ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δεικτοδότηση μονοδιάστατων δεδομένων σε μνήμη φλάς

Indexing 1d data on flash memory

Διπλωματική Εργασία

Δεληγιάννη Ισαβέλλα

Επιβλέποντες Καθηγητές: Μποζάνης Παναγιώτης
Αναπληρωτής Καθηγητής

Τσομπανοπούλου Παναγιώτα
Επίκουρη Καθηγήτρια

Βόλος, Σεπτέμβριος 2013

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της διπλωματικής εργασίας κ. Μποζάνη Παναγιώτη – Αναπληρωτή Καθηγητή και την κα. Τσομπανοπούλου Παναγιώτα – Επίκουρη Καθηγήτρια για την καθοδήγηση τους, τις πολύτιμες συμβουλές τους και τη στήριξη τους.

Θα ήθελα επίσης να ευχαριστήσω θερμά τον κ. Φεύγα Αθανάσιο, για τον χρόνο και τη βοήθεια που μου αφιέρωσε προκειμένου να ολοκληρωθεί η παρούσα διπλωματική εργασία.

Οφείλω ένα μεγάλο ευχαριστώ στην οικογένεια μου, για την αγάπη τους, την υπομονή τους και τη στήριξη τους στο πρόσωπο μου όλα αυτά τα χρόνια και ιδιαίτερα στη μητέρα μου.

Τέλος, σημαντικό ρόλο κατείχαν οι φίλοι μου. Τους ευχαριστώ πολύ που είναι πάντα δίπλα μου και με στηρίζουν στα βήματα μου όλα αυτά τα χρόνια.

Βόλος, Σεπτέμβριος 2013

ΠΕΡΙΕΧΟΜΕΝΑ

1. ***Εισαγωγικά-Βασικές Έννοιες μνήμης Flash***

   1.1 *Εισαγωγή*
   1.2 *Χαρακτηριστικά της μνήμης Flash*
      1.2.1 NAND Μνήμη Flash
      1.2.2 NOR Μνήμη Flash
      Σύγκριση
   1.3 *Δίσκοι SSD VS Δίσκοι HDD*

   Βιβλιογραφία

2. ***B^+-δέντρα - Δομές Δεδομένων ευρετηρίου σε μνήμη Flash***

   2.1 *Βασικές αρχές στις δομές δεδομένων ευρετηρίου.*
   2.2 *Εισαγωγή στα B^+-δέντρα*
   2.3 *Παρουσίαση ψευδοκώδικα.*
   2.5 *Παραδείγματα πολλαπλών λειτουργιών, βήμα προς βήμα.*
   2.5 *Κατηγορίες B^+-δέντρων.*
      - *B^+-δέντρα Δίσκου*
      - *B^+-δέντρα Ημερολογίου*

   Βιβλιογραφία

3. ***Αυτορυθμιζόμενη ευρετηρίαση - B^+-δέντρα(ST)***

   3.1. *FlashDB – Βάση δεδομένων για δίκτυα αισθητήρων βασισμένα σε μνήμη Flash – Αρχιτεκτονική της FlashDB*
   3.2. *Εισαγωγή - Σχεδίαση αυτορυθμιζόμενων B^+-δέντρων(**B^+-trees(ST)**)*
   3.3. *Περιοχή Προσωρινής Αποθήκευσης Ημερολογίου(Log Buffer) - Πίνακας Μετάφρασης Κόμβων(NTT)*
   3.4. *Βασικές Λειτουργίες των B^+-δέντρων(ST)*
   3.5. *Δομή Τομέα - NTT - Εγγραφών Ημερολογίου*

3.6. *Βασικά Ζητήματα Αυτορύθμισης*

3.6.1.1. *Αλγόριθμος Εναλλαγής Κατάστασης*

3.7. *Βελτιστοποίηση B^+-δέντρων(ST)*

3.7.1.1. *Συμπίεση Ημερολογίου - Συλλογή Απορριμμάτων*

3.7.1.2. *Μηχανισμός Συλλογής Απορριμμάτων Ημερολογίου (LGC) VS Μηχανισμός Συλλογής Απορριμμάτων του Διαχειριστή Αποθήκευσης (Garbage Collection in storage manager(GC))*

3.8. *Εκτίμηση Απόδοσης της FlashDB*

Βιβλιογραφία

4. BFTL : Μείωση των λειτουργιών εγγραφής στους δίσκους SSD, υποβαθμίζοντας την επίδοση της αναζήτησης συγκριτικά με εκείνης των B^+-δέντρων.

4.1. *Εισαγωγή στο στρώμα BFTL*

4.2. *Σχεδιασμός και υλοποίηση του Στρώματος BFTL*

4.2.1. *Φυσική αναπαράσταση του κόμβου ενός B-δέντρου: Μονάδα Εγγραφής*

4.3. *Πολιτική Αποθήκευσης*

Θεώρημα 1

Αλγόριθμος 1

4.4. *Ο Πίνακας Μετάφρασης Κόμβων(Node Translation Table)*

Αλγόριθμος 2

4.5. *Ανάλυση Πολυπλοκότητας BFTL*

4.5.1. *Εκτίμηση Απόδοσης BFTL*

4.5.2. *Απόδοση δημιουργίας Δομών Ευρετηρίου B-δέντρων*

4.5.3. *Απόδοση Διατήρησης B-δέντρων*

4.5.4. *Αποτελέσματα Πειραμάτων σχετικά με την απόδοση του FTL και BFTL κατά την αναζήτηση στοιχείων σε B-δέντρα*

Βιβλιογραφία

5. ***Δομές ευρετηρίων που επικεντρώνονται στην εκμετάλλευση των διαδοχικών λειτουργιών εισόδου/εξόδου στους δίσκους SSD, αντικαθιστώντας τις τυχαίες εγγραφές με τις διαδοχικές.***

   5.1. *Εισαγωγή στα AS B-δέντρα*
   5.2. *AS B-δέντρα* ***VS*** *LA-δέντρα, BFTL, FD-δέντρα*
   5.3. *Ανάλυση AS B-δέντρου*
      5.3.1. *Sequential Writer - SW*
         Αλγόριθμος 1
      5.3.2. *Write Buffer - WB*
         Αλγόριθμος 2
         Αλγόριθμος 3
      5.3.3. *Address Mapping Table (AMT - Πίνακας Αντιστοιχήσεων Διευθύνσεων)*
      5.3.4. *Node Validation Manager - NVM*
      5.3.5. *Απόδοση Πειραμάτων*

   Βιβλιογραφία

6. ***FlashB- δέντρα: Καινοτόμο σχήμα εφαρμογής δομής ευρετηρίου, με σκοπό το μηδενισμό του αριθμού των αναδιοργανώσεων του B-δέντρου.***

   6.1. *Εισαγωγή στα FlashB-δέντρα*
   6.2. *Αρχές Σχεδίασης*
      6.2.1. *Πολιτική Διαίρεσης*
         Στρατηγική Modify-Two-Node
      6.2.2. *Πολιτική Συνένωσης/Περιστροφής*
         Στρατηγική Lazy-Coalescence
   6.3. *Σχεδίαση του Online Transition Algorithm*
   6.4. *Αξιολόγηση Επίδοσης*
   6.5. *Αποτελέσματα Πειραμάτων*

   Βιβλιογραφία

7. ***LA- δέντρα:*** **Στόχος η βελτίωση της επίδοσης των συστημάτων, *ελαχιστοποιώντας όσο είναι εφικτό την πρόσβαση στη μνήμη Flash.***

7.1. *Εισαγωγή στα LA-δέντρα*

7.2. *Σχεδιασμός ενός LA-δέντρου*

7.2.1 *Cascaded Buffers*

7.2.2 *Adaptive Buffering*

7.3. *Περιγραφή των βασικών λειτουργιών σε ένα LA-δέντρο*

7.3.1 *Λειτουργίες Εισαγωγής και Διαγραφής*

7.3.2 *Λειτουργίες Αναζήτησης*

7.4. *Λειτουργίες εκκαθάρισης των buffer*

7.5. *Ο Αλγόριθμος ADAPT*

Βιβλιογραφία

ΚΕΦΑΛΑΙΟ 1

1.1 Εισαγωγή

Ένα από τα μεγαλύτερα επιτεύγματα που έχουν γίνει μέχρι σήμερα στον τομέα των υπολογιστών και συγκεκριμένα στην αποθήκευση και μεταφορά δεδομένων, είναι η μνήμη Flash. Στις μέρες μας, η μνήμη Flash συναντάται σε συσκευές παλάμης, κινητά τηλέφωνα, ψηφιακές κάμερες, κονσόλες παιχνιδιών, κάρτες μνήμης, MP3 players και πολλά άλλα μέσα τα οποία έχουν γίνει αναπόσπαστο κομμάτι της ζωής μας. Στις περισσότερες συσκευές παλάμης, μνήμη Flash χρησιμοποιείται ως εξωτερική μνήμη εξαιτίας κυρίως της χαμηλής της τιμής και του συνεχώς αυξανόμενου μεγέθους της. Η μνήμη Flash αποτελεί σήμερα μια ευρέως αναπτυσσόμενη τεχνολογία με όλο και μεγαλύτερο αντίκτυπο στην καθημερινή μας ζωή.

Η τεχνολογική πρόοδος που σημειώνεται στα ενσωματωμένα συστήματα (embedded systems) καθώς και στα δίκτυα αισθητήρων (sensor networks), βασίζεται σε συσκευές μικρής κλίμακας (small-scale devices) στις οποίες οι μαγνητικοί δίσκοι (magnetic disks) είναι ακατάλληλοι. Συνεπώς είναι αναγκαία η σχεδίαση αλγορίθμων και δομών δεδομένων χαμηλής πολυπλοκότητας με μικρές απαιτήσεις σε μνήμη.

Ένα από τα βασικότερα προβλήματα στο χώρο των υπολογιστών αποτελεί η αποθήκευση του μεγάλου όγκου πληροφορίας και συνεπώς η αποδοτική αναζήτηση και ανάκτηση αυτής. Μέχρι σήμερα έχουμε αναπτύξει ποικίλους αλγόριθμους και δομές δεδομένων κύριας μνήμης προσπαθώντας να επιλύσουμε αυτό το πρόβλημα. Στην περίπτωση όμως που ο όγκος δεδομένων είναι τόσο μεγάλος ώστε να μην χωράει στην κύρια μνήμη του συστήματος, η ανάγκη για χρήση αλγορίθμων εξωτερικής μνήμης κρίνεται επιτακτική. Για την ακρίβεια, η διαδικασία που ακολουθείται είναι η αποθήκευση των δεδομένων σε εξωτερικά συστήματα αποθήκευσης (external storage systems) και έπειτα η προσπέλαση αυτών από κατάλληλους αλγορίθμους προκειμένου να εφαρμοστούν οι επιθυμητές λειτουργίες. Πολλές από τις δομές δεδομένων που εφαρμόζονται στην κύρια μνήμη έχουν πλέον επεκταθεί και για χρήση σε εξωτερική μνήμη.

Τι γίνεται όμως στην περίπτωση των κινητών συσκευών που διακρίνονται για την χαμηλή επεξεργαστική ισχύ και την χαμηλή σε χωρητικότητα κύρια μνήμη?

Λόγω της μικρής σε χωρητικότητα κύριας μνήμης που έχουν οι συσκευές αυτές, είναι σαφές ότι δε μπορούν να διαχειριστούν μεγάλο όγκο πληροφορίας με αποτέλεσμα να κρίνεται αναγκαία η χρήση εξωτερικής μνήμης. Στις περισσότερες συσκευές παλάμης, η μνήμη Flash (flash memory) χρησιμοποιείται σαν εξωτερική μνήμη, κυρίως λόγω της χαμηλής της τιμής και του συνεχώς αυξανόμενου μεγέθους της. Τα τελευταία χρόνια γίνονται όλο και περισσότερες τροποποιήσεις κλασσικών δομών δεδομένων εξωτερικής μνήμης (όπως τα B-δέντρα), προκειμένου να είναι συμβατές με μνήμη Flash.

1.2 Χαρακτηριστικά της μνήμης Flash

Η μνήμη Flash είναι ένας **EEPROM** (*Electronically Erasable Programmable Read-Only Memory*) **τύπος μνήμης**. Είναι μη ευμετάβλητη (*non-volatile*) (διατηρεί δηλαδή τα δεδομένα της και χωρίς την παροχή ρεύματος), με αποτέλεσμα να αποτελεί τον ιδανικό τύπο μνήμης για τη μόνιμη αποθήκευση δεδομένων σε κινητά τηλέφωνα και συσκευές παλάμης. Σε αντίθεση με τις παλαιότερες μνήμες EPROM (*Erasable Programmable Read-Only Memory*), οι EEPROM διαγράφονται και προγραμματίζονται ανά λέξη με ηλεκτρικές μεθόδους, χωρίς τη χρήση υπεριώδους ακτινοβολίας. Αυτό σημαίνει ότι δεν είναι απαραίτητη η απομάκρυνση της μνήμης από το σύστημα λειτουργίας για να προγραμματιστεί.

Παρουσιάζει μια μοναδική συμπεριφορά ανάγνωσης/εγγραφής/διαγραφής σε σύγκριση με τις υπόλοιπες μνήμες. Πιο αναλυτικά, ο αριθμός των εγγραφών που πραγματοποιούνται σε μια μνήμη Flash είναι συγκεκριμένος. Ποικίλλει από 10.000 έως 1.000.000, ενώ όταν ξεπεραστούν αυτά τα όρια η μνήμη Flash φθείρεται και δε μπορεί να χρησιμοποιηθεί περαιτέρω.

Διακρίνεται για την αυξημένη αντοχή της σε δονήσεις (shock resistant), σε αντίθεση με τους σκληρούς δίσκους καθώς και για την ελάχιστη απαίτηση

κατανάλωσης ενέργειας (energy-efficient). Προσφέρει επίσης γρήγορους χρόνους πρόσβασης για ανάγνωση, χωρίς ωστόσο να είναι ταχύτερη από την κύρια μνήμη DRAM ενός συμβατικού υπολογιστή (laptop/desktop).

Υπάρχουν δύο βασικές κατηγορίες μνήμης Flash. Η **NOR μνήμη Flash** και η **NAND μνήμη Flash**. Τα ονόματα αποδίδονται στον τύπο των λογικών πυλών που χρησιμοποιούνται σε κάθε κελί της μνήμης (όπου οι λογικές πύλες είναι ένα block από ψηφιακά κυκλώματα). Η NOR μνήμη Flash εισήχθηκε το 1998 από την Intel και η NAND μνήμη Flash το 1989 από την Toshiba. Οι δύο αυτοί τύποι μνήμης λειτουργούν διαφορετικά. Η NOR μνήμη Flash είναι μεν γρηγορότερη αλλά πιο ακριβή. Χρειάζεται επίσης περισσότερο χρόνο για τη λειτουργία της διαγραφής και εγγραφής δεδομένων σε συγκεκριμένες διευθύνσεις μνήμης. Η NOR μνήμη Flash χρησιμοποιείται συνήθως σε κινητά τηλέφωνα. Η NAND μνήμη Flash έχει αισθητά υψηλότερη χωρητικότητα αποθήκευσης σε σχέση με την NOR.

Και στους δύο αυτούς τύπους μνήμης, οι λειτουργίες εγγραφής επαναφέρουν την τιμή των bits στην αρχική τιμή 0 (*clear bits* – αλλάζουν την τιμή από 0 σε 1). Έπειτα, ο μοναδικός τρόπος να αλλάξουν τιμή τα bits (*set bits* – αλλαγή της τιμής από 0 σε 1), είναι να σβηστεί μια ολόκληρη περιοχή μνήμης. Αυτή η περιοχή μνήμης ονομάζεται μονάδα σβησίματος (***erase unit***) και είναι καθορισμένου μεγέθους.

Εξαιτίας όλων των παραπάνω χαρακτηριστικών λοιπόν, υπάρχουν ειδικές δομές δεδομένων και αλγόριθμοι, προκειμένου να είναι αποδοτική η μνήμη Flash. Αυτοί οι αλγόριθμοι και οι δομές δεδομένων υποστηρίζουν ανανέωση δεδομένων σε διαφορετική θέση (not-in-place updates of data), εξισορροπούν τη φθορά των μπλοκ της μνήμης και μειώνουν τον αριθμό των διαγραφών. Παρουσιάζονται παρακάτω πιο αναλυτικά οι 2 τύποι μνήμης Flash. [3]

1.2.1 NAND Μνήμη Flash

Ένα chip NAND μνήμης Flash, αποτελείται από ένα καθορισμένο αριθμό από μπλοκ (*block*), το οποία με τη σειρά τους αποτελούνται από ένα αριθμό σελίδων (*pages*). Κάθε σελίδα αποτελείται από μία περιοχή με κύρια δεδομένα (*main data area*) και μια ελεύθερη περιοχή (*space area*). Η ελεύθερη περιοχή κάθε σελίδας, χρησιμοποιείται για την αποθήκευση κώδικα διόρθωσης λαθών (*Error*

Correction Code - ECC) και για πληροφορίες διαχείρισης. Επιπλέον, κάθε μπλοκ μιας τυπικής NAND μνήμης Flash, μπορεί να σβηστεί (*erased*) έως 1.000.000 φορές. Στην περίπτωση που το όριο αυτό ξεπεραστεί, το μπλοκ φθείρεται (*worn out*) και υφίσταται λάθη κατά την εγγραφή.

Έως σήμερα, έχουν προταθεί διάφοροι τύποι NAND μνήμης Flash. Η πρώτη **Μονού-Επιπέδου Κελιού (Single-Level Cell – SLC)** μνήμη NAND, είναι οργανωμένη με τέτοιον τρόπο, ώστε κάθε μπλοκ να περιέχει 32 σελίδες και το μέγεθος κάθε σελίδας να ορίζεται ως (512+16) bytes συμπεριλαμβανομένων και των 16 bytes της ελεύθερης περιοχής. Η αμέσως επόμενη γενιά NAND μνήμης Flash, ονομάζεται **''μεγάλου μπλοκ SLC NAND'' (large block SLC NAND)** και προσφέρει μεγάλη χωρητικότητα. Με λίγα λόγια, ο αριθμός σελίδων σε ένα μπλοκ στην δεύτερη κατηγορία είναι διπλάσιος συγκριτικά με αυτόν της πρώτης και το μέγεθος της σελίδας είναι κατά τέσσερις φορές μεγαλύτερο.

