάζεται τείνει να κάνει γρηγορότερα τα splits έχει επίπτωση και στο utilization το οποίο πέφτει (τουλάχιστον προσωρινά). Ίσως υπάρχει ομως κέρδος λόγω των πιο κυβοειδών σχημάτων (εφόσον ο Robinson όντως είχε χρησιμοποιήσει πιο απλό αλγόριθμο από αυτόν της εργασίας). Αυτό θα μπορούσε να αντικατοπτριστεί σε υψηλότερες τιμές του Query Efficiency όμως και πάλι η σύγκριση με την δομή του Robinson δεν θα ήταν εύκολη καθώς δεν είναι γνωστός ο ακριβής αλγόριθμος που χρησιμοποίησε ενώ και τα ερωτήματα range που εκτελούσε ήταν ad hoc. Επίσης στην καταμέτρηση στοιχείων όπως το pages read δεν είναι γνωστό αν χρησιμοποιήθηκε από τον Robinson κάποια Least Recently Used τακτική διατήρησης σελίδων στην μνήμη, κάτι που θα άλλαζε αρκετά τα αποτελέσματα (στην εργασία αυτή μετράται το worst case scenario δηλαδή στην μνήμη χωράει μόνο μια-δύο σελίδες και κάθε βήμα διάσχισης του δένδρου

ισοδυναμεί μέ ένα read)

* * *

Στην συνέχεια έγιναν repeated runs της δομής ώστε (αν και δεν έγινε στατιστική ανάλυση) να φανεί ποια είναι η κατανομή του utilization και αν μπορεί να έχει μεγάλες αποκλίσεις από το 0.60 για τις 2 διαστάσεις, 0.55 για τις 3.

2 διαστάσεις 25 regions 42 points
Utilization values:

0,5796136	0,6253079	0,6370375	0,6571121	0,5901148	0,5258988

3 διαστάσεις 18 regions 31 points
Utilization values:

0,5393059	0,51135	0,4836242	0,5777016	0,5110527	0,5554395

Παρατηρήθηκε ότι η διασπορά είναι μικρή και της τάξεως του 0.10 το πολύ

* * *

Όσον αφορά το στατιστικό Query Efficiency από τα δεδομένα του Robinson κρίθηκε ότι είναι σε μεγάλο βαθμό εξαρτώμενο από το μέγεθος του Query Region, σε αντίθεση με τον ισχυρισμό του Robison οπότε δεν έγινε νέα αναλυτική μελέτη του. Έγινε όμως προσπάθεια λήψης ενός μέσου όρου ώστε να συγκριθεί αυτός με την επιθυμητή τιμή του στατιστικού που είναι η τιμή 1. Όπως και στις δοκιμές του Robinson έγινε αυτή τη φορά η εισαγωγή 100000 σημείων.

2 διαστάσεις 25 regions 42 points

<Utilization>0,5797697

<AverageQueryEfficiency>0,4882991

<Height> 4

3 διαστάσεις 18 regions 31 points

<Utilization>0,5213731

<AverageQueryEfficiency>0,4266692

<Height>5

Παρατηρήθηκε ότι το utilization δεν αλλάζει πολύ κατά την εισαγωγή 100000 αντί για 10000 σημείων. Οι τιμές του QueryEfficieny ήταν σχετικά καλές, μόλις κατά γύρω 60% μειωμένες σε σύγκριση με το ιδανικό (στο paper οι τιμές του στατιστικού αυτού ήταν αρκετά χαμηλότερες για ορισμένα queries αλλά πάνω κάτω ίσες με αυτήν για τα περισσότερα)

Οι αριθμοί reads και writes παρέμειναν περίπου ίσοι του ύψους και της μονάδας αντίστοιχα

* * *

Ο τελευταίος τύπος δοκιμής που έγινε ήτανε η σύγκριση της απόδοσης της δομής καθώς οι διαστάσεις αυξάνονται. Για αυτόν τον σκοπό θεωρήθηκε ένα μέγεθος 4096 byte ανά σελίδα ώστε με βάση αυτήν την τιμή να υπολογίζεται αυτόματα ο μέγιστος δυνατός αριθμός καταχωρήσεων points ή regions σε ένα page με βάση το μέγεθος που έχουν αυτές οι δομές όταν τα δεδομένα είναι ν διαστάσεων. Ο υπολογισμός έγινε με βάση τον ακόλουθο κώδικα (η χρήση της σταθεράς 4 γίνεται γιατί ένα float έχει 4 bytes)

```
int blocksize = 4096;
int regionsize = 2 * 4 * dimensions + 4;
int pointsize = 4 * dimensions;
Node.SIZE = blocksize / regionsize;
Node.SIZEP = blocksize / pointsize;
```

Σε κάθε kb tree έγινε εισαγωγή 100000 σημείων

Τα αποτελέσματα είχαν ως εξής

Διαστάσεις	Sizes	Height	Average Pages Read	Average Pages Written
2	204 512	3	2,32057	1,00547
3	146 341	3	2,61121	1,00993
6	78 170	3	2,92786	1,02562
12	40 85	4	3,51759	1,05643
18	27 56	4	3.91319	1,06997

Διαστάσεις	Utilization	Query Efficieny
2	0,7142705	0,4521922
3	0,5952098	0,4023610
6	0,4722500	0,2109383
12	0,4437590	0,0772450
18	0,5574296	0,0541537

Με βάση αυτά τα δεδομένα μπορεί κανείς να κάνει τις ακόλουθες παρατηρήσεις. Ο αριθμός εγγραφών παραμένει σταθερά στην μια εγγραφή κατά μ.ο. όσες και να είναι οι διαστάσεις. Το ύψος του δένδρου φαίνεται να αυξάνεται αλλά σίγουρα όχι γραμμικά. Το utilization αν και πέφτει από τα επίπεδα του 0.60 παραμένει σχετικά υψηλό και κοντά στο 0.50. Ο αριθμός των αναγνώσεων παραμένει περίπου ίσο με το ύψος του δένδρου