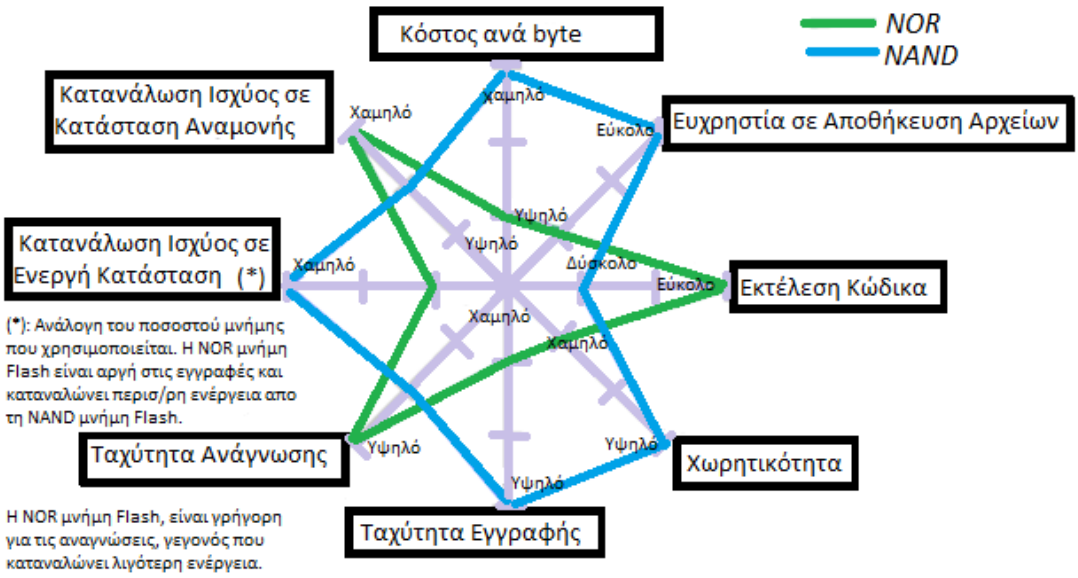
Αμέσως μετά εμφανίζεται η **Πολλαπλού-Επιπέδου Κελιού (Multiple-Level Cell MLC NAND)**, στην οποία κάθε τρανζίστορ μπορεί να βρίσκεται σε μία από τις 4 διαφορετικές καταστάσεις, παρέχοντας τη δυνατότητα κωδικοποίησης δεδομένων, με σκοπό την αποθήκευση δύο bit ανά κελί μνήμης (memory cell). Το γεγονός αυτό διπλασιάζει τη χωρητικότητα της μνήμης NAND μνήμης Flash. Συνεπώς, στην MLC NAND μνήμη Flash, κάθε σελίδα αποθηκεύει 4096 bytes δεδομένων, ενώ το μέγεθος της ελεύθερής της περιοχής είναι 128 bytes. Ο αριθμός των σελίδων που περιέχονται σε ένα block ανέρχονται στις 128. [1]

1.2.2 NOR Μνήμη Flash

Η NOR μνήμη Flash παρουσιάστηκε πρώτη εκ των 2 τύπων και παρέχει τυχαία προσπέλαση (random access). Τα περιεχόμενα των μνημών αυτών μπορούν να προσπελαστούν ή να προγραμματιστούν ανά λέξη, αλλά η διαγραφή (όλα τα bits τίθενται σε '1') γίνεται ανά μπλοκ διευθύνσεων. Διευθυνσιοδοτείται δηλαδή σε επίπεδο byte, από την Κεντρική Μονάδα Επεξεργασίας (CPU) και αν χρειαστεί μπορεί να χρησιμοποιηθεί σαν κύρια μνήμη RAM. Κυρίως χρησιμοποιείται για την αποθήκευση στατικών δεδομένων αφού παρουσιάζει υψηλούς χρόνους εγγραφής. Ως κύριο πλεονέκτημα της NOR μνήμης Flash, είναι

η εγγραφή των δεδομένων σε επίπεδο byte σε αντίθεση με τη NAND μνήμη Flash που υποστηρίζει εγγραφή δεδομένων σε επίπεδο σελίδας. Επιπρόσθετα, η NOR μνήμη Flash, παρέχει ταχύτερους χρόνους προσπέλασης δεδομένων (~ 200ns) σε σύγκριση με την NAND μνήμη Flash (50-80μs), υστερώντας όμως σε πυκνότητα (density) και αποδοτικότητα ισχύος (power efficiency). [2]

Παρουσιάζεται παρακάτω ένα συνοπτικό διάγραμμα που δείχνει αναλυτικά τα χαρακτηριστικά των δύο αυτών τύπων μνήμης:



Σχήμα 1: Σύγκριση Χαρακτηριστικών NAND και NOR μνήμης Flash

1.3 Δίσκοι SSD VS Δίσκοι (HDD)

Ο σκληρός δίσκος Flash αποτελεί ένα σχετικά γρήγορο φορέα δεδομένων. Τα ανθεκτικά αυτά μέσα αποθήκευσης διαθέτουν μάλλον μικρό χρόνο ζωής.

Ο σκληρός δίσκος ενός υπολογιστή αποτελεί, εκτός από τον επεξεργαστή και τον εσωτερικό δίσκο (RAM), ένα από τα σημαντικότερα στοιχεία του. Εδώ δεν αποθηκεύονται μόνο όλα τα αρχεία, αλλά και το ίδιο το λειτουργικό σύστημα. Εκτός από τους κοινούς σκληρούς δίσκους, τους λεγόμενους HDD (Hard Disk Drive = μονάδα σκληρού δίσκου – Εικόνα 1.3.2) διατίθενται σήμερα και σκληροί δίσκοι Flash. Ανήκουν στην ομάδα των λεγόμενων SSD (Solid State Drive/Disk= δίσκος στερεάς κατάστασης – Εικόνα 1.3.1).

Σε σύγκριση με άλλους σκληρούς δίσκους δεν διαθέτουν καθόλου περιστρεφόμενο δίσκο. Επίσης, στο σκληρό δίσκο Flash δεν υπάρχουν καθόλου μηχανικά μέρη. Έτσι αυτές οι συσκευές δε ζεσταίνονται, δεν κάνουν θόρυβο, δεν κινδυνεύουν να ‘’αστοχήσουν’’ (να χαλάσουν ή να καταστρέψουν δεδομένα) λόγω δονήσεων, εξοικονομούν ρεύμα και παρέχουν τάχιστη πρόσβαση στα δεδομένα. Ωστόσο, οι σκληροί δίσκοι Flash είναι πιο αργοί κατά την εγγραφή δεδομένων και διαθέτουν περιορισμένη διάρκεια ζωής.

Ο τρόπος δομής των σκληρών δίσκων Flash τους καθιστά γρήγορους και οικονομικούς στην κατανάλωση ρεύματος

Στους σκληρούς δίσκους Flash δεν απαιτείται η κινητοποίηση του μαγνητικού δίσκου για την πρόσβαση στα δεδομένα, όπως συμβαίνει με τους κοινούς δίσκους. Εδώ αρκεί ένας ηλεκτρικός παλμός για τον έλεγχο της αντίστοιχης περιοχής.

Τα Flash αποτελούν ένα είδος EEPROM τσιπ (Electronically Erasable Programmable Read Only Memory), που σημαίνει πως τα δεδομένα στη μνήμη μπορούν να εγγραφούν ή να διαγραφούν με μια μόνο κίνηση μέσω προγράμματος.

Ένας σκληρός δίσκος με 7.200 περιστροφές ανά λεπτό απαιτεί για την ανάκληση δεδομένων από δέκα έως 15 χιλιοστά του δευτερολέπτου, ενώ ο χρόνος απόκρισης για την ανάγνωση δεδομένων στο σκληρό δίσκο Flash κυμαίνεται στα 0,1 χιλιοστά του δευτερολέπτου. Έτσι είναι περίπου 100 φορές γρηγορότεροι, όσον αφορά την ανάγνωση δεδομένων.

Οι σκληροί δίσκοι Flash, ωστόσο, δεν είναι ιδιαίτερα γρήγοροι στην εγγραφή δεδομένων: Πριν την εγγραφή κάποιου τμήματος των δεδομένων απαιτείται η ολική διαγραφή αυτής της περιοχής.

Μικρός χρόνος ζωής εξαιτίας του περιορισμένου αριθμού επανεγγραφών

Η κατασκευή του σκληρού δίσκου Flash διαθέτει ένα ακόμη μειονέκτημα: Τα δεδομένα δεν μπορούν να διαγραφούν και να μετεγγραφούν άπειρες φορές. Τα περισσότερα προϊόντα αντέχουν περίπου 100.000 επανεγγραφές.

Κατάλληλα μέσα αποθήκευσης για λειτουργικά συστήματα και προγράμματα

Οι σκληροί δίσκοι Flash είναι κατάλληλοι ιδιαίτερα για την αποθήκευση λειτουργικών συστημάτων και προγραμμάτων. Με την εγκατάσταση ενός σκληρού δίσκου Flash σε κάποιον παλιό ή αδύναμο υπολογιστή επιταχύνεται σημαντικά η ταχύτητα λειτουργίας, εφόσον τα δεδομένα διαβάζονται σημαντικά ταχύτερα. Οι δίσκοι Flash δεν ενδείκνυνται, ωστόσο, για την αποθήκευση αρχείων. Αν χρησιμοποιηθεί ο δίσκος Flash για την επεξεργασία εικόνας, ως μέσο αποθήκευσης δηλαδή αρχείων που αλλάζουν συχνά, αυτό θα μειώσει τη διάρκεια της ζωής του σκληρού δίσκου Flash και θα αυξήσει το συνολικό χρόνο εργασίας.

Η ταχύτητα μεταφοράς των δεδομένων στους σκληρούς δίσκους Flash μπορεί να επιβραδυνθεί μετά από μακρόχρονη χρήση, εφόσον στο σκληρό δίσκο Flash διατίθενται πλέον ολοένα και λιγότερα ελεύθερα μπλοκ δεδομένων. Οι σκληροί δίσκοι Flash απαιτούν πολύ χρόνο για την επανεγγραφή ήδη χρησιμοποιημένων μπλοκ δεδομένων.

Ακριβό μέσο αποθήκευσης

Οι σκληροί δίσκοι Flash είναι σχετικά ακριβοί. Έτσι, ένας σκληρός δίσκος Flash στα 128 GB μπορεί να κοστίσει μερικές εκατοντάδες Ευρώ, ενώ ο κοινός σκληρός δίσκος με χωρητικότητα δύο Terabyte (περίπου 2.000 GB) διατίθεται λίγο παραπάνω από 100 Ευρώ. [4]

Εικόνα 1.3.1: Δίσκος SSD [5]

Εικόνα 1.3.2: Δίσκος HDD [5]

Βιβλιογραφία:

[1] http://en.wikipedia.org/wiki/Multi-level_cell

[2] http://whatis.techtarget.com/definition/NOR-flash-memory

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

[4]http://www.mediamarkt.gr/mp/article/%CE%A3%CE%BA%CE%BB%CE%B7%CF%81%CF%8C%CF%82-%CE%B4%CE%AF%CF%83%CE%BA%CE%BF%CF%82-Flash,792021.html

[5]http://www.adslgr.com/forum/threads/613942-%CE%A4%CE%B9-%CE%B5%CE%AF%CE%BD%CE%B1%CE%B9-SSD-%CE%BC%CE%BD%CE%AE%CE%BC%CE%B5%CF%82-NAND

[6] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Flash_memory.html

[7] http://features.techworld.com/storage/3227303/what-is-flash-memory/

[8] http://www.wisegeek.com/what-is-the-difference-between-flash-memory-and-a-hard-drive.htm

[9]https://discussions.apple.com/thread/4038323?start=0&tstart=0

ΚΕΦΑΛΑΙΟ 2

2.1 Βασικές αρχές στις δομές δεδομένων ευρετηρίου

Στην επιστήμη των υπολογιστών, η έννοια της **δομής δεδομένων** αναφέρεται στους διαφορετικούς δυνατούς τρόπους οργάνωσης και αποθήκευσης δεδομένων μέσα σε έναν υπολογιστή, ώστε τα δεδομένα αυτά να μπορούν να χρησιμοποιηθούν αποδοτικά. Για παράδειγμα ένα σύνολο από δεδομένα μπορεί να αποθηκευτεί σε δομή πίνακα, στοίβας, συνδεδεμένης λίστας, σωρού, ουράς και ούτω καθ εξής.

Σε συγκεκριμένες εφαρμογές χρησιμοποιούνται συγκεκριμένες δομές δεδομένων, δηλαδή αυτές που είναι οι πιο αποδοτικές αλγοριθμικά για το κάθε είδος εφαρμογής. Για παράδειγμα, τα B-δέντρα χρησιμοποιούνται σε υλοποιήσεις βάσεων δεδομένων.

Δομές δεδομένων χρησιμοποιούνται σχεδόν σε κάθε πρόγραμμα ή σύστημα λογισμικού. Παρέχουν έναν τρόπο αποδοτικής διαχείρισης μεγάλου όγκου δεδομένων, όπως μεγάλες βάσεις δεδομένων και υπηρεσίες ευρετηρίου στο διαδίκτυο. Οι αποδοτικές δομές δεδομένων θεωρούνται συχνά ιδιαίτερα σημαντικές στη δημιουργία ενός αποδοτικού αλγορίθμου, σε τέτοιο βαθμό, ώστε κάποιες μέθοδοι σχεδίασης και γλώσσες προγραμματισμού να δίνουν έμφαση σε δομές δεδομένων, παρά σε αλγορίθμους, ως το βασικό κριτήριο σχεδίασης λογισμικού. [1]

Καθώς όμως η μνήμη Flash καθιερώνεται όλο και περισσότερο ως αποθηκευτικό μέσο σε πολλά ενσωματωμένα συστήματα, πολλά συστήματα αρχείων (Database Management Systems-DBMS) καθώς και συστήματα διαχείρισης βάσεων δεδομένων οικοδομούνται πάνω της. Εξαιτίας των χαρακτηριστικών όμως της μνήμης Flash όπως παρουσιάστηκαν, οι κλασσικοί σχεδιασμοί των δομών ευρετηρίου μειώνουν την απόδοση των συστημάτων αποθήκευσης (Flash memory storage systems).

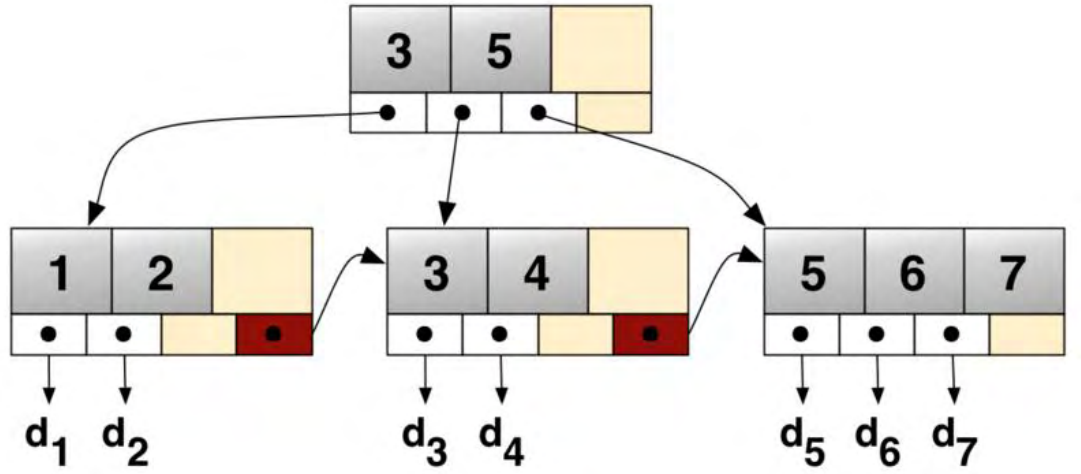
Το γεγονός ότι η μνήμη Flash δε μπορεί να επανεγγραφεί (overwritten/updated) αν πρώτα δεν έχει υποστεί διαγραφή, ορίζεται ένας από τους βασικότερους λόγους που οδηγεί στην μη αποτελεσματική χρήση των κλασσικών δομών ευρετηρίου στη μνήμη Flash. Αυτό έχει ως αποτέλεσμα τη συνύπαρξη μη έγκυρων εκδόσεων δεδομένων (invalid versions) μαζί με τις τελευταίες εκδόσεις αυτών (latest versions). Επιπρόσθετα οι συχνές διαγραφές συγκεκριμένων θέσεων της μνήμης Flash, μειώνουν ταχύτατα και

τη διάρκεια ζωής της αφού όπως προαναφέρθηκε, κάθε μονάδα σβησίματος έχει έναν περιορισμό στο πλήθος των διαγραφών που μπορεί να υποστεί (limited cycle count on erase operations). [2]

Κρίνεται συνεπώς αναγκαίος ο σχεδιασμός αποδοτικών δομών ευρετηρίου (efficient index structure), οι οποίες θα εντοπίζουν ταχύτατα ένα συγκεκριμένο αντικείμενο από μια μεγάλη ποσότητα εγγραφών καταλόγου ή εγγραφών μιας βάσης δεδομένων.

2.2 B^+-δέντρα και οι κατηγορίες στις οποίες χωρίζονται

Το B^+- δέντρο [9], είναι μια δομή που διαχειρίζεται μεγάλο όγκο δεδομένων. Διατηρεί τα δεδομένα ταξινομημένα παρέχοντας έτσι λογαριθμικούς χρόνους ανάκτησης, εισαγωγής και διαγραφής δεδομένων. Ένα B^+- δέντρο χρησιμοποιείται κυρίως σε συστήματα αρχείων καθώς και συστήματα διαχείρισης βάσεων δεδομένων (DBMS). Υποστηρίζει λειτουργίες εισαγωγής, διαγραφής, αναζήτησης και φόρτωσης δεδομένων. Χρησιμοποιείται κυρίως ως ένα εξωτερικό (στο δίσκο) ευρετήριο (εκτός της RAM) και διατηρεί μεγάλες συλλογές δεδομένων. Το B^+- δέντρο είναι ισοζυγισμένο (balanced), γεγονός που ορίζει τις διαδρομές από τη ρίζα ως τα φύλλα (τελευταίοι κόμβοι) να είναι ισοϋψείς. Συνεπώς είναι προφανές ότι και μετά από κάθε λειτουργία εισαγωγής/ανάκτησης/διαγραφής πρέπει να γίνονται οι απαραίτητες ενέργειες ώστε να παραμένει ισοζυγισμένο. Ένας κόμβος του B^+-δέντρου μπορεί να περιέχει ένα μεγάλο αριθμό από d κλειδιά-τιμές (K_1,K_2,…,K_d) και d+1 δείκτες (P_1,P_2,..,P_{d+1}). Οι τιμές των κλειδιών που αποθηκεύονται σε κάθε κόμβο είναι ταξινομημένες. Τα στοιχεία $\mathbf{K_i}$ αντιπροσωπεύουν τα κλειδιά με τα οποία γίνεται η ταξινόμηση του δέντρου και τα στοιχεία $\mathbf{P_i}$ τους δείκτες στους κόμβους-παιδιά ή σε εγγραφές δεδομένων (στην περίπτωση των κόμβων-φύλλων). Ως τάξη (order) του δέντρου, ορίζεται ο μέγιστος αριθμός κλειδιών d, και το συμβολίζουμε με n. Ισχύει για κάθε κόμβο, εκτός της ρίζας $\lceil n/2 \rceil \leq d \leq n$. Ας δούμε πως γίνεται αναλυτικά η λειτουργία της αναζήτησης, εισαγωγής και διαγραφής σε ένα B^+-δέντρο. Θεωρούμε πάντα ότι δεν έχουμε διπλές εγγραφές (δηλαδή δύο εγγραφές με το ίδιο κλειδί). Ένα B^+-δέντρο έχει της εξής μορφή:



[3]

Για την αναζήτηση μιας εγγραφής (record) με τιμή-κλειδιού K σε ένα B^+-δέντρο, πραγματοποιείται επίσκεψη πολλών κόμβων, σε ένα μονοπάτι που ξεκινά πάντα από τη ρίζα και φτάνει σε κάποιο κόμβο-φύλλο. Σε κάθε κόμβο, η διαδικασία ανάκτησης (retrieval process) αναζητά το μικρότερο κλειδί K_i που είναι μεγαλύτερο του K και στη συνέχεια φορτώνει τον κόμβο που υποδεικνύεται από τον αντίστοιχο δείκτη P_i. Στην περίπτωση που δεν υπάρχει τέτοιο κλειδί, ακολουθείται ο τελευταίος δείκτης P_m όπου με m συμβολίζεται ο αριθμός των δεικτών στον κόμβο. Όταν η διαδικασία ανάκτησης φτάσει σε κόμβο-φύλλο, ελέγχεται το κατά πόσο ο κόμβος περιέχει το κλειδί K_i το οποίο ισούται με K. Εάν στον κόμβο περιέχεται όντως ένα τέτοιο κλειδί, τότε επιστρέφεται η εγγραφή στην οποία δείχνει ο δείκτης P_i. Αλλιώς η διαδικασία ανάκτησης αποτυγχάνει και επιστρέφει μήνυμα σφάλματος.

2.3 Παρουσίαση ψευδοκώδικα

Παρακάτω παρουσιάζεται η διαδικασία που περιγράφηκε και με ψευδοκώδικα.

```
Function: search (k)
        return tree_search (k, root);

Function: tree_search (k, node)
```

```
if node is a leaf then
        return node;
switch k do
        case k < k_0
                return tree_search(k, p_0);
        case k_i ≤ k < k_{i+1}
                return tree_search(k, p_i);
        case k_d ≤ k
                return tree_search(k, p_d);
```

[1]

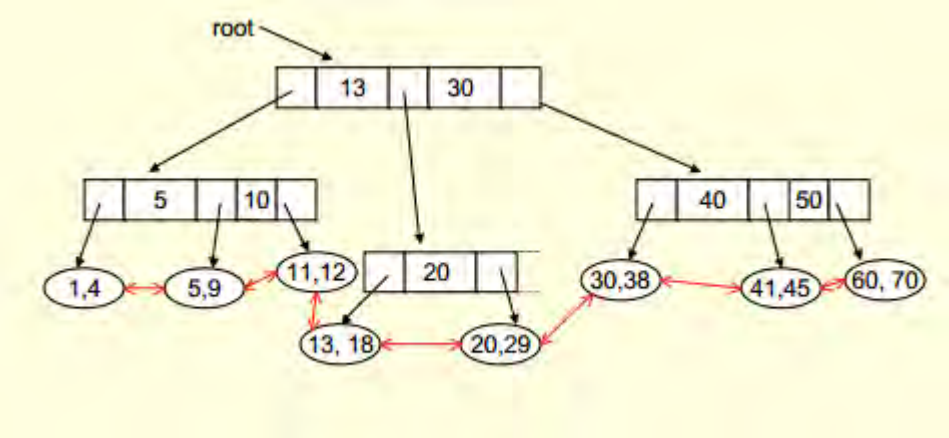
Για την εισαγωγή ενός κλειδιού K σε ένα B^+-δέντρο, ακολουθείται η εξής διαδικασία. Αρχικά εντοπίζεται ο κόμβος στον οποίο θα εισαχθεί η νέα καταγραφή. Εάν ο κόμβος δεν είναι γεμάτος (εξαρτάται από την τάξη του δέντρου), εισάγεται η νέα εγγραφή. Σε αντίθετη περίπτωση, διασπάται ο κόμβος. Δεσμεύεται ένας νέος κόμβος φύλλο και μεταφέρονται τα μισά στοιχεία του κόμβου στον νέο κόμβο. Εισάγεται έπειτα το μικρότερο κλειδί και η διεύθυνση του νέου κόμβου, στον γονέα του. Εάν ο κόμβος-γονέας είναι και αυτός με τη σειρά του γεμάτος, διασπάται και αυτός. Εισάγεται το μεσαίο κλειδί στον κόμβο-γονέα. Επαναλαμβάνεται η ίδια διαδικασία έως ότου βρεθεί γονέας που δε χρειάζεται να διασπαστεί. Τέλος, εάν η ρίζα διασπαστεί δημιουργείται νέα ρίζα η οποία έχει 1 κλειδί και 2 δείκτες σε κόμβους-παιδιά. Έτσι λοιπόν συμπεραίνουμε, ότι ένα B^+-δέντρο μεγαλώνει από τη ρίζα και όχι από τα φύλλα.

Τέλος, ας δούμε πώς επιτυγχάνεται η λειτουργία της διαγραφής μιας τιμής-κλειδιού K από ένα B^+-δέντρο. Ξεκινώντας από τον κόμβο-ρίζα, βρίσκουμε τον κόμβο-φύλλο L στον οποίο περιέχεται η εγγραφή που αναζητούμε να διαγράψουμε. Αφαιρείται στη συνέχεια η εγγραφή. Εάν ο L είναι τουλάχιστον μισογεμάτος, η διαδικασία της διαγραφής, έχει ολοκληρωθεί. Εάν όμως ο L έχει λιγότερες εγγραφές από ότι θα έπρεπε, καταβάλλεται προσπάθεια αναδιανομής εγγραφών στον κόμβο, δανείζοντας από τον κόμβο-αδελφό (γειτονικός κόμβος με τον οποίο μοιράζονται τον ίδιο γονέα). Εάν η αναδιανομή αποτύχει, συγχωνεύονται τότε ο L και ο κόμβος-αδελφός του. Στην περίπτωση της συγχώνευσης, πρέπει να διαγραφεί η εγγραφή (που έχει δείκτη στον L ή στον κόμβο-αδελφό του) από τον κόμβο-γονέα του L. Σαν αποτέλεσμα της συγχώνευσης, στην περίπτωση που επεκταθεί η διαδικασία μέχρι τη ρίζα, είναι η μείωση του ύψους του B^+-δέντρου.

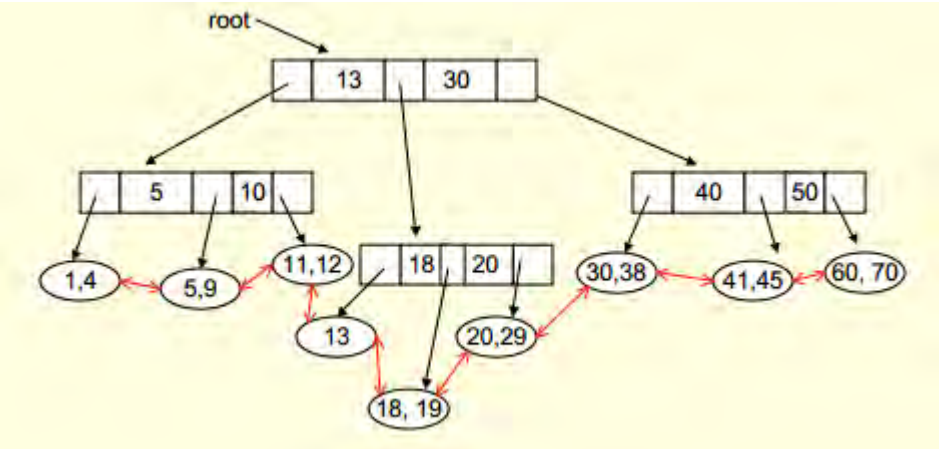
Να σημειωθεί σε αυτό το σημείο ότι εάν το B^+- δέντρο είναι τάξεως d, με n εγγραφές δεικτών, τότε το μεγαλύτερο κόστος αναζήτησης στην χειρότερη περίπτωση, είτε για εισαγωγή στοιχείου είτε για διαγραφή αυτού, είναι ανάλογο του $\log_{d/2}n$. Συνεπώς, όταν το μέγεθος του κόμβου αυξάνεται, το κόστος των παραπάνω πράξεων πέφτει.

2.4 Παραδείγματα Πολλαπλών λειτουργιών, βήμα προς βήμα:

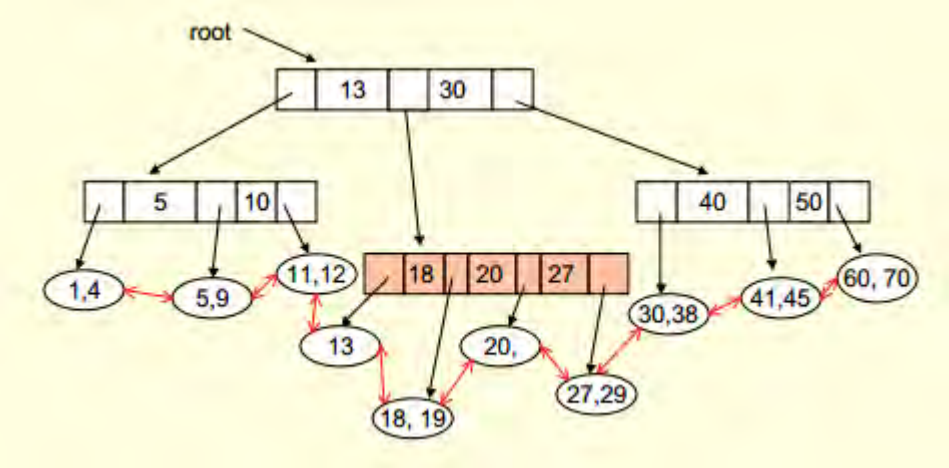
Παρακάτω, παρουσιάζουμε αναλυτικά τα βήματα που ακολουθούνται για τις λειτουργίες εισαγωγής και διαγραφής στοιχείων σε ένα B^+- δέντρο τάξεως 1.



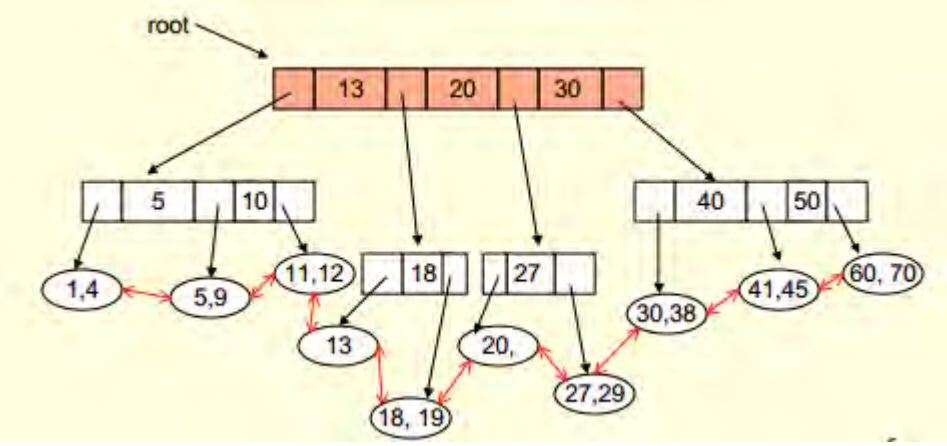
Εισαγωγή 19-Διαχωρισμός φύλλου-Επέκταση γονιού με το κλειδί 18.



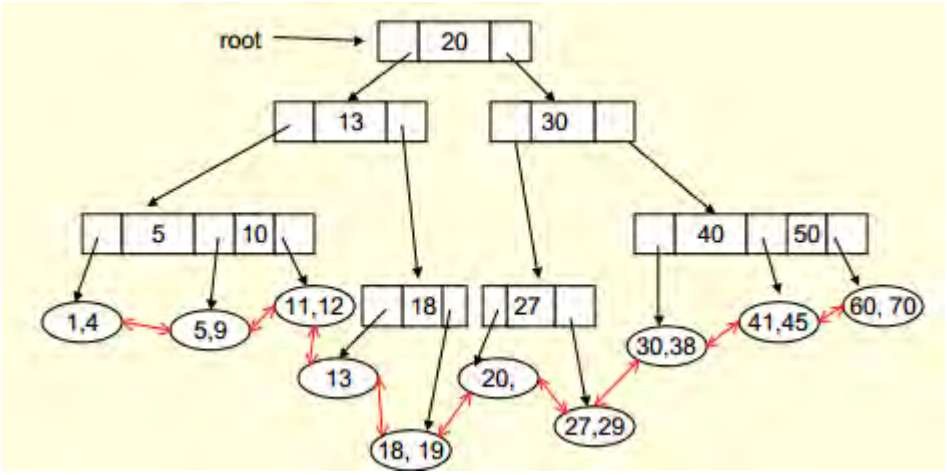
Εισαγωγή 27-Διαχωρισμός φύλλου-Επέκταση γονιού με το κλειδί 27-Διαίρεση του κόμβου.



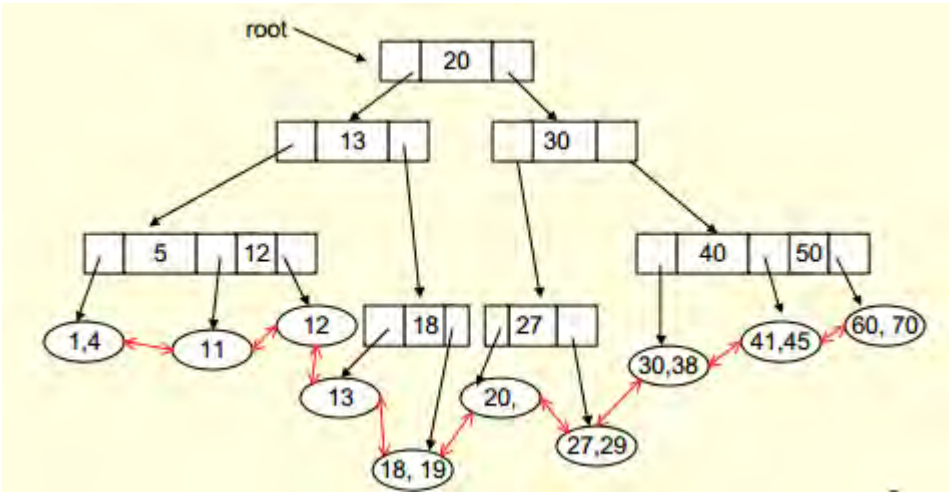
Εισαγωγή 27-Διαχωρισμός φύλλου-Διαχωρισμός γονιού-Επέκταση ''παππού'' με το κλειδί 20.



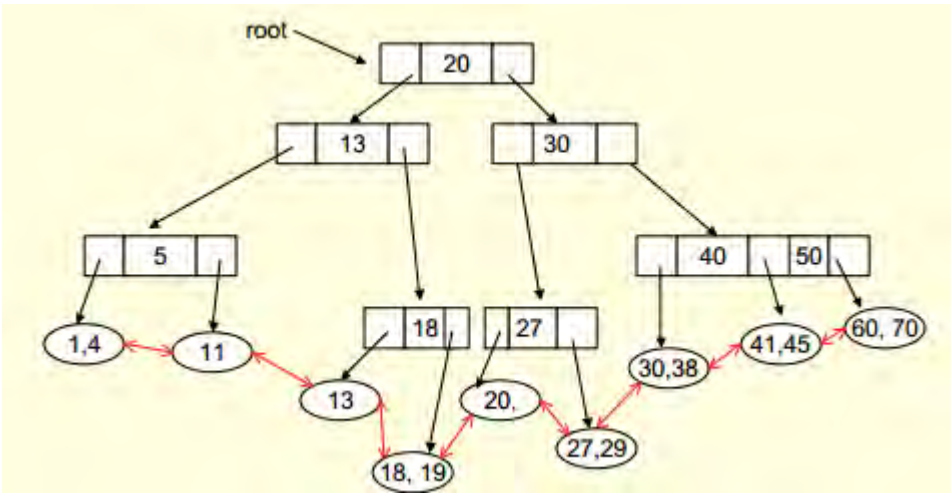
Εισαγωγή 27-Διαχωρισμός φύλλου-Διαχωρισμός γονιού-Διαχωρισμός ''παππού''-Νέα ρίζα με κλειδί 20.



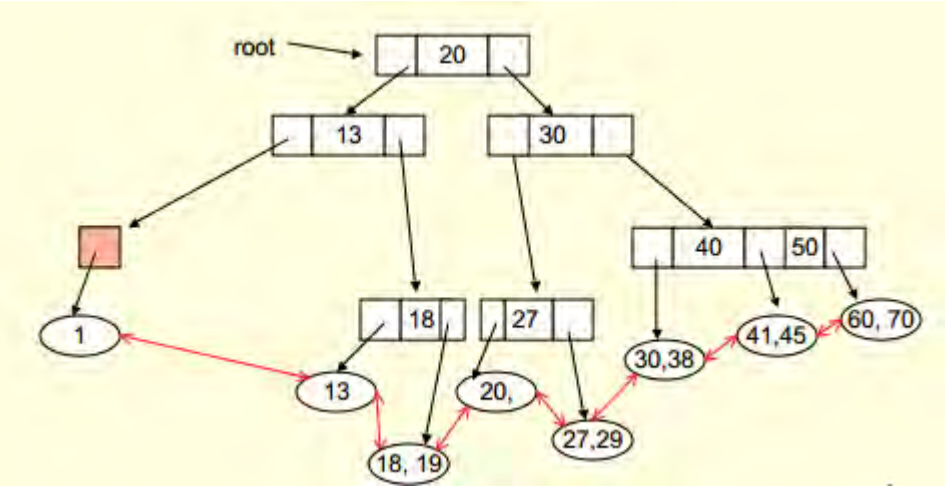
Διαγραφή κλειδιού 5-Διαγραφή κλειδιού 9-Αναδιανομή από τον δεξί ‘’αδελφό’’.



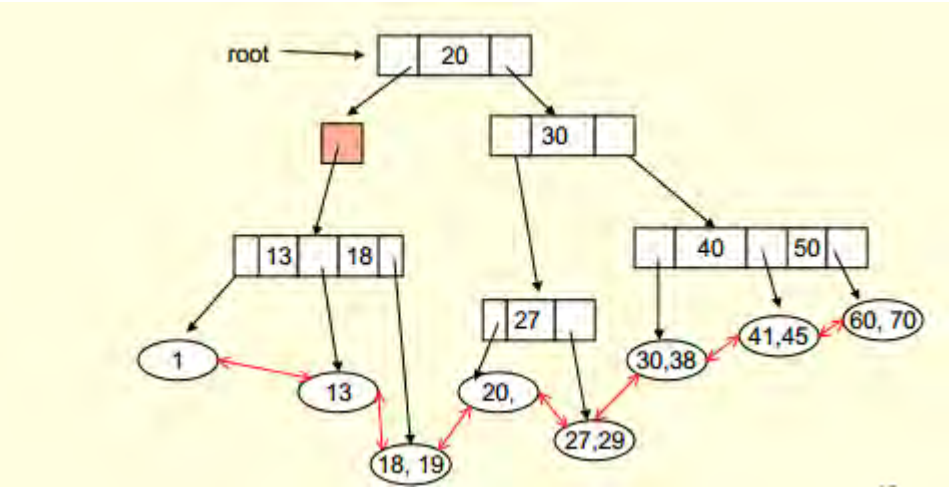
Διαγραφή του 12-Συγχώνευση φύλλων-Διαγραφή κλειδιού από τον γονέα.



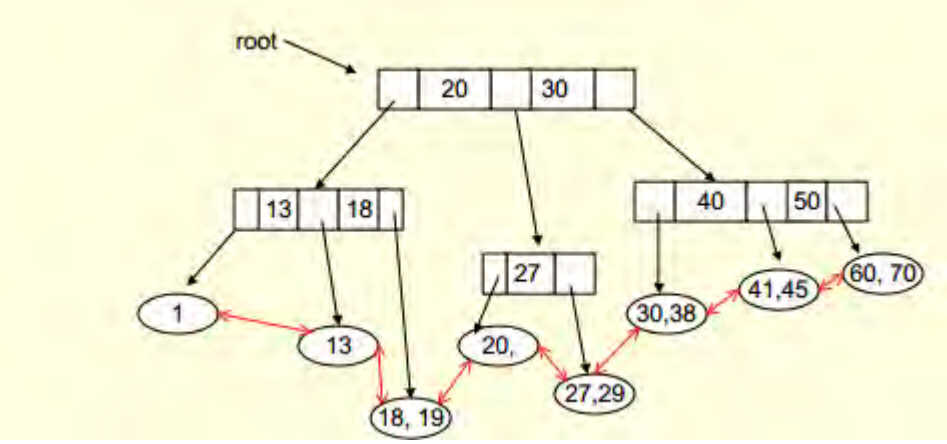
Διαγραφή του κλειδιού 4-Διαγραφή του κλειδιού 11-Συγχώνευση φύλλων-Διαγραφή του κλειδιού από τον γονέα-Γονέας όχι αρκετά γεμάτος.



Διαγραφή του κλειδιού 4-Διαγραφή του κλειδιού 11-Συγχώνευση φύλλων-Συγχώνευση γονέων-Κατεβαίνει κατά ένα, σε ύψος του δέντρου, το κλειδί 13-''Παππούς'' όχι αρκετά γεμάτος.



Διαγραφή του κλειδιού 4-Διαγραφή του κλειδιού 11-Συγχώνευση φύλλων-Συγχώνευση γονέων-Κατεβαίνει κατά ένα, σε ύψος του δέντρου, το κλειδί 13-Κατεβαίνει το κλειδί 20-Αφαίρεση της ρίζας.



[6]

2.5 Κατηγορίες B^+-δέντρων

‘Εχουν προταθεί διάφορα B^+-δέντρα κατάλληλα για χρήση σε μνήμη Flash. Το **B^+-δέντρο** (**Δίσκου**) (B^+-tree(Disk)) και το **B^+-δέντρο (Ημερολογίου)** (B^+-tree(Log)).

Στα **B^+-δέντρα (Δίσκου),** ένας κόμβος αποθηκεύεται ανάλογα με το μέγεθος του, σε πολλαπλές συνεχόμενες σελίδες της μνήμης Flash. Για να διαβαστεί επιτυχώς ένας κόμβος, πρέπει να διαβαστούν οι αντίστοιχες σελίδες. Αντίστοιχα, για την ανανέωση ενός κόμβου, διαβάζονται οι αντίστοιχες σελίδες στη μνήμη RAM, τροποποιούνται, και έπειτα ξαναγράφονται στη μνήμη Flash.