Οι περισσότεροι δείκτες φαίνονται να δείχνουν καλά αποτελέσματα και σε περισσότερες διαστάσεις με εξαίρεη όμως το query efficiency που πέφτει σχετικά απότομα και τείνει στην τιμή μηδέν, αυτήν δηλαδή για την οποία σαρώνεται όλο το δένδρο σε κάθε region query. Δηλαδή αν και σε πολλές διαστάσεις η ταχύτητα εισαγωγής είναι σχετικά σταθερή, η εκτέλεση αναζητήσεων επιβραδύνεται κατά πολύ και ο κύριος λόγος είναι το ότι το ύψος του δένδρου είναι μικρό σε σχέση με το πλήθος των διαστάσεων και έτσι η δομή δεν προλαβαίνει να γίνει selective σε ν διαστάσεις όταν το πλήθος συγκρίσεων που κάνει περιορίζεται από το ύψος και το fan out (το fan out επίσης μικραίνει καθώς όπως είναι αναμενόμενο τα πολυδιάστατα δεδομένα καταλαμβάνουν περισσότερο χώρο σε ένα block).

Καλύτερες δομές για πολυδιάστατα δεδομένα αναφέρονται στο [Gaede and Gunther 1998]

* * *

Τέλος εναλλακτικοί αλγόριθμοι split που δοκιμάστηκαν, με βάση τα 3 κριτήρια της 4.2.1.7 δεν φάνηκε να έχουν κάποια επίπτωση στο utilization. Αν υπάρχει κάποια θετική διαφοροποίηση αυτή ίσως έχει να κάνει με το πόσο κυβοειδή είναι τα regions, κάτι που θα βελτιώνει τον χρόνο εκτέλεσης των queries. Διαισθητικά στην γραφική αναπαράσταση της εφαρμογής για δύο διαστάσεων δεδομένα, η βελτίωση του σχήματος τον ψηλών σε επίπεδο regions ήταν αντιληπτή. Πιθανόν η μη σημαντική διαφορά ανάμεσα στην χρήση των τριών κριτηρίων να οφείλεται και στην

ομοιόμορφη κατανομή των δεδομένων καθώς αυτή αποτελεί μια λιγότερο προβληματική περίπτωση.και σε skewed δεδομένα να χρειάζεται καλύτερη απόφαση για το που θα γίνει η διάσπαση.

* * *

Κρίνοντας το kdb tree με βάση τα πειραματικά δεδομένα βλέπουμε ότι σε γενικές γραμμές επαληθεύεται ο ισχυρισμός του Robinson για utilization κοντά στα επίπεδα του b-tree τουλάχιστον στις δύο ή τρείς διαστάσεις. Τα αποτελέσματα αυτά όμως προέκυψαν για ομοιόμορφα κατανεμημένα δεδομένα και δεν είναι γνωστό αν αυτού του είδος το σύνολο εισαγωγής επηρεάζει θετικά τις μετρήσεις utilization. Στα πλαίσια αυτής της εργασίας δεν κρίθηκε αναγκαία η δοκιμή με skewed non uniform (π.χ. gaussian mixture) δεδομένα.

9. ΒΙΒΛΙΟΓΡΑΦΙΑ

John Robinson (1981), *The K-D-B Tree: A search structure for large multidimensional dynamic indexes*, Proceedings of the 1981 ACM SIGMOD international conference on Management of data

Douglas Cromer (1979) , *The Ubiquitous B-Tree,* ACM Computing Surveys (CSUR), v.11 n.2, p.121-137, June 1979

Andrew Moore(1991), *An introductory tutorial on kd trees*, Extract from Andrew Moore's PHD Thesis: Efficient Memory based Learning for Robot Control, PHD Thessis Technical Report No.209, Computer Laboratory, University of Cambridge

Gaede and Gunther (1998), *Multidimensional Access Methods ,* ACM Computing Surveys (CSUR) Volume 30, Issue 2 (June 1998) Pages: 170 - 231

Lomet and Salzberg (1989) , *The hB-tree: A robust multiattribute search structure*

Andrew Troelsen (2007), *Pro C# 2008 and the .NET Platform, Apress*

Wikipedia contributors, "*XML*", *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=XML&oldid=342245802 (accessed February 16, 2010).

Miroslav Nic, *XSLT Reference*, Retrieved from WWW 16 February 2010 (http://www.zvon.org/xxl/XSLTreference/Output/index.html)

Elliotte Rusty Harold & W. Scott Means (2001), *XML in a Nutshell*, O'Reilly (από http://oreilly.com/catalog/xmlnut/chapter/ch09.html)

Bernard Meyer (1991), *Applying Design By Contract*, Computer, v.25 n.10, p.40-51, IEEE Computer Society Press (από http://se.ethz.ch/~meyer/publications/computer/contract.pdf)

MSDN XAML Tutorial

(από http://msdn.microsoft.com/en-us/library/ms752059.aspx)

MSDN WPF Tutorial

(από http://msdn.microsoft.com/en-us/library/ms754130.aspx)

Εργαλεία προγραμματισμού

MS Visual C# Express 2008

Mindfusion XMLViewer (http://download.cnet.com/XML-Viewer/3000-7241_4-10223729.html)

Nunit (http://www.nunit.org/index.php)

.NET 3.5

(http://www.microsoft.com/downloads/details.aspx?familyid=ab99342f-5d1a-413d-8319-81da479ab0d7&displaylang=en)