Ένα από τα βασικότερα προτερήματα των B^+-δέντρων (Δίσκου), είναι η φορητότητα του κώδικα (code portability). Οι υπάρχουσες υλοποιήσεις των B^+-δέντρων (Δίσκου) για σκληρούς δίσκους, μπορούν να εφαρμοστούν με επιτυχία και στη μνήμη Flash χωρίς αλλαγές. Το μειονέκτημα τους όμως από την άλλη, είναι ότι η διαδικασία της ανανέωσης κοστίζει ακριβά. Ακόμα και εάν ένα μικρό μέρος του B^+-δέντρου υποστεί αλλαγές (για παράδειγμα αλλάξει ένας δείκτης), τότε ολόκληρος ο κόμβος πρέπει να διαβαστεί στη RAM και να ξαναγραφτεί μετά, πάλι στη μνήμη Flash. Έτσι, τα B^+-δέντρα (Δίσκου) κρίνονται ακατάλληλα για εφαρμογές που έχουν ως επί το πλείστον εγγραφές (write operations).

Τα **B^+-δέντρα (Ημερολογίου),** εμπνευσμένα από τα συστήματα αρχείων με δομή ημερολογίου, καταφέρνουν να αποφύγουν το υψηλό κόστος ανανέωσης σε σύγκριση με εκείνων των B^+-δέντρων (Δίσκου). Η βασική ιδέα που οδήγησε στη δημιουργία των B^+-δέντρων (Ημερολογίου), είναι η οργάνωση της δομής ευρετηρίου σαν ένα ημερολόγιο. Μια λειτουργία εγγραφής σε έναν κόμβο του B^+-δέντρου, κωδικοποιείται σαν μια εγγραφή ημερολογίου και τοποθετείται σε έναν buffer της μνήμης RAM. Όταν ο buffer περιέχει αρκετά δεδομένα ώστε να γεμίσει μια σελίδα, τότε και μόνο τότε τα δεδομένα αυτά γράφονται στη μνήμη Flash.

Το βασικό πλεονέκτημα των B^+-δέντρων (Ημερολογίου), είναι το χαμηλό κόστος ανανέωσης, εφόσον το κόστος εγγραφής σελίδας εξοφλείται μέσω των πολλαπλών ανανεώσεων. Παρόλα αυτά, η λειτουργία ανάγνωσης ενός κόμβου του δέντρου κοστίζει ακριβά, αφού πρέπει να διαβαστούν πολλές εγγραφές ημερολογίου (και οι οποίες πιθανόν να είναι διασκορπισμένες σε πολλαπλές σελίδες) προκειμένου να κατασκευαστεί ο ζητούμενος κόμβος.

Ποιον όμως από τους δύο παραπάνω σχεδιασμούς των B^+-δέντρων πρέπει κάποιος να επιλέξει και για ποιόν λόγο?

Η επιλογή, εξαρτάται από το είδος της εφαρμογής που εκτελείται και από τα χαρακτηριστικά της συσκευής. Το B^+-δέντρο (Ημερολογίου), προτιμάται στις περιπτώσεις εφαρμογών, που στηρίζονται κατά βάση σε λειτουργίες εγγραφής (για παράδειγμα οι λειτουργίες αναζήτησης δεδομένων να είναι σπάνιες) και σε συσκευές όπου η λειτουργία της εγγραφής είναι σημαντικά ακριβότερη από εκείνη της ανάγνωσης (π.χ SD κάρτα). Σύμφωνα με πρόσφατες έρευνες, τα B^+-δέντρα (Ημερολογίου) αποτιμούνται ως 80% πιο αποτελεσματικά από τα B^+-δέντρα (Δίσκου), συνδυαζόμενα με κάρτα SD και εφαρμογές που απαιτούσαν μόνο εγγραφές. Παρόλα αυτά, αλλάζοντας είτε τις ιδιότητες της εφαρμογής είτε τις ιδιότητες της συσκευής, το B^+-δέντρο (Δίσκου) μπορεί να εκτελεστεί καλύτερα από το B^+-δέντρο (Ημερολογίου). [4]

Βιβλιογραφία

[1]http://el.wikipedia.org/wiki/%CE%94%CE%BF%CE%BC%CE%AE_%CE%B4%CE%B5%CE%B4%CE%BF%CE%BC%CE%AD%CE%BD%CF%89%CE%BD

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

[3]http://en.wikipedia.org/wiki/B%2B_tree

[4]http://faculty.cs.tamu.edu/ajiang/FlashDB.pdf

[5]http://www.dblab.upatras.gr/download/courses/db2/Slides/1_IndexingHashingI.pdf

[6]http://www.cs.princeton.edu/courses/archive/fall08/cos597A/Notes/BplusInsertDelete.pdf

[7]http://www.seanster.com/BplusTree/BplusTree.html

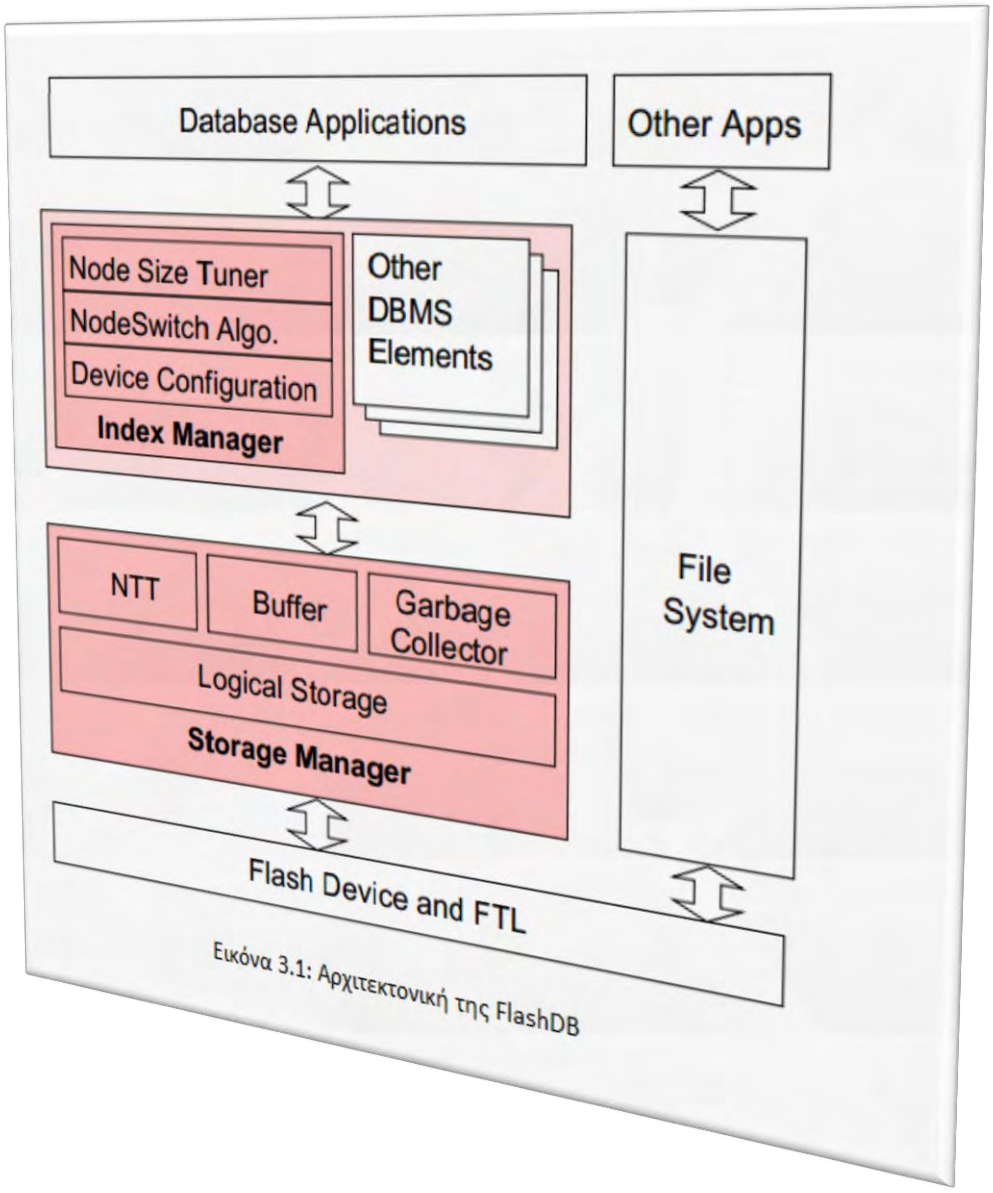
[8]http://dblab.cs.toronto.edu/courses/443/2013/05.btree-index.html

[9]http://www.mec.ac.in/resources/notes/notes/ds/bplus.htm

Κεφάλαιο 3

3.1 FlashDB - Βάση δεδομένων για δίκτυα αισθητήρων βασισμένα σε μνήμη Flash

Το FlashDB (Flash DataBase), είναι μια αυτορυθμιζόμενη (self-tuning) βάση δεδομένων κατάλληλη για μνήμη Flash. Αφού δηλαδή αρχικά διαμορφωθεί με τα κόστη εγγραφής και ανάγνωσης σελίδων της υπάρχουσας αποθηκευτικής συσκευής, τότε αυτόματα προσαρμόζει την αποθηκευτική δομή της (storage structure) με τέτοιον τρόπο ώστε να βελτιστοποιεί την κατανάλωση ενέργειας και την καθυστέρηση που σχετίζονται με το φόρτο εργασίας (workload) που της έχει ανατεθεί. **Η βάση δεδομένων FlashDB αποτελείται** από **2 κυρίως μέρη**: *ένα σύστημα διαχείρισης βάσεων δεδομένων* που εφαρμόζει τις λειτουργίες της βάσης δεδομένων, όπως για παράδειγμα διαχείριση ευρετηρίων (index management) και συλλογή επερωτήσεων και δεύτερο έναν *διαχειριστή αποθήκευσης* (storage manager) οποίος αναλαμβάνει αποτελεσματικά την προσωρινή αποθήκευση δεδομένων (data buffering) και τη συλλογή απορριμμάτων (garbage collection). Πιο αναλυτικά (Εικόνα 3.1):



Εικόνα 3.1: Αρχιτεκτονική της FlashDB

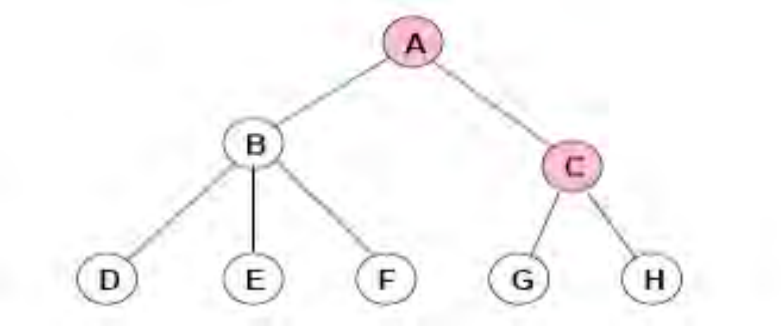
[2]

3.2 Σχεδίαση αυτορυθμιζόμενων B^+-δέντρων(ST) (B^+-trees (Self-Tuning))

Το καινοτόμο νέο χαρακτηριστικό των B^+-δέντρων(ST), είναι η δυνατότητα να αποθηκεύουν έναν κόμβο είτε σε κατάσταση ημερολογίου (Log Mode), είτε σε κατάσταση δίσκου (Disk Mode). Όταν ένας κόμβος είναι σε κατάσταση ημερολογίου, τότε κάθε λειτουργία ανανέωσης του συγκεκριμένου κόμβου (για παράδειγμα πρόσθεσης ή διαγραφής κλειδιού), είναι γραμμένη σαν μια ξεχωριστή εγγραφή ημερολογίου, όμοια με εκείνης του B^+-δέντρου

(Ημερολογίου). Επομένως, για να διαβαστεί ένας κόμβος που βρίσκεται σε κατάσταση ημερολογίου, πρέπει να διαβαστούν όλες οι εγγραφές ημερολογίου του κόμβου, οι οποίες ενδεχομένως να είναι διασκορπισμένες σε πολλές σελίδες. Όταν ο κόμβος είναι σε κατάσταση δίσκου, ολόκληρος ο κόμβος είναι γραμμένος σε συνεχόμενες σελίδες (των οποίων ο αριθμός εξαρτάται από το μέγεθος του κόμβου) και επομένως για να διαβαστεί αρκεί να διαβαστούν οι αντίστοιχοι τομείς.

Σε οποιαδήποτε χρονική στιγμή, κάποιοι λογικοί κόμβοι που απαρτίζουν το B^+-δέντρο(ST) μπορεί να είναι σε κατάσταση ημερολογίου ενώ κάποιοι άλλοι σε κατάσταση δίσκου. Επιπλέον, οι κόμβοι μπορούν να αλλάζουν δυναμικά τις καταστάσεις στις οποίες είναι γραμμένοι, καθώς μεταβάλλονται ο φόρτος εργασίας (workload) και η αποθηκευτική συσκευή. Στην Εικόνα 3.2, διακρίνεται ένα στιγμιότυπο ενός B^+-δέντρου(ST).



Εικόνα 3.2 [1]

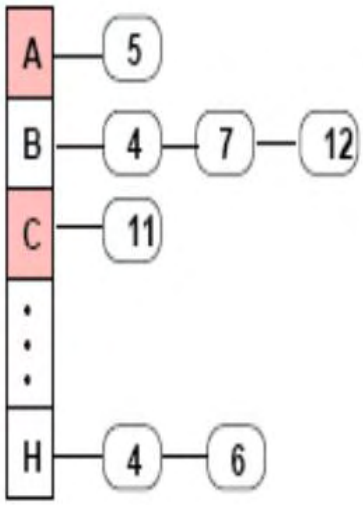
Οι χρωματισμένοι κόμβοι είναι σε κατάσταση δίσκου ενώ οι λευκοί είναι σε κατάσταση ημερολογίου.

3.3 Περιοχή Προσωρινής Αποθήκευσης Ημερολογίου(Log Buffer) – Πίνακας Μετάφρασης Κόμβων(NTT)

Το B^+-δέντρο(ST), εισάγει 2 επιπλέον κομμάτια στο διαχειριστή αποθήκευσης(Storage Manager) της βάσης δεδομένων FlashDB: *μια περιοχή προσωρινής αποθήκευσης ημερολογίου* (Log Buffer) και *έναν πίνακα μετάφρασης κόμβων*

(Node Translation Table - NTT). Η περιοχή προσωρινής αποθήκευσης ημερολογίου, η οποία μπορεί να αποθηκεύσει τόσα δεδομένα όσα είναι αποθηκευμένα σε ένα τομέα. Χρησιμοποιείται μόνο από τους κόμβους οι οποίοι εκείνη τη στιγμή είναι σε κατάσταση ημερολογίου. Όταν τροποποιείται ένας κόμβος, ο οποίος βρισκόταν σε κατάσταση ημερολογίου (Log Mode), τότε οι αντίστοιχες εγγραφές ημερολογίου κρατούνται προσωρινά στην περιοχή προσωρινής αποθήκευσης ημερολογίου. Όταν στην περιοχή αυτή, μαζευτούν τόσες εγγραφές σε ποσότητα, ίση με το μέγεθος μιας σελίδας, τότε αυτές οι εγγραφές γράφονται μαζί στη μνήμη Flash, αποφεύγοντας έτσι τις δαπανηρές μικρές εγγραφές.

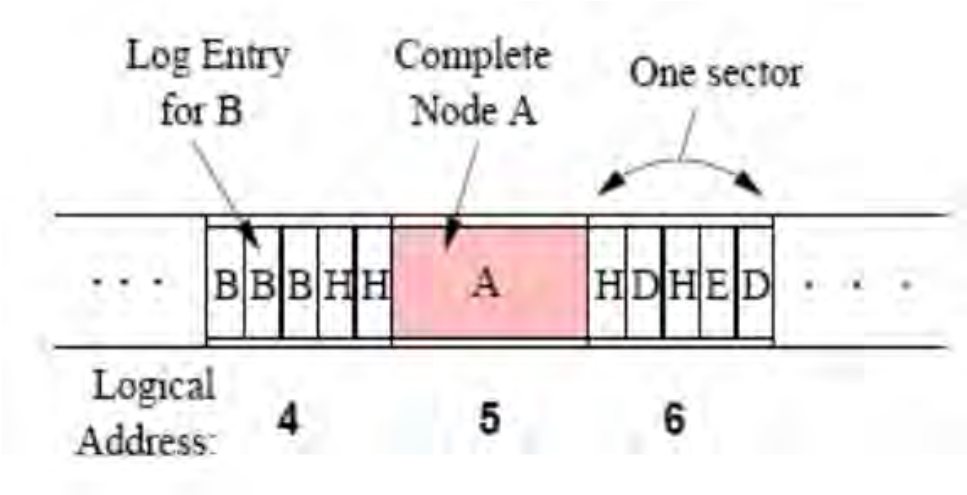
Ο πίνακας μετάφρασης κόμβων (NTT), αντιστοιχίζει τους λογικούς κόμβους του B^+-δέντρου(ST) στην τρέχουσα κατάστασή τους (ημερολογίου ή δίσκου) και στη φυσική τους αναπαράσταση. Στην Εικόνα 3.3, παρουσιάζεται ένα μέρος του πίνακα μετάφρασης κόμβων που αντιστοιχεί στο δέντρο της Εικόνας 3.2.



Εικόνα 3.3 [1]

Πιο αναλυτικά, για έναν κόμβο που βρίσκεται σε κατάσταση δίσκου (για παράδειγμα ο κόμβος Α), ο πίνακας μετάφρασης κόμβων καταγράφει τη διεύθυνση του τομέα της μνήμης Flash, στον οποίο είναι αποθηκευμένος ο κόμβος (εδώ για παράδειγμα, 5). Για έναν κόμβο που βρίσκεται σε κατάσταση ημερολογίου, ο πίνακας μετάφρασης κόμβων διατηρεί μια συνδεδεμένη λίστα όλων των διευθύνσεων των τομέων, οι οποίοι περιέχουν τουλάχιστον μία έγκυρη εγγραφή ημερολογίου για αυτόν τον κόμβο. Για παράδειγμα, στο παραπάνω σχήμα, ο κόμβος Β έχει τουλάχιστον μια

εγγραφή ημερολογίου σε κάθε έναν από τους 4,7 και 12 τομείς. Σε αυτό το σημείο, παρατηρείται ότι ένας τομέας που περιέχει εγγραφές για τον κόμβο B παραδείγματος χάρη, μπορεί να περιέχει εγγραφές και για άλλους κόμβους (λ.χ ο τομέας 4 στην Εικόνα 3.3.1) αφού η περιοχή προσωρινής αποθήκευσης ημερολογίου περιέχει εγγραφές διαφορετικών κόμβων όταν τα περιεχόμενα αυτά μεταφέρονται στη μνήμη Flash.



Εικόνα 3.3.1 [1]

Τομείς της μνήμης Flash. Ο τομέας 5, περιέχει ολόκληρο τον κόμβο A που είναι σε κατάσταση δίσκου, ενώ οι τομείς 4 και 6 περιέχουν εγγραφές ημερολογίου για διαφορετικούς κόμβους που βρίσκονται σε κατάσταση ημερολογίου.

3.4 Βασικές Λειτουργίες των B^+-δέντρων(ST)

Όπως είναι γνωστό μέχρι τώρα, λειτουργίες όπως αναζήτηση, διαγραφή, προσθήκη κλειδιού σε ένα B^+-δέντρο, μεταφράζονται σε δημιουργία, ανάγνωση και αναζήτηση των κόμβων του. Δεδομένου του πίνακα μετάφρασης κόμβων (NTT), αυτές οι λειτουργίες των κόμβων, αναπαρίστανται ως εξής: Για τη δημιουργία ενός κόμβου με το αναγνωριστικό (id) x, δημιουργείται μια εγγραφή με αναγνωριστικό x η οποία αποθηκεύεται στον πίνακα μετάφρασης κόμβων, σε κατάσταση ημερολογίου. Για την ανάγνωση ή την ανανέωση ενός κόμβου x, διαβάζεται σε πρώτη φάση η κατάσταση του κόμβου από τον πίνακα μετάφρασης κόμβων (NTT[x]). Εάν έπειτα, ο κόμβος x είναι σε κατάσταση δίσκου, τότε ο

κόμβος διαβάζεται από τον /ανανεώνεται στον τομέα που υποδεικνύει ο πίνακας μετάφρασης κόμβων. Εάν ο κόμβος x είναι σε κατάσταση ημερολογίου, τότε οι λειτουργίες είναι πιο περίπλοκες. Για την ανανέωση του κόμβου x, κατασκευάζεται μια εγγραφή ημερολογίου η οποία αντικατοπτρίζει τη λειτουργία της ανανέωσης και τοποθετείται στη συνέχεια στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer). Αργότερα, όταν η περιοχή προσωρινής αποθήκευσης ημερολογίου περιέχει εγγραφές σε ποσότητα ίση με το μέγεθος ενός τομέα, τότε αυτές οι εγγραφές γράφονται σε ένα διαθέσιμο τομέα της μνήμης Flash και η διεύθυνση του τομέα προστίθεται στην αρχή της συνδεδεμένης λίστας του στοιχείου NTT[x] του πίνακα μετάφρασης κόμβων. Για την ανάγνωση του κόμβου x, διαβάζεται η περιοχή προσωρινής αποθήκευσης ημερολογίου καθώς και όλοι οι τομείς της συνδεδεμένης λίστας του στοιχείου NTT[x], για την συλλογή και συντακτική ανάλυση (parse) των εγγραφών ημερολογίου του κόμβου x, πράξεις που οδηγούν με τη σειρά τους στην κατασκευή του λογικού κόμβου x.

3.5 Δομή Τομέα - NTT - Εγγραφών Ημερολογίου

Τα δεδομένα ενός B^+-δέντρου(ST), προτού γραφούν σε έναν τομέα της μνήμης Flash, περιβάλλονται από μια μικρή επικεφαλίδα η οποία περιέχει τα παρακάτω πεδία:

1. ***Checksum***- για τον έλεγχο των λαθών κατά τη διάρκεια της ανάγνωσης.
2. ***NodeMode***- για τον καθορισμό της κατάστασης των κόμβων (θα είναι δίσκου ή ημερολογίου)

Κάθε εγγραφή του πίνακα μετάφρασης κόμβων, περιέχει τα παρακάτω πεδία:

1. ***SectorList***- δείχνει στους τομείς οι οποίοι περιέχουν τον αντίστοιχο κόμβο ή στις εγγραφές ημερολογίου αυτού.
2. ***IsLeaf***- λαμβάνει την τιμή true (λογικό 1) στην περίπτωση που ο λογικός κόμβος είναι φύλλο.
3. ***LogVersion***- είναι η τελευταία έκδοση των εγγραφών ημερολογίου του κόμβου.

Κάθε εγγραφή ημερολογίου (Log Entry) ενός κόμβου που βρίσκεται σε κατάσταση ημερολογίου (Log Mode), περιέχει τα εξής πεδία:

1. ***NodeID***- το οποίο είναι το αναγνωριστικό του λογικού κόμβου με το οποίο ξεχωρίζονται οι εγγραφές ημερολογίου άλλων κόμβων.
2. ***Log Type***- περιγράφει τη λειτουργία που συντελείται στον λογικό κόμβο και η οποία μπορεί να είναι $3^{ων}$ ειδών: ADD_KEY, DELETE_KEY και UPDATE_POINTER. Οι πρώτοι 2 τύποι, χρησιμοποιούνται για την προσθήκη ή και διαγραφή αντίστοιχα, μιας εγγραφής (κλειδί, δείκτης) σε έναν κόμβο του B^{+}-δέντρου. Ο τελευταίος τύπος, είναι απαραίτητος για την ανανέωση του τελευταίου δείκτη, ενός εσωτερικού κόμβου (δηλαδή όχι φύλλο-internal node). Τέλος, είναι σχετικά εύκολη υπόθεση, η μετατροπή ενός κόμβου σε εγγραφές ημερολογίου ή ακόμα και η κατασκευή ενός, από εγγραφές ημερολογίου αυτών των $3^{ων}$ τύπων.
3. ***SequenceNumber***- αυξάνεται κατά μία μονάδα, κάθε φορά που παράγεται μία εγγραφή ημερολογίου για τον εκάστοτε λογικό κόμβο. Χρησιμοποιείται για τη σωστή χρήση των εγγραφών ημερολογίου, σύμφωνα με τη σειρά που αυτές παρήχθησαν.
4. ***LogVersion***- πρόκειται για την τελευταία και πιο πρόσφατη έκδοση των εγγραφών ημερολογίου.

Οι εγγραφές ημερολογίου, περιέχουν αρκετή πληροφορία, ώστε ακόμα και αν η εφαρμογή ‘’καταρρεύσει’’ (crashes) και χάσει τον πίνακα μετάφρασης κόμβων που βρίσκεται στην κύρια μνήμη (μνήμη RAM), ο πίνακας να μπορεί να αναδομηθεί σαρώνοντας ολόκληρη την μνήμη Flash. Μια τόσο ακριβή λειτουργία καλό θα ήταν να αποφεύγεται γεγονός που μπορεί να επιτευχθεί με περιοδικούς ελέγχους του πίνακα μετάφρασης κόμβων στη μνήμη Flash.

3.6 Βασικά Ζητήματα Αυτορύθμισης

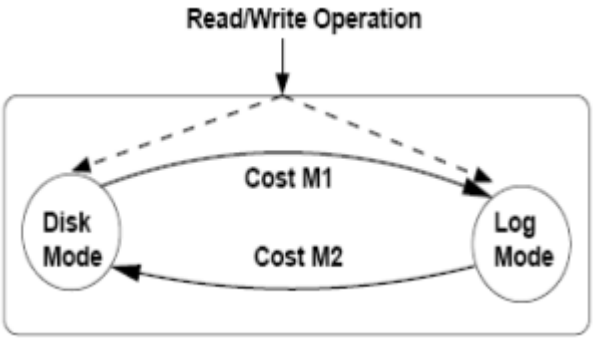
Για να γίνει ένα B^{+}-δέντρο(ST) αποδοτικό και αυτορυθμιζόμενο (self-tuning), πρέπει η κατάσταση ενός κόμβου, να αποφασίζεται και να ανανεώνεται με προσοχή. Επιπλέον, το μέγεθος ενός κόμβου ευρετηρίου, πρέπει να επιλέγεται με το βέλτιστο τρόπο.

3.6.1 Αλγόριθμος Εναλλαγής Κατάστασης

Ο βασικός μηχανισμός ενός B^+-δέντρου(ST), είναι ένας αλγόριθμος που αποφασίζει πότε ένας κόμβος πρέπει να αλλάξει την κατάσταση του (Mode) από κατάσταση δίσκου (Disk Mode) σε κατάσταση ημερολογίου (Log Mode) και ανάποδα. Η εναλλαγή μεταξύ των καταστάσεων επιφέρει επιπλέον κόστος. Πιο συγκεκριμένα για την εναλλαγή της κατάστασης ενός κόμβου x ***από κατάσταση ημερολογίου σε κατάσταση δίσκου***, διαβάζονται αρχικά οι εγγραφές ημερολογίου (log entries) του κόμβου και έπειτα ο κόμβος γράφεται σε κατάσταση δίσκου. Τώρα για την εναλλαγή της κατάστασης ενός κόμβου x ***από την κατάσταση δίσκου στην κατάσταση ημερολογίου,*** αρχικά ο κόμβος διαβάζεται σε κατάσταση δίσκου, έπειτα κωδικοποιείται σε μια ομάδα εγγραφών ημερολογίου που αντιπροσωπεύουν τον κόμβο και τέλος οι εγγραφές ημερολογίου τοποθετούνται στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer). Ανάλογα, τροποποιείται και το αντίστοιχο στοιχείο NTT[x] του πίνακα μετάφρασης κόμβων.

Στη συνέχεια, παρουσιάζεται ο αλγόριθμος εναλλαγής κατάστασης. Από τη στιγμή που ένας κόμβος, ο οποίος βρίσκεται σε κατάσταση δίσκου, είναι όπως έχει αναφερθεί παραπάνω, ο καταλληλότερος για λειτουργίες ανάγνωσης και ένας κόμβος που βρίσκεται σε κατάσταση ημερολογίου, καταλληλότερος για λειτουργίες εγγραφής, τότε είναι λογικό ποια κατάσταση πρέπει να προτιμάται και πότε. Όταν αναμένεται να υποστεί πολλές λειτουργίες εγγραφής (έστω περισσότερες από τις λειτουργίες ανάγνωσης), τότε ο κόμβος πρέπει να βρίσκεται σε κατάσταση ημερολογίου, ενώ όταν αναμένεται να υποστεί πολλές λειτουργίες ανάγνωσης, πρέπει να βρίσκεται σε κατάσταση δίσκου. Όμως, εφόσον η εναλλαγή καταστάσεων των κόμβων, επιφέρει επιπλέον κόστος, πρέπει να διασφαλιστεί ότι οι κόμβοι δεν αλλάζουν κατάσταση χωρίς λόγο. Επιπλέον, επειδή οι αποφάσεις εναλλαγής κατάστασης πρέπει να λαμβάνονται δυναμικά, απαιτείται ένας online αλγόριθμος που θα αποφασίζει για την εναλλαγή.

Το πρόβλημα της εναλλαγής καταστάσεων, μπορεί να αναχθεί σε ένα σύστημα καθηκόντων 2 καταστάσεων (Two-state Task System) το οποίο και παρουσιάζεται παρακάτω.



[1]

Εικόνα 3.6

Ένας κόμβος μπορεί να βρεθεί σε 2 καταστάσεις: είτε σε κατάσταση Δίσκου, είτε σε κατάσταση Ημερολογίου. Μια ανάγνωση R ή μια εγγραφή W, μπορεί να εξυπηρετηθεί από έναν κόμβο, ανεξαρτήτως καταστάσεως στην οποία βρίσκεται, αλλά τα κόστη διαφέρουν. Ο κόμβος, μπορεί να μεταβεί από την μια κατάσταση στην άλλη, με την προϋπόθεση όμως κάποιου κόστους. Στόχος λοιπόν, είναι να βρεθεί ένας online αλγόριθμος, βάσει του οποίου οι κόμβοι θα αλλάζουν δυναμικά την κατάσταση τους, ελαχιστοποιώντας το συνολικό κόστος εξυπηρέτησης των αιτήσεων και του κόστους εναλλαγής.[2]

Ο online αλγόριθμος εναλλαγής κατάστασης (SwitchMode Algorithm), παρουσιάζεται παρακάτω:

ALGORITHM **1.** *:* SWITCHMODE

(The following algorithm runs for each B-tree node n)

1. Initialize $S \leftarrow 0$ when migrate to the current mode.
2. For every read-write operation O
   - Suppose c_1 is the cost of serving O in current mode and c_2 is the cost of serving it in the other mode.
   - $S \leftarrow S + (c_1 - c_2)$
3. Suppose, M_1 and M_2 are the costs for transition between two modes. Then, switch to the other mode if $S \geq M_1 + M_2$

Έστω ότι με S συμβολίζουμε το μετρητή ανά κόμβο. Ο αλγόριθμος αυτός, είναι απλός και πρακτικός. Το μόνο που χρειάζεται, είναι να διαμορφωθεί με βάση τα κόστη ανάγνωσης και εγγραφής σελίδας, ενώ υλοποιείται με μόνο έναν μετρητή ανά κόμβο. Ο αλγόριθμος, λειτουργεί ανεξάρτητα για κάθε κόμβο του δέντρου και είναι 3-ανταγωνιστικός (3-competitive), αφού το κατώτερο όριο του ανταγωνιστικού λόγου είναι ίσο με 3. Ο ανταγωνιστικός λόγος ενός online αλγορίθμου, είναι η χειρότερη περίπτωση του λόγου (ratio), μεταξύ του κόστους που οφείλεται στον online αλγόριθμο προς το κόστος καλύτερης περίπτωσης (best-case cost) (που πιθανόν οφείλεται σε έναν offline βέλτιστο αλγόριθμο).

Ο μετρητής λοιπόν, που χρησιμοποιεί ο αλγόριθμος εναλλαγής κατάστασης για κάθε κόμβο του δέντρου, αντιπροσωπεύει τη συσσωρευμένη (accumulated) διαφορά στα κόστη των δύο καταστάσεων από την τελευταία εναλλαγή κατάστασης του κόμβου. Ο μετρητής του κόμβου x αποθηκεύεται στο στοιχείο NTT[x] του πίνακα μετάφρασης κόμβων. Υποθέτουμε ότι με c_r και c_w συμβολίζονται τα κόστη ανάγνωσης και εγγραφής ενός τομέα. Επίσης υποθέτουμε ότι ένας κόμβος του B^+-δέντρου(ST), που βρίσκεται σε κατάσταση δίσκου, καταλαμβάνει το πολύ k_s τομείς.

Έτσι, ο υπολογισμός του κόστους σε κατάσταση δίσκου είναι απλός: κάθε λειτουργία ανάγνωσης και εγγραφής στον κόμβο κοστίζει $k_s * c_r$ και $k_s * c_w$ αντίστοιχα.

Σε κατάσταση ημερολογίου, εάν μια λειτουργία εγγραφής παράξει l εγγραφές ημερολογίου και μια σελίδα της μνήμης Flash μπορεί να περιέχει το πολύ k_e εγγραφές ημερολογίου, τότε το κόστος της λειτουργίας εγγραφής θα είναι $l*c_w/k_e$. Παρόμοια, αν οι εγγραφές ημερολογίου ενός κόμβου διασκορπιστούν σε p σελίδες της μνήμης Flash, τότε το κόστος ανάγνωσης του κόμβου αυτού θα είναι: $p * c_r$. Εάν ο κόμβος είναι ήδη σε κατάσταση ημερολογίου, τότε οι τιμές των p και l μπορούν να προσδιοριστούν με ακρίβεια. Εάν όμως, ο κόμβος είναι ήδη σε κατάσταση δίσκου, τότε οι τιμές των p και l μπορούν απλά να εκτιμηθούν και όχι να προσδιοριστούν με ακρίβεια.

3.7 Βελτιστοποίηση B^+-δέντρων(ST)

3.7.1 Συμπίεση Ημερολογίου - Συλλογή Απορριμμάτων

Κατά τη δημιουργία μιας δομής ευρετηρίου, ένας κόμβος x ενός B^+-δέντρου(ST), μπορεί να ανανεωθεί πολλές φορές με αποτέλεσμα την διασκόρπιση ενός μεγάλου αριθμού εγγραφών ημερολογίου, σε ένα μεγάλο αριθμό τομέων στη μνήμη Flash. Το γεγονός αυτό έχει δύο μειονεκτήματα. Αρχικά, το πεδίο NTT[x].SectorList γίνεται πολύ μεγάλο και συνεπώς αυξάνεται και ο χώρος που καταλαμβάνεται από τον πίνακα μετάφρασης κόμβων NTT. Επιπλέον, η λειτουργία ανάγνωσης του κόμβου x κοστίζει ακριβότερα, αφού απαιτεί να διαβαστεί ένας μεγάλος αριθμός τομέων για αυτό το σκοπό. Για την αντιμετώπιση αυτών των προβλημάτων, χρησιμοποιούνται δύο τύποι **συμπίεσης ημερολογίου (log compaction)**.

Καταρχήν, διαβάζονται όλες οι εγγραφές ημερολογίου του κόμβου x στη μνήμη RAM και έπειτα ξαναγράφονται σε έναν μικρό αριθμό νέων τομέων [3]. Με αυτό επιτυγχάνεται βελτίωση της απόδοσης του συστήματος αφού οι εγγραφές ημερολογίου του κόμβου μπορούν να μοιράζονται κοινούς τομείς, με εγγραφές ημερολογίου άλλων κόμβων

και συνεπώς εξοικονομούνται έτσι τομείς της μνήμης Flash. Παρόλα αυτά όμως, η τεχνική αυτή δεν εγγυάται ένα άνω φράγμα του αριθμού των τομέων που απαιτούνται για έναν κόμβο, αφού ο αριθμός των εγγραφών ημερολογίου αυτού, μπορεί να μεγαλώσει ανεξέλεγκτα με το πέρασμα του χρόνου.

Προτείνεται λοιπόν ένας σημασιολογικός μηχανισμός συμπίεσης (***semantic compaction mechanism***), σύμφωνα με τον οποίο, εγγραφές ημερολογίου αντίθετης σημασιολογίας απορρίπτονται κατά τη διάρκεια της συμπίεσης. Για παράδειγμα, εάν το δεδομένο k εισαχθεί στον κόμβο x και έπειτα διαγραφεί από αυτόν (πιθανόν κατά τη λειτουργία διάσπασης ενός κόμβου αφότου ο κόμβος x θεωρηθεί ‘’γεμάτος’’), τότε ο κόμβος x θα έχει εγγραφές ημερολογίου ADD_KEY k και DELETE_KEY k. Οι δύο αυτές εγγραφές ημερολογίου ακυρώνουν η μία την άλλη και επομένως αποβάλλονται. Όμοια, πολλαπλές UPDATE_POINTER εγγραφές ημερολογίου του κόμβου x, μπορούν να αντικατασταθούν από την τελευταία εγγραφή ημερολογίου. Για τέτοιου είδους συμπίεση, λαμβάνεται υπόψη ο αριθμός ακολουθίας (***sequence number***) των εγγραφών ημερολογίου, ώστε να εφαρμόζονται με σειρά προτεραιότητας. Συμπερασματικά λοιπόν, εάν ένας κόμβος μπορεί να περιέχει το πολύ n δεδομένα, τότε θα έχει το πολύ n+1 εγγραφές ημερολογίου, φράσσοντας έτσι το μέγεθος της συνδεδεμένης λίστας του στοιχείου NTT[x] στο ***(n+1)/EntriesPerSector***. Απαιτείται επίσης, οι εγγραφές ημερολογίου να έχουν έναν αριθμό έκδοσης, ο οποίος θα αυξάνεται κατά ένα, μετά από κάθε σημασιολογική συμπίεση. Συνεπώς, μετά την διαδικασία της συμπίεσης, το στοιχείο NTT[x] ανανεώνεται, ώστε να περιλαμβάνει την τρέχουσα λίστα με τις διευθύνσεις των τομέων του κόμβου x.

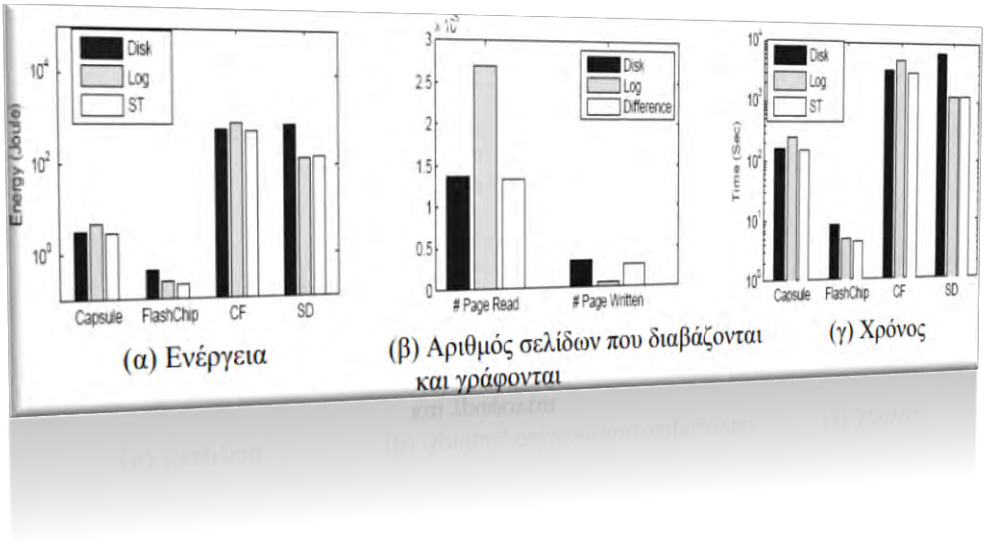
Ο σημασιολογικός μηχανισμός συμπίεσης, εισάγει την έννοια των ‘’πεπαλαιωμένων’’ εγγραφών (*stale*) ημερολογίου (δηλαδή έχουν τους παλαιότερους αριθμούς έκδοσης) τις οποίες και αποσύρει βάσει του μηχανισμού συλλογής απορριμμάτων ημερολογίου (***Log Garbage Collection - LGC***), για την ανάκτηση χώρου.

3.7.2 Μηχανισμός Συλλογής Απορριμμάτων Ημερολογίου (LGC) VS Μηχανισμός Συλλογής Απορριμμάτων του Διαχειριστή Αποθήκευσης (Garbage Collection in storage manager(GC))

Ο μηχανισμός συλλογής απορριμμάτων του διαχειριστή αποθήκευσης (GC), ανακτά κάποιο ποσοστό του χώρου από μη έγκυρες σελίδες (*dirty pages*), ενώ ο μηχανισμός συλλογής απορριμμάτων ημερολογίου (LGC), από μη έγκυρες εγγραφές ημερολογίου (*dirty log entries*). Ο LGC ενεργοποιείται, όταν η μνήμη Flash δεν έχει πια αρκετό διαθέσιμο ελεύθερο χώρο (για παράδειγμα όταν ο διαχειριστής αποθήκευσης αποτύχει να δεσμεύσει ένα νέο τομέα). Ξεκινά, σαρώνοντας ολόκληρη τη μνήμη Flash. Για κάθε τομέα, ο LGC αρχικά εξετάζει τις πληροφορίες που εμπεριέχονται στην επικεφαλίδα του τομέα, για να διαπιστώσει αν περιέχει εγγραφές ημερολογίου. Οι τομείς αυτοί, των οποίων οι επικεφαλίδες θα περιέχουν εγγραφές ημερολογίου, ονομάζονται ***τομείς ημερολογίου (Log Sectors).*** Σε κάθε τομέα ημερολογίου, ο μηχανισμός συλλογής απορριμμάτων μετράει τον αριθμό των πεπαλαιωμένων εγγραφών ημερολογίου. Εάν ο αριθμός αυτός, ξεπερνά ένα συγκεκριμένο όριο, τότε ο τομέας επιλέγεται για τη συλλογή απορριμμάτων. Έπειτα, ο μηχανισμός συλλογής απορριμμάτων γράφει τις νέες εγγραφές ημερολογίου στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer), αφαιρεί τη διεύθυνση του τομέα από τον πίνακα μετάφρασης κόμβων και επιστρέφει τον τομέα στο διαχειριστή αποθήκευσης. Τέλος, οι εγγραφές ημερολογίου, μεταφέρονται από τη περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer) στη μνήμη Flash και οι νέες διευθύνσεις προστίθενται στον πίνακα μετάφρασης κόμβων.

3.8 Εκτίμηση Απόδοσης της FlashDB

Σε αυτό το σημείο παρουσιάζονται κάποια στατιστικά στοιχεία, που προέκυψαν από την έρευνα ως προς την επίδοση ενός B^+-δέντρου(ST), εφαρμόζοντας το σε διαφορετικές συσκευές και με διάφορους φόρτους εργασίας (workloads).



Εικόνα 3.8 [1]

Όπως φαίνεται και στο σχήμα 9(α), παρατηρείται ότι **το B^+-δέντρο (Ημερολογίου)**, είναι **40% και 80% πιο αποδοτικό** από το **B^+-δέντρο (Δίσκου)**, όταν χρησιμοποιείται ως μνήμη Flash το *FlashChip της Samsung* και το *Secure Digital της Kingston* αντίστοιχα. Αντιθέτως, όταν ως μνήμη Flash χρησιμοποιείται το *Capsule της Toshiba* και το *Compact Flash της Sandisk*, τότε το B^+-δέντρο (Δίσκου) είναι 38% και 32% πιο αποδοτικό από το B^+-δέντρο (Ημερολογίου) αντίστοιχα. Το γεγονός αυτό, υποδεικνύει ότι σε καμία από τις 2 περιπτώσεις δεν επιτυγχάνεται ευελιξία σε συνδυασμό με αυτούς τους τύπους μνήμης Flash.

Η εξήγηση βρίσκεται στην Εικόνα 3.8(β), στην οποία φαίνεται ο αριθμός των σελίδων που διαβάστηκαν και εγγράφηκαν από ένα B^+-δέντρο (Δίσκου) και από ένα B^+-δέντρο (Ημερολογίου). Όπως διακρίνεται και στο σχήμα, οι εγγραφές στο B^+-δέντρο (Ημερολογίου) είναι 28K λιγότερες από εκείνες ενός B^+-δέντρου (Δίσκου) και οι αναγνώσεις στο B^+-δέντρο (Ημερολογίου) είναι 132K περισσότερες από εκείνες σε ένα B^+-δέντρο (Δίσκου).

Τέλος, το B^+-δέντρο(ST), αποδίδει καλύτερα ή έστω το ίδιο καλά με το καλύτερο εκ των B^+-δέντρο (Ημερολογίου) και B^+-δέντρο (Δίσκου), για όλα τα πακέτα μνήμης Flash. Αυτό οφείλεται, στο γεγονός ότι στο B^+-δέντρο(ST), κάθε κόμβος ξεχωριστά, παραμένει στη βέλτιστη κατάσταση ανάλογα με τον αντίστοιχο φόρτο εργασίας και τον τύπο μνήμης Flash που χρησιμοποιείται. Με τη χρήση των FlashChip και Secure Digital, οι περισσότεροι κόμβοι, παραμένουν σε κατάσταση ημερολογίου, επιτυγχάνοντας επίδοση, όμοια με εκείνης ενός B^+-δέντρου (Ημερολογίου). Για την ακρίβεια, το B^+-δέντρο(ST),

καταναλώνει λιγότερη ενέργεια από το B^+-δέντρο (Ημερολογίου), αφού κάποιοι από τους περισσότερο-αναγνώσιμους-κόμβους δίπλα στη ρίζα, παραμένουν σε κατάσταση δίσκου.

Βιβλιογραφία

[1] http://faculty.cs.tamu.edu/ajiang/FlashDB.pdf

[2] BORODIN, A., LINIAL, N., AND SAKS, M. An optimal online algorithm for metrical task systems. In ACM STOC (1987).

[3] WU, C.-H., CHANG, L.-P., AND KUO, T.-W. An efficient b-tree layer for flash-memory storage systems. In RTCSA (2003).

Κεφάλαιο 4

4.1 Εισαγωγή στο στρώμα BFTL

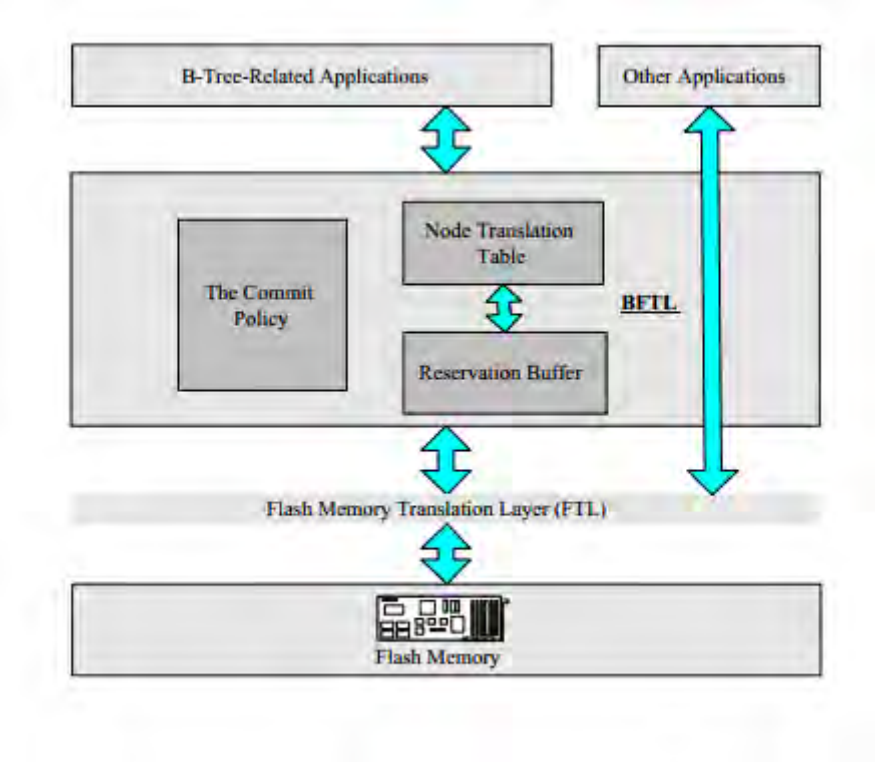
Καθώς η χωρητικότητα αποθήκευσης (storage capacity) της μνήμης Flash αυξάνεται συνεχώς, πολλά συστήματα (π.χ. κινητά τηλέφωνα), τη χρησιμοποιούν ως δευτερεύουσα αποθηκευτική συσκευή. Το BFTL (που ορίζεται ως ένα αποδοτικό B-δένδρο για Flash συστήματα αποθήκευσης - an efficient B-tree layer for Flash memory storage systems) ενοποιεί τη δομή ευρετηρίου B-δέντρο, με τους μηχανισμούς που παρέχει το FTL (Flash Translation Layer). Το BFTL, εισάγεται σαν στρώμα ανάμεσα στο σύστημα αρχείων (file system) και το FTL. Μπορεί να ενεργοποιηθεί ή όχι, ανάλογα με τις απαιτήσεις του εκάστοτε συστήματος. *Βασικός στόχος του στρώματος BFTL, είναι η κατάλληλη υλοποίηση των B-δένδρων, η οποία θα μειώσει τον φόρτο που προκαλείται από τον χειρισμό αυτών.* [1]

Η υλοποίηση του BFTL γίνεται διαφανώς, πάνω από το FTL, με αποτέλεσμα να μη χρειάζεται καμία τροποποίηση σε υπάρχουσες εφαρμογές που σχετίζονται με B-δέντρα**. Ο φόρτος των εντατικών λειτουργιών επιπέδου byte, προκαλείται από την εισαγωγή εγγραφών (record inserting), τη διαγραφή εγγραφών (record deleting) καθώς και από την αναδιοργάνωση των B-δέντρων.** Για παράδειγμα, η εισαγωγή μιας εγγραφής έχει ως αποτέλεσμα, την εισαγωγή ενός δείκτη δεδομένων (data pointer) σε ένα κόμβο-φύλλο (leaf node) και την πιθανή εισαγωγή δεικτών δένδρου (tree pointers) στο B-δέντρο. Ενέργειες σαν αυτές, προκαλούν την αντιγραφή πολλών δεδομένων (δηλ. αντιγραφή δεδομένων που δεν έχουν μεταβληθεί και των δεικτών προς τους σχετιζόμενους κόμβους) λόγω των ανανεώσεων εκτός θέσης (out-place update) στη μνήμη Flash. Αποδεικνύεται στην πορεία, ότι η προτεινόμενη μεθοδολογία, βελτιώνει σημαντικά την επίδοση του συστήματος μειώνοντας ταυτόχρονα το φόρτο διαχείρισης της μνήμης Flash και την κατανάλωση ενέργειας, όταν υιοθετούνται από τη μνήμη Flash δομές δεδομένων ευρετηρίου. Τέλος, το στρώμα BFTL μπορεί να προσαρμοστεί και στα συστήματα αρχείων ειδικά για μνήμη Flash.

4.2 Σχεδιασμός και υλοποίηση του στρώματος BFTL

Προτείνεται λοιπόν, το στρώμα B-δέντρου για Flash συστήματα αποθήκευσης (BFTL), το οποίο ως κύριο στόχο, έχει την ελάττωση των περιττών εγγραφών δεδομένων οι οποίες οφείλονται στους περιορισμούς της NAND μνήμης Flash. Συγκεκριμένα, παρουσιάζεται η αρχιτεκτονική ενός συστήματος που υιοθετεί το BFTL, καθώς και οι λειτουργίες των βασικών συστατικών του BFTL.

Το BFTL επικεντρώνεται κυρίως στην αποδοτική υλοποίηση δομών ευρετηρίου B-δένδρων πάνω από το FTL, παρέχοντας λειτουργίες επιπέδου συστήματος αρχείων για τη δημιουργία και διατήρηση αυτών. Στην εικόνα 4.2, απεικονίζεται η αρχιτεκτονική συστήματος που χρησιμοποιεί το BFTL.



Εικόνα 4.2 [1]

Πιο αναλυτικά, παρατηρείται ότι, το BFTL αποτελείται από μία *ειδική περιοχή προσωρινής αποθήκευσης* (reservation buffer) και από *έναν πίνακα μετάφρασης κόμβων* (node translation table). Όταν οι εφαρμογές εισάγουν, διαγράφουν ή τροποποιούν κάποια εγγραφή (record) τότε οι νέο-

δημιουργούμενες εγγραφές που προκύπτουν (μη έγκυρες/άκυρες εγγραφές - ‘’dirty records’’) θα αποθηκευτούν προσωρινά στην ειδική περιοχή προσωρινής αποθήκευσης του BFTL (reservation buffer). Έπειτα, από τη στιγμή που η ειδική περιοχή προσωρινής αποθήκευσης περιέχει έναν επαρκή αριθμό εγγραφών, οι μη έγκυρες εγγραφές, θα μεταφερθούν σιγά-σιγά στη μνήμη Flash. Αξίζει να αναφέρουμε σε αυτό το σημείο, ότι οι διαγραφές των εγγραφών, πραγματοποιούνται προσθέτοντας ‘’εγγραφές μη εγκυρότητας’’ (invalidation records) στην ειδική περιοχή προσωρινής αποθήκευσης.

Για τη μεταφορά των μη έγκυρων εγγραφών στην ειδική περιοχή προσωρινής αποθήκευσης, το BFTL κατασκευάζει μία μονάδα ευρετηρίου (index unit) για κάθε μία εξ’αυτών. Οι μονάδες ευρετηρίου, απεικονίζουν/αντανακλούν τις εισαγωγές και διαγραφές πρωτεύοντος κλειδιού στο B-δέντρο, οι οποίες προκαλούνται από τις μη έγκυρες εγγραφές. Η αποθήκευση των μονάδων ευρετηρίου καθώς και των μη έγκυρων εγγραφών πραγματοποιείται με 2 τρόπους. Πιο αναλυτικά, οι μη έγκυρες εγγραφές, γράφονται (ή ανανεώνονται) στις ανατεθείσες (ή στις αρχικές) θέσεις. Από την άλλη πλευρά, εξαιτίας του μικρού μεγέθους των μονάδων ευρετηρίου (σε σύγκριση με το μέγεθος μιας σελίδας), η αποθήκευση της πραγματοποιείται από μια πολιτική αποθήκευσης (commit policy). Πιο αναλυτικά, πολλές μονάδες ευρετηρίου, τοποθετούνται αποδοτικά σε λίγους τομείς (sectors), με σκοπό τη μείωση των εγγραφών σε επίπεδο σελίδων (page writes). Γίνεται μια προσπάθεια δηλαδή, μονάδες ευρετηρίου, οι οποίες ανήκουν σε κόμβους διαφορετικών B-δέντρων, να ομαδοποιηθούν σε μικρό αριθμό τομέων. Κατά τη διάρκεια της ομαδοποίησης αυτής, παρόλο που ο αριθμός των τομέων οι οποίοι έχουν υποστεί αλλαγές ελαττώνεται, υπάρχει περίπτωση οι μονάδες ευρετηρίου του κόμβου ενός B-δέντρου, να διασκορπίζονται σε διαφορετικούς τομείς. Εισάγεται λοιπόν σε αυτό το σημείο, ο *πίνακας μετάφρασης κόμβων* (Node Translation Table), προκειμένου το BFTL να μπορεί να αναγνωρίσει μονάδες ευρετηρίου που ανήκουν σε έναν κόμβο.

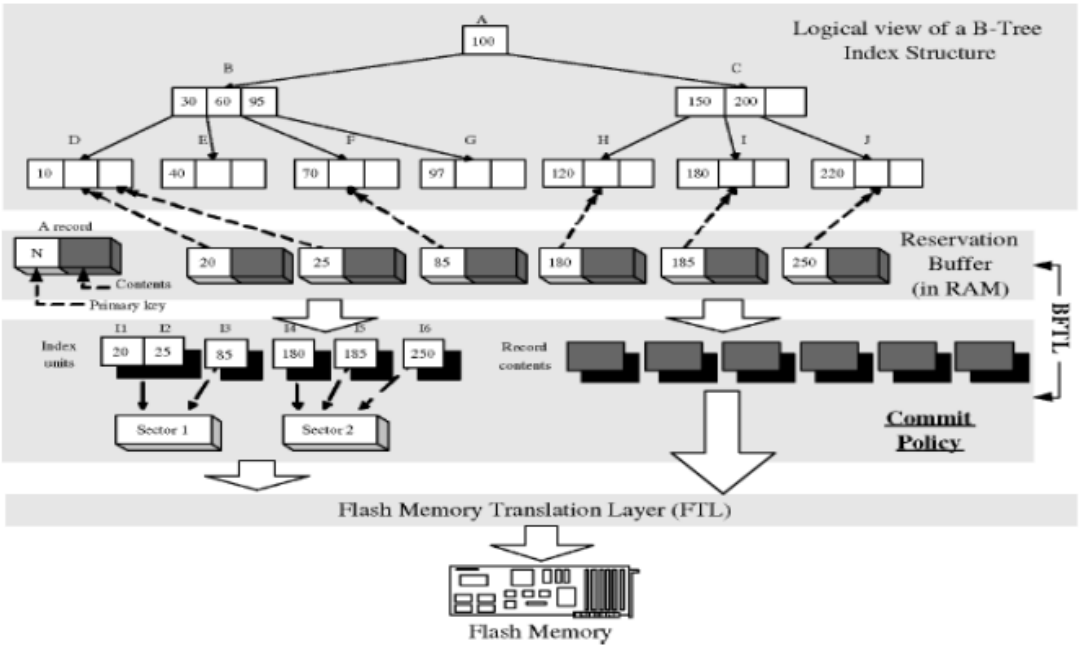
4.2.1 Φυσική αναπαράσταση του κόμβου ενός B-δέντρου: Εγγραφή Ευρετηρίου

Ένας κόμβος B-δέντρου αναπαρίσταται ως μια μονάδα ευρετηρίου και αποτελείται από επτά μέρη: ένα δείκτη προς τα δεδομένα (data_ptr), έναν κόμβο πατέρα (parent_node), ένα πρωτεύον κλειδί (primary_key), έναν αριστερό δείκτη (left_ptr), έναν δεξιό δείκτη (right_ptr), ένα αναγνωριστικό αριθμό (identifier number) και ένα δείκτη λειτουργίας (op_flag). Ο δείκτης προς τα δεδομένα, ο κόμβος πατέρας, ο αριστερός δείκτης, ο δεξής δείκτης και το πρωτεύον κλειδί αποτελούν συστατικά μέρη του αρχικού (original) κόμβου B-δέντρου. Αναπαριστούν, μια αναφορά προς το σώμα της εγγραφής (record body), έναν δείκτη προς τον κόμβο πατέρα, έναν δείκτη προς τον αριστερό κόμβο παιδί, έναν δείκτη προς τον δεξιό κόμβο παιδί καθώς και το πρωτεύον κλειδί αντίστοιχα. Εκτός όμως από τα συστατικά μέρη του αρχικού κόμβου B-δέντρου, είναι απαραίτητος και ένας αναγνωριστικός αριθμός (identifier number). Ο αναγνωριστικός αυτός αριθμός μιας μονάδας ευρετηρίου, υποδηλώνει τον κόμβο B-δέντρου στον οποίο ανήκει η μονάδα ευρετηρίου. Ο δείκτης λειτουργίας καθορίζει τη λειτουργία που επιτελείται από τη μονάδα ευρετηρίου. Κάποιες από τις λειτουργίες αυτές, μπορεί να είναι μια εισαγωγή (insertion), μια διαγραφή (deletion) ή ακόμα και μια ανανέωση (update). Επιπρόσθετα, προστίθενται χρονοσφραγίδες σε κάθε σύνολο μονάδων ευρετηρίου που προωθούνται στη μνήμη Flash, ώστε το στρώμα BFTL να μη κάνει χρήση παλαιών (stale) μονάδων ευρετηρίου. Αναφέρεται σε αυτό το σημείο, ότι το στρώμα BFTL χρησιμοποιεί το FTL για την αποθήκευση των μονάδων ευρετηρίου στη μνήμη Flash, οι οποίες μπορεί να είναι διασκορπισμένες στη μνήμη Flash. Η λογική πλευρά (logical view) ενός κόμβου B-δέντρου, κατασκευάζεται με τη βοήθεια του BFTL. Παρόλα αυτά, η σάρωση της μνήμης Flash, προκειμένου να επιτευχθεί η συλλογή των μονάδων ευρετηρίου των κόμβων του ίδιου B-δέντρου, καθίσταται αναποτελεσματική. Για αυτό το λόγο, το στρώμα BFTL, χρησιμοποιεί τον πίνακα μετάφρασης κόμβων (παρουσιάζεται παρακάτω) ώστε να διαχειριστεί τη συλλογή των μονάδων ευρετηρίου.

4.3 Πολιτική Αποθήκευσης

Το βασικό τεχνικό ζήτημα, είναι η αποδοτική τοποθέτηση πολλών μονάδων ευρετηρίου μέσα σε λίγους τομείς (sectors). Αναπτύχθηκε μια πολιτική αποθήκευσης (commit policy) για τις μονάδες ευρετηρίου η οποία ορίζει μια ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) ως εξής: η ειδική περιοχή προσωρινής αποθήκευσης είναι μια περιοχή εγγραφής δεδομένων, η οποία βρίσκεται στη μνήμη RAM. Όταν ένας κόμβος του B-δέντρου, εισάγεται, διαγράφεται ή τροποποιείται, τότε κάθε νέο-δημιουργούμενη εγγραφή (newly generated record) που προκύπτει, αποθηκεύεται πρώτα στην ειδική περιοχή προσωρινής αποθήκευσης. Οι εγγραφές της ειδικής περιοχής προσωρινής αποθήκευσης, αντιπροσωπεύουν λειτουργίες, οι οποίες δεν έχουν ακόμη εφαρμοστεί στο B-δέντρο. Για κάθε εγγραφή r της ειδικής περιοχής προσωρινής αποθήκευσης, υπάρχει ένας αντίστοιχος κόμβος του B-δέντρου, στον οποίο η εγγραφή r ανήκει. Η πολιτική αποθήκευσης, εστιάζει σε αυτή την σχέση μεταξύ των εγγραφών της ειδικής περιοχής προσωρινής αποθήκευσης και των κόμβων του B-δέντρου.

Η αρχειοθέτηση των μη έγκυρων εγγραφών (dirty records) στην ειδική περιοχή προσωρινής αποθήκευσης, προφυλάσσει τις δομές ευρετηρίου B-δέντρων της μνήμης Flash, από το να υπόκεινται συνεχώς σε τροποποιήσεις. Παρόλα αυτά, η χωρητικότητα της ειδικής περιοχής προσωρινής αποθήκευσης, δεν είναι απεριόριστη. Από τη στιγμή, που η ειδική περιοχή προσωρινής αποθήκευσης γεμίσει, κάποιες από τις μη έγκυρες εγγραφές μεταφέρονται (γράφονται) στη μνήμη Flash. Στο BFTL προτείνεται η μεταφορά στη μνήμη Flash, ολόκληρου του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης. Εκτός από την αποθήκευση των καταγραφών, το στρώμα BFTL κατασκευάζει μονάδες ευρετηρίου, για να δηλώσει τις τροποποιήσεις των δομών ευρετηρίου B-δέντρου. Από την άλλη πλευρά, γίνεται προσπάθεια ώστε να μην διασκορπίζονται οι μονάδες ευρετηρίου, του ίδιου κόμβου ενός B-δέντρου, σε πολλούς τομείς, ώστε να θεωρείται αποτελεσματική η συλλογή αυτών. Προτείνεται λοιπόν μια πολιτική αποθήκευσης ώστε να ικανοποιηθούν και οι 2 αυτές απαιτήσεις. Η πολιτική αποθήκευσης παρουσιάζεται πιο αναλυτικά με το παρακάτω παράδειγμα (Εικόνα 4.3):



Εικόνα 4.3 [1]

Ο χειρισμός μιας δομής ευρετηρίου B-δέντρου στο σχήμα, χωρίζεται σε 3 μέρη: Στην λογική πλευρά του B-δέντρου (logical view), στο στρώμα BFTL και στο FTL. Υποθέτοντας ότι η ειδική περιοχή προσωρινής αποθήκευσης έχει χώρο για 6 εγγραφές, των οποίων τα πρωτεύοντα κλειδιά είναι 20, 25, 85, 180, 185 και 250 αντίστοιχα. Όταν η ειδική περιοχή προσωρινής αποθήκευσης γεμίσει, οι εγγραφές πρέπει να γραφούν στη μνήμη Flash. Το στρώμα BFTL παράγει αρχικά, 6 μονάδες ευρετηρίου (I1-I6) για τις 6 εγγραφές. Με βάση τα πρωτεύοντα κλειδιά των εγγραφών και το εύρος τιμών των κόμβων φύλλου (D, E, F, G, H, I και J), οι μονάδες ευρετηρίου χωρίζονται σε πέντε ξένα σύνολα (disjoint sets) : {I1, I2} ∈ D, {I3} ∈ F, {I4} ∈ H, {I5} ∈ I, {I6} ∈ J. Ο παραπάνω διαχωρισμός, προφυλάσσει τις μονάδες ευρετηρίου του ίδιου κόμβου, από το να διασκορπιστούν μεταξύ τους.

Υποθέτουμε ότι οι τομείς (sectors), που παρέχονται από το FTL, αποθηκεύουν τρεις μονάδες ευρετηρίου ο καθένας. Επομένως, τα σύνολα {I1, I2} και {I3}, τοποθετούνται στον πρώτο τομέα και τα σύνολα {I4}, {I5} και {I6} στον δεύτερο τομέα, αφού ο πρώτος είναι γεμάτος. Τελικά, πραγματοποιούνται μόνο δύο εγγραφές τομέων. Αν όμως δεν χρησιμοποιούνταν η ειδική περιοχή προσωρινής αποθήκευσης και η πολιτική αποθήκευσης, τότε θα απαιτούνταν έξι εγγραφές τομέων, για το χειρισμό των τροποποιήσεων της δομής ευρετηρίου, αντί για δύο.

ΘΕΩΡΗΜΑ 1: *Το πρόβλημα ομαδοποίησης των μονάδων ευρετηρίου σε τομείς, είναι πολυπλοκότητας NP (NP-Hard).*

Απόδειξη: Το πρόβλημα της ομαδοποίησης των μονάδων ευρετηρίου σε τομείς, ανάγεται στο πρόβλημα της τοποθέτησης αντικειμένων σε κάδους (bin packing problem). Ορίζεται έστω B η χωρητικότητα του κάδου (bin) και K ο αριθμός των αντικειμένων (items), όπου κάθε αντικείμενο έχει ένα μέγεθος. Λύση του προβλήματος αυτού, είναι η τοποθέτηση των αντικειμένων μέσα στου κάδους με τέτοιο τρόπο, ώστε να ελαχιστοποιείται ο αριθμός των κάδων που θα χρησιμοποιηθούν.

Η αναγωγή, γίνεται ως εξής: έστω ότι η χωρητικότητα ενός τομέα (sector), είναι η χωρητικότητα ενός κάδου (B) και κάθε αντικείμενο (item) είναι ένα ξένο σύνολο μονάδων ευρετηρίου. Ο αριθμός των ξένων συνόλων είναι ίδιος με τον αριθμό των αντικειμένων, δηλαδή K. Το μέγεθος ενός ξένου συνόλου, είναι το μέγεθος του αντίστοιχου αντικειμένου. Εάν υπάρχει μια λύση για το πρόβλημα ομαδοποίησης των μονάδων ευρετηρίου, τότε η λύση αυτή θα εφαρμόζεται και στο πρόβλημα αυτό, της τοποθέτησης αντικειμένων σε κάδο.

Algorithm 1. The FIRST-FIT-based Commit Policy

```
1: Let Φ denote the set of the disjoint sets of index units
2: Let Θ denote the set of the sectors
3: while Φ is not empty do
4:     Let ds be a disjoint set in Φ
5:     if there exists a used sector sec in Θ that has available free space for ds then
6:         ds is stored in sector sec
7:     else
8:         create a new sector nsec to store ds
9:         Θ ← Θ + nsec
10:    end if
11:    Φ ← Φ - ds
12: end while
13: flush out Θ to flash memory
```

Υπάρχουν πολλοί προσεγγιστικοί αλγόριθμοι για το πρόβλημα της τοποθέτησης αντικειμένων σε κάδους (bin packing problem). Το στρώμα BFTL υιοθετεί τον παραπάνω αλγόριθμο (FIRST-FIT algorithm) προκειμένου να

περιορίσει το πλήθος των σελίδων που γράφονται, εξαιτίας της πολιτικής αποθήκευσης. Υποθέτουμε σε αυτή την περίπτωση, ότι ο κόμβος ενός B-δέντρου, ‘’χωράει’’ σε έναν τομέα, έτσι ώστε το μέγεθος ενός ξένου συνόλου να μην ξεπερνά αυτό του τομέα.

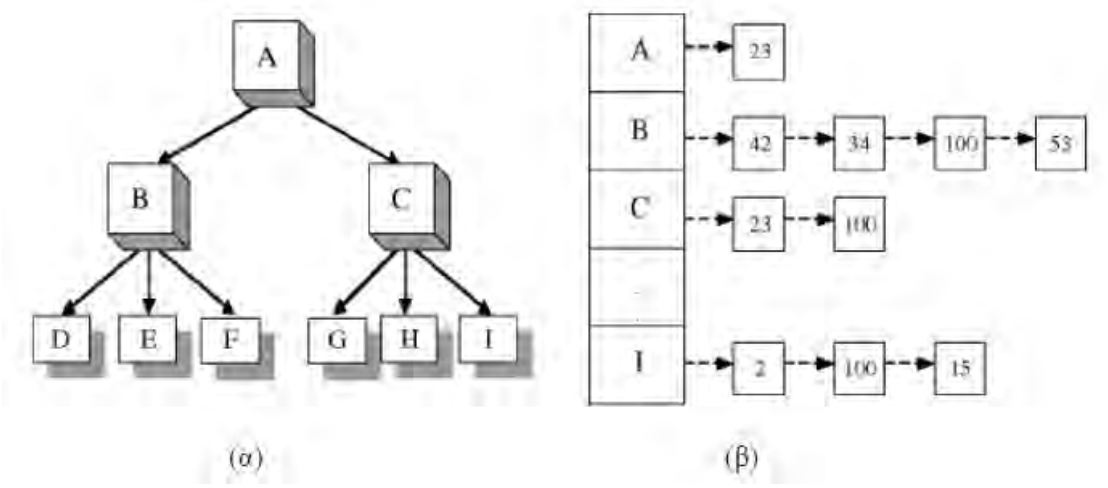
4.4 Ο Πίνακας Μετάφρασης Κόμβων (Node Translation Table)

Από τη στιγμή που η ακολουθούμενη πολιτική αποθήκευσης (commit policy), διασκορπίζει τις μονάδες ευρετηρίου ενός κόμβου του B-δέντρου στη μνήμη Flash, υιοθετείται ένας πίνακας μετάφρασης κόμβων, με τη βοήθεια του οποίου γίνεται η συλλογή των μονάδων ευρετηρίου ενός κόμβου. Αναλύεται παρακάτω, η σχεδίαση και ο τρόπος λειτουργίας του πίνακα μετάφρασης κόμβων.

Η κατασκευή της λογικής πλευράς (logical view) ενός κόμβου B-δέντρου, απαιτεί πρόσβαση σε όλες τις μονάδες ευρετηρίου του κόμβου και για το λόγο, η ανάκτηση των απαιτούμενων εξ’ αυτών μονάδων ευρετηρίου πρέπει να γίνεται με αποδοτικό τρόπο. Ένας πίνακας μετάφρασης κόμβων χρησιμοποιείται σαν επικουρική δομή δεδομένων, για την συλλογή των μονάδων ευρετηρίου. Αξίζει σε αυτό το σημείο να σημειωθεί, ότι ο πίνακας μετάφρασης κόμβων, έχει αρκετές ομοιότητες με τον πίνακα μετάφρασης λογικών διευθύνσεων (logical address translation table), ο οποίος αντιστοιχίζει τη λογική διεύθυνση του μπλοκ (LBA) σε έναν φυσικό αριθμό σελίδας (physical page number). Ο πίνακας μετάφρασης κόμβων, αντιστοιχίζει έναν κόμβο έναν κόμβο B-δέντρου σε μια συλλογή από λογικές διευθύνσεις μπλοκ (LBA’s), στις οποίες βρίσκονται αποθηκευμένες οι σχετιζόμενες με τον κόμβο μονάδες ευρετηρίου. Με άλλα λόγια, όλες οι LBA των μονάδων ευρετηρίου ενός κόμβου B-δέντρου, συνδέονται μετά τις αντίστοιχες εγγραφές του πίνακα μετάφρασης κόμβων. Προκειμένου να σχηματιστεί η λογική πλευρά ενός κόμβου B-δέντρου, το BFTL επισκέπτεται (διαβάζει) όλους τους τομείς στους οποίους βρίσκονται οι σχετιζόμενες με τον κόμβο μονάδες ευρετηρίου, και μετά κατασκευάζει μια ενημερωμένη λογική πλευρά (an up-to-date logical view) του κόμβου. **Ο πίνακας μετάφρασης κόμβων, ανακατασκευάζεται σαρώνοντας τη μνήμη Flash κάθε φορά που εκκινείται το σύστημα (powered-up).**

Στο σχήμα που ακολουθεί (Εικόνα 4.4(α)), παρουσιάζεται ένα B-δέντρο με εννέα κόμβους. Το αντίστοιχο σχήμα (Εικόνα 4.4(β)) αποτελεί ένα πιθανό

στιγμιότυπο του πίνακα μετάφρασης κόμβων, στο οποίο διακρίνεται κάθε κόμβος του B-δέντρου να αποτελείται από διάφορες μονάδες ευρετηρίου και οι οποίες αποθηκεύονται σε διάφορους τομείς.



Εικόνα 4.4

Οι LBA's των τομέων, ενώνονται μεταξύ τους με μορφή λίστας, αμέσως μετά την αντίστοιχη εγγραφή στον πίνακα. Όταν ένας κόμβος του B-δέντρου επισκεφθεί, όλες οι μονάδες ευρετηρίου που ανήκουν σε αυτό τον κόμβο, συλλέγονται από τη σάρωση εκείνων των τομέων, των οποίων τα LBA's είναι αποθηκευμένα στη λίστα. Για παράδειγμα, για την κατασκευή της λογικής πλευράς του κόμβου C (της εικόνας 4.4(α)) πρέπει να διαβαστούν οι λογικές διευθύνσεις μπλοκ 23 και 100 (LBA 23, LBA 100) από το BFTL ώστε να συλλεχθούν οι απαραίτητες μονάδες ευρετηρίου. Στην εικόνα 4.4(β), φαίνεται ότι η λογική διεύθυνση μπλοκ 100 (LBA 100), περιέχει μονάδες ευρετηρίου των κόμβων B, C και I του B-δέντρου της εικόνας 4.4(α). Επομένως, όταν πραγματοποιείται εγγραφή σε έναν τομέα, η λογική διεύθυνση μπλοκ του εγγεγραμμένου τομέα, προστίθενται στις ανάλογες εγγραφές (entries) του πίνακα μετάφρασης κόμβων.

Όμως υπάρχει το πρόβλημα της απροσδόκητης αύξησης των λιστών του πίνακα μετάφρασης κόμβων. Για παράδειγμα, εάν η λίστα μιας εγγραφής του πίνακα μετάφρασης κόμβων αποτελείται από 100 στοιχεία (αποθηκεύει δηλαδή 100 διευθύνσεις τομέων), τότε η επίσκεψη του αντίστοιχου κόμβου μπορεί να απαιτήσει έως και 100 αναγνώσεις τομέων. Εάν ο πίνακας μετάφρασης κόμβων μεγαλώνει με ανεξέλεγκτο τρόπο, μπορεί όχι απλά να

μειωθεί σε μεγάλο βαθμό η απόδοση του συστήματος, αλλά και να χρησιμοποιηθεί μεγάλο τμήμα της κύριας μνήμης.

Για την αντιμετώπιση του παραπάνω προβλήματος λοιπόν, προτείνεται η **συμπίεση (compaction) του πίνακα μετάφρασης κόμβων**, όποτε αυτό κρίνεται απαραίτητο. Προκειμένου να ελέγχεται το μέγιστο μήκος των λιστών του πίνακα μετάφρασης κόμβων, χρησιμοποιείται μια παράμετρος συστήματος C. Όταν το μήκος μιας λίστας μεγαλώσει, πέραν του C, τότε η λίστα, συμπιέζεται. Για τη συμπίεση μίας λίστας, όλες οι σχετικές μονάδες ευρετηρίου, μεταφέρονται στη μνήμη RAM και στη συνέχεια, γράφονται στη μνήμη Flash σε όσο το δυνατόν μικρότερο αριθμό τομέων. Ως αποτέλεσμα, το μέγεθος του πίνακα μετάφρασης κόμβων, έχει ως ασυμπτωτικό άνω όριο το O(N*C), με το N να συμβολίζει το πλήθος των κόμβων του B-δέντρου. Από την άλλη πλευρά, ο αριθμός αναγνώσεων, που απαιτούνται κατά την επίσκεψη ενός κόμβου, σε επίπεδο τομέων, φράσσεται από την παράμετρο C. Παρακάτω, παρουσιάζεται ψευδοκώδικας ενός αναθεωρημένου αλγορίθμου της πολιτικής αποθήκευσης (commit policy), ο οποίος διαχειρίζεται τις λειτουργίες του πίνακα μετάφρασης κόμβων.

Algorithm 2. A Revised Commit Policy with the Considerations of the Node-Translation Table

```
1:  Let Φ denote the set of the disjoint sets of index units
2:  Let Θ denote the set of the sectors
3:  Let ntt denote a node-translation table
4:  while Φ is not empty do
5:     Let ds be a disjoint set in Φ
6:     Let en be the corresponding entry of ntt of a B-tree node that ds would update
7:     if the length of the list of the corresponding entry en is beyond C then
8:        execute the compaction of the list
9:     end if
10:    if there exists a used sector sec in Θ that has available free space for ds then
11:       ds is stored in sector sec
12:       record the LBA of sec in the list after the corresponding entry en of ntt
13:    else
14:       create a new sector nsec to store ds
15:       record the LBA of nsec in the list after the corresponding entry en of ntt
16:       Θ ← Θ + nsec
17:    end if
18:    Φ ← Φ - ds
19: end while
20: flush out Θ to flash memory
```

4.5 Ανάλυση Πολυπλοκότητας BFTL

Προκειμένου να αναλυθεί η συμπεριφορά του BFTL και του FTL, θα υπολογίσουμε το πλήθος των τομέων που διαβάζονται και γράφονται από το BFTL και FTL αντίστοιχα, στην περίπτωση εισαγωγής n εγγραφών (records).

Έχοντας ήδη μια δομή ευρετηρίου B-δέντρου που βρίσκεται στη μνήμη Flash και χωρίς να αποκλείσουμε τη γενίκευση της κατάστασης, έστω ότι ένας κόμβος του B-δέντρου ‘’χωράει’’ σε έναν τομέα (που παρέχεται από το FTL). Υποθέτουμε ότι θέλουμε να εισάγουμε n καταγραφές (records). Αυτό σημαίνει ότι n πρωτεύοντα κλειδιά, θα εισαχθούν στην δομή ευρετηρίου. Οι τιμές όλων των πρωτευόντων κλειδιών (primary keys) των n εγγραφών, είναι διαφορετικές μεταξύ τους.

Αρχικά, ερευνώντας τη συμπεριφορά του FTL, ένας κόμβος B-δέντρου, αποθηκεύεται σε έναν ακριβώς τομέα στο FTL. Απαιτείται μια εγγραφή τομέα για κάθε εισαγωγή πρωτεύοντος κλειδιού, με την προϋπόθεση βέβαια ότι δεν παρουσιάζεται υπερχείλιση κόμβου (node overflow) (διάσπαση κόμβων).

Στην περίπτωση υπερχείλισης κόμβου, το πρωτεύον κλειδί του κόμβου, μεταφέρεται στον κόμβο πατέρα και ο κόμβος διασπάται (node splitting) σε δύο νέους κόμβους. Από την άλλη πλευρά, εάν ένας κόμβος δεν είναι γεμάτος κατά το ήμισυ, τότε αυτός ο κόμβος μπορεί να συγχωνευτεί με άλλους μισογεμάτους κόμβους αδέλφια (sibling nodes) ή να εναλλάξει (rotate) ένα πρωτεύον κλειδί με έναν από τους κόμβους αδέλφια. Η διάσπαση, απαιτεί τρείς εγγραφές τομέων υπό το FTL (οι δύο πρώτες εγγραφές τομέων χρειάζονται για τους δύο νέους κόμβους και η τρίτη για τον κόμβο πατέρα). Επιπλέον, η συγχώνευση (merging) και η εναλλαγή (rotating), απαιτούν, το πολύ, τρείς εγγραφές τομέων έκαστη, υπό το FTL. Έστω **H** το τωρινό ύψος του B-δέντρου, με $\mathbf{N_{split}}$ τον αριθμό των κόμβων που έχουν διασπαστεί και με $\mathbf{N_{merge/rotate}}$ τον αριθμό των κόμβων που συγχωνεύτηκαν/υπέστησαν εναλλαγή κατά τη διάρκεια εισαγωγής εγγραφών. Ο αριθμός των τομέων, που διαβάζονται και γράφονται από το FTL για το χειρισμό των εισαγωγών, μπορεί να αναπαρασταθεί, ως εξής:

$$\begin{cases} R_{FTL} = \mathrm{O}(n * H) \\ W_{FTL} = \mathrm{O}(n + 3 * N_{split} + 3 * N_{merge/rotate}) \end{cases}$$

Σχέση (1) [1]

Ας υποθέσουμε τώρα, ότι το μέγεθος του τομέα, παραμένει ίδιο υπό το BFTL (να σημειωθεί ότι το στρώμα BFTL βρίσκεται πάνω από το FTL) και ότι το ύψος του B-δέντρου είναι H. Θέλουμε να υπολογίσουμε τους αριθμούς των τομέων που θα εγγραφούν ή θα διαβαστούν στη/από τη μνήμη Flash, όταν εισαχθούν n καταγραφές. Επειδή το BFTL υιοθετεί τον πίνακα μετάφρασης κόμβων για τη συλλογή των μονάδων ευρετηρίου (index units) ενός κόμβου, το πλήθος των τομέων που διαβάζονται για την κατασκευή ενός κόμβου B-δέντρου, εξαρτάται από το μέγεθος των λιστών του πίνακα μετάφρασης κόμβων. Έστω ότι το μέγεθος της λίστας έχει ως άνω όριο το C (που αναφέραμε στην ανάλυση του πίνακα μετάφρασης κόμβων). Τότε ο αριθμός των τομέων που διαβάζονται από το BFTL για το χειρισμό των εισαγωγών, μπορεί να αναπαρασταθεί ως εξής:

$$R_{BFTL} = \mathrm{O}(n * H * C)$$

Σχέση (2) [1]

Η Σχέση (2), δείχνει ότι το BFTL διαβάζει περισσότερους τομείς από το FTL για το χειρισμό των εισαγωγών. Στην πραγματικότητα, το BFTL ανταλλάσσει αποδοτικά εγγραφές με επιπλέον αναγνώσεις. Ο αριθμός των τομέων που εγγράφονται από το BFTL, υπολογίζεται ως εξής: Επειδή το BFTL υιοθετεί την ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) με σκοπό να διατηρήσει τις εγγραφές στη RAM, όλες τις τροποποιήσεις των κόμβων του B-δέντρου (δηλαδή, οι μονάδες ευρετηρίου) τοποθετούνται (packed) σε λίγους μόνο τομείς. Έστω ότι η χωρητικότητα της ειδικής περιοχής προσωρινής αποθήκευσης του BFTL, είναι b εγγραφές. Ως αποτέλεσμα, το περιεχόμενο της ειδικής περιοχής προσωρινής αποθήκευσης, μεταφέρεται στη μνήμη Flash, βάσει της πολιτικής αποθήκευσης (commit policy), τουλάχιστον $\lceil n/b \rceil$ φορές κατά τη διάρκεια του χειρισμού των n εισαγωγών. Έστω N^i_{split} και $N^i_{merge/rotate}$ οι αριθμοί των κόμβων που διασπώνται και συγχωνεύονται/εναλλάσσονται αντίστοιχα, κατά το χειρισμό της i-οστής

μεταφοράς του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη Flash. Είναι φανερό, ότι $\sum_{i=1}^{n/b\rceil} N^i_{split} = N_{split}$ και $\sum_{i=1}^{n/b\rceil} N^i_{merge/rotate} = N_{merge/rotate}$ μιας και τα B-δέντρα υπό το BFTL και το FTL είναι λογικά πανομοιότυπα (logically identical).

Σε κάθε ξεχωριστό βήμα της μεταφοράς του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη Flash, μεταφέρονται *(b + N^i_{split} * fanout + $N^i_{merge/rotate}$ * fanout)* μη έγκυρες μονάδες ευρετηρίου (dirty index units). Υπενθυμίζουμε εδώ, ότι το fanout είναι ο μέγιστος αριθμός παιδιών ανά κόμβο. Ο όρος N^i_{split} * *fanout* υπάρχει στον παραπάνω τύπο, επειδή κάθε διάσπαση κόμβου έχει ως αποτέλεσμα την ανανέωση του κόμβου πατέρα και δύο νέων κόμβων όπου χρειάζονται σε πλήθος, fanout μονάδες ευρετηρίου.

Ο όρος $N^i_{merge/rotate}$ * *fanout* υποδηλώνει ότι κάθε συγχώνευση/εναλλαγή απαιτεί το πολύ fanout σε πλήθος, μονάδες σβησίματος (δηλ. κάθε συγχώνευση έχει ως αποτέλεσμα τη δημιουργία ενός νέου κόμβου και την ανανέωση του κόμβου πατέρα στην οποία απαιτούνται fanout σε πλήθος μονάδες ευρετηρίου).

Όπως και στο FTL, έστω ότι ένας κόμβος B-δέντρου ‘’χωρά’’ σε έναν τομέα. Αυτό συνεπάγεται, ότι ένας τομέας αποθηκεύει (fanout-1) μονάδες ευρετηρίου. Έστω Λ = (fanout – 1). Ο αριθμός των τομέων που εγγράφονται κατά την i-οστή μεταφορά του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη Flash, είναι περίπου ίσος με *($\frac{b}{\Lambda}$ + N^i_{split} + $N^i_{merge/rotate}$)*. Για την ολοκληρωτική μεταφορά του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη Flash, πρέπει να γραφούν τουλάχιστον $\sum_{i=1}^{n/b\rceil}$ *(* $\frac{b}{\Lambda}$ + N^i_{split} + $N^i_{merge/rotate}$ *) = (* $\sum_{i=1}^{n/b\rceil} \frac{b}{\Lambda}$ *) +* N_{split} + $N_{merge/rotate}$ τομείς.

Από τη στιγμή, που το BFTL υιοθετεί τον αλγόριθμο ‘’πρώτου ταιριάσματος’’ (FIRST-FIT algorithm), ο αριθμός των τομέων που γράφονται από το BFTL δίνεται από τον εξής τύπο:

$$W_{BFTL} = O\left(2 * \left(\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda} + N_{split} + N_{merge/rotate}\right)\right) = O\left(\frac{2*n}{\Lambda} + 2*N_{split} + 2*N_{merge/rotate}\right)$$

Σχέση (3) [1]

Συγκρίνοντας τις Σχέσεις (1) και (3), παρατηρείται ότι το W_{BFTL} είναι πολύ μικρότερο από το W_{FTL} , από τη στιγμή που το Λ (ο αριθμός των μονάδων ευρετηρίου που αποθηκεύονται σε έναν τομέα) είναι μεγαλύτερο από δύο. Παρόλα αυτά, η συμπίεση (compaction) του πίνακα μετάφρασης κόμβων, εισάγει επιπλέον φόρτο κατά το χρόνο εκτέλεσης. Ακολουθεί η απόδειξη, ότι όταν το Λ ισούται με είκοσι, ο αριθμός των τομέων που γράφονται από το FTL.

4.5.1. Εκτίμηση Απόδοσης BFTL

Στη συνέχεια, εκτιμάται η απόδοση του BFTL, με σκοπό να παρουσιαστούν τα οφέλη από την εφαρμογή του. Περιορίζοντας τις περιττές εγγραφές δεδομένων στη μνήμη Flash, οι επιδόσεις των λειτουργιών/πράξεων πάνω στα B-δέντρα βελτιώνονται αισθητά.

Ένα σύστημα βασισμένο σε NAND μνήμη Flash, εφαρμόστηκε για την αναγνώριση της απόδοσης του BFTL. Το μέγεθος της NAND μνήμης Flash που χρησιμοποιήθηκε, είναι 4MB. Προκειμένου να εκτιμηθεί η απόδοση του FTL, κατασκευάζεται απευθείας από το FTL ένα B-δέντρο. Υιοθετείται από το FTL η άπληστη πολιτική ανακύκλωσης μπλοκ, για τη συλλογή απορριμμάτων.

Μεγάλη βαρύτητα, δόθηκε στην εκτίμηση της απόδοσης των λειτουργιών ευρετηρίου (index operations). Ο αριθμός fanout που χρησιμοποιήθηκε στα πειράματα είναι ίσος με 21 και το μέγεθος ενός κόμβου B-δέντρου ‘’χωράει’’ σε έναν τομέα. Για την εκτίμηση της απόδοσής του, το BFTL διαμορφώθηκε ως εξής: η ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) αποθηκεύει εξήντα εγγραφές, ενώ οι λίστες του πίνακα μετάφρασης κόμβων έχουν ως μέγιστο μέγεθος τα τέσσερα στοιχεία.

Στα πειράματα που έλαβαν χώρα, μετρήθηκαν οι μέσοι χρόνοι απόκρισης εισαγωγών και διαγραφών (insertions and deletions). Όσο μικρότερος ο μέσος χρόνος απόκρισης, τόσο πιο αποδοτική είναι η λειτουργία εισαγωγής/διαγραφής. Μετρήθηκε επίσης, το πλήθος των σελίδων που αναγνώστηκαν (page read), το πλήθος των σελίδων που εγγράφηκαν (page write) καθώς και το πλήθος των σβησμένων μπλοκ (erased block). Επιπλέον, μετρήθηκαν, η κατανάλωση ενέργειας των

FTL και BFTL, και τα αποτελέσματα των πειραμάτων παρουσιάζονται παρακάτω.

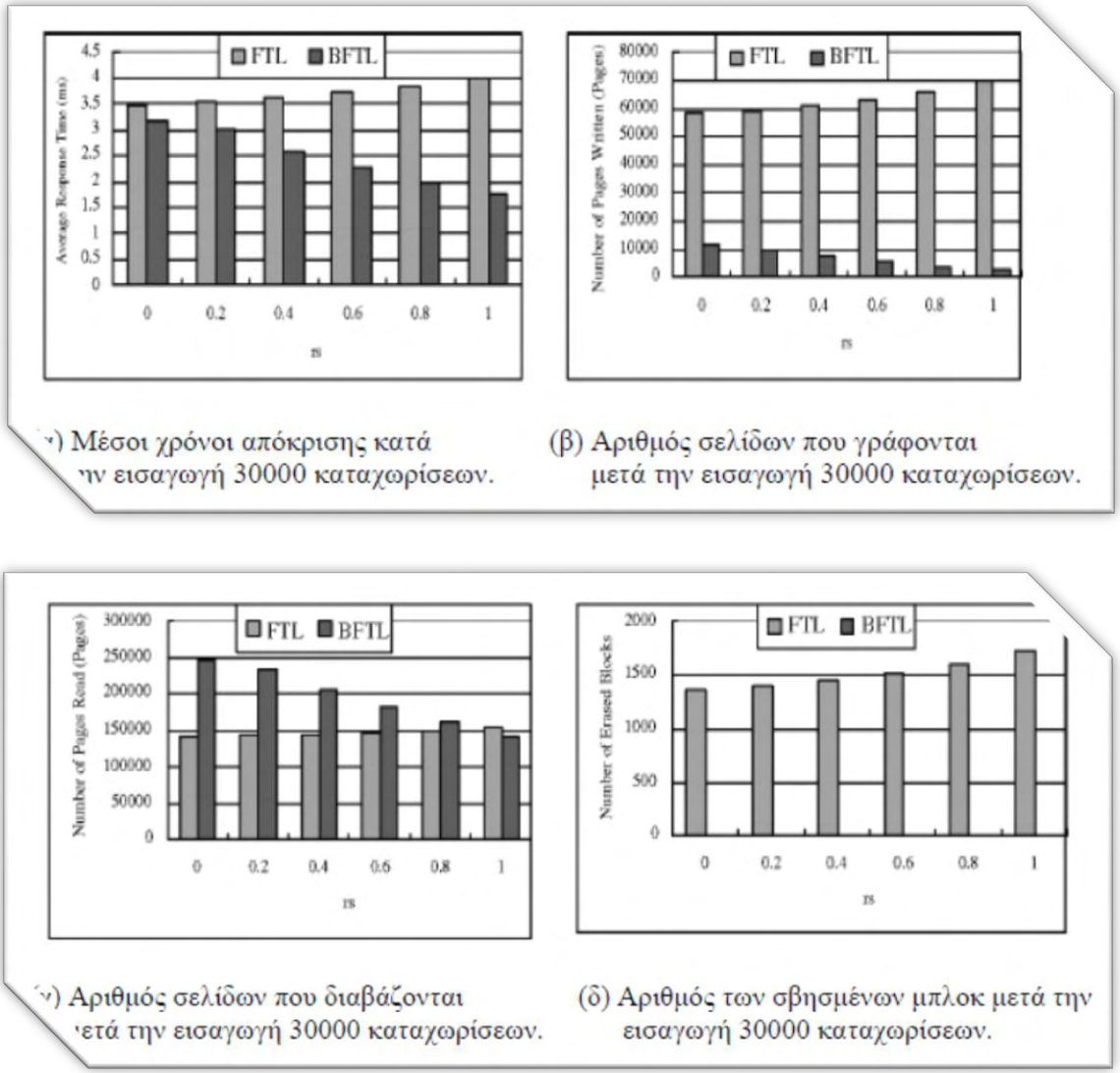
4.5.2. Απόδοση δημιουργίας Δομών Ευρετηρίου B-δέντρων

Τα αποτελέσματα των πειραμάτων σχετικά με την απόδοση του FTL και του BFTL, κατά τη δημιουργία B-δέντρων. Ο φόρτος εργασίας αποτελείται από εισαγωγές εγγραφών (insertions) μόνο. Σε κάθε νέα εκτέλεση των πειραμάτων, εισάγονται 30.000 εγγραφές. Παρόλο που ένα B-δέντρο, που κατασκευάστηκε από 30.000 εγγραφές υπό το FTL, κατέλαβε 1197KB μνήμης Flash, το συνολικό ποσό δεδομένων που γράφηκε από το FTL ήταν 14MB.

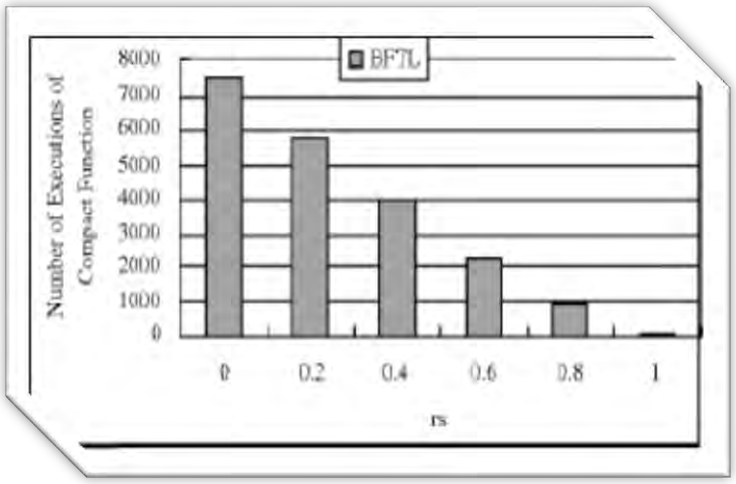
Επειδή στα πειράματα, χρησιμοποιήθηκε NAND Flash, μεγέθους 4MB, οι μηχανισμοί συλλογής απορριμμάτων ενεργοποιήθηκαν πολύ νωρίς για την ανάκτηση ελεύθερου χώρου. Στα πειράματα, χρησιμοποιήθηκε ένας λόγος rs, για τον έλεγχο της κατανομής τιμών των εισαγόμενων κλειδιών:

- Εάν rs == 0, όλα τα κλειδιά παράγονται τυχαία
- Εάν rs == 1, τότε οι τιμές των εισαγόμενων κλειδιών ακολουθούν αύξουσα σειρά.
- Εάν rs == 0.5, οι τιμές των μισών κλειδιών, ακολουθούν αύξουσα σειρά, ενώ τα υπόλοιπα κλειδιά παράγονται τυχαία.

Στις Εικόνες 4.5.2.1 και 4.5.2.2, ο άξονας των x συμβολίζει τις τιμές του rs.



<u>Εικόνα 4.5.2.1</u> : *Αποτελέσματα πειραμάτων κατά τη δημιουργία δομών B-δέντρου.*[1]



<u>Εικόνα 4.5.2.2</u> : *Αριθμός εκτελέσεων της συνάρτησης συμπίεσης.*[1]

Πιο αναλυτικά:

Η Εικόνα 2.5.2.1(α) δείχνει το μέσο χρόνο απόκρισης των λειτουργιών εισαγωγής. Παρατηρούμε ότι το BFTL είναι πολύ πιο αποδοτικό από το FTL. Ειδικότερα, ο χρόνος απόκρισης του BFTL είναι 1/3 του αντίστοιχου χρόνου του FTL, όταν οι τιμές των κλειδιών ακολουθούν αύξουσα σειρά(rs=1). Το BFTL ξεπερνά σε απόδοση το FTL ακόμα και όταν οι τιμές των κλειδιών, παράγονται τυχαία (rs=0). Όταν τα κλειδιά παράγονται ακολουθιακά (rs=1), ο αριθμός των τομέων που γράφονται, μειώνεται, επειδή οι μονάδες ευρετηρίου που ανήκουν σε ένα κόμβο , δε διασκορπίζονται σε μεγάλο βαθμό στη μνήμη Flash. Επιπλέον, το μέγεθος των λιστών του πίνακα μετάφρασης κόμβων είναι σχετικά μικρό και η συμπίεση των λιστών δεν αποφέρει ιδιαίτερο φόρτο εργασίας.

Όπως προαναφέρθηκε, η εγγραφή στη μνήμη Flash θεωρείται σχετικά ακριβή ως λειτουργία, διότι οι εγγραφές φθείρουν τη μνήμη Flash, καταναλώνουν περισσότερη ενέργεια και προκαλεί την εμφάνιση της διαδικασίας συλλογής απορριμμάτων.

Οι Εικόνες 4.5.2.1(β) και 4.5.2.1(γ) δείχνουν τον αριθμό των σελίδων που εγγράφονται και των αριθμό των σελίδων που αναγιγνώσκονται προκειμένου να πραγματοποιηθούν τα πειράματα. Οι αριθμοί, αντιπροσωπεύουν τις χρήσεις της μνήμης Flash από το FTL και BFTL, αντίστοιχα. Θέλοντας να ερευνήσουμε περαιτέρω τη συμπεριφορά των FTL και BFTL, συνδυάζοντας τις Εικόνες 4.5.2.1(β) και 4.5.2.1(γ), παρατηρείται ότι, το BFTL ανταλλάσσει αποδοτικά εγγραφές με πρόσθετες (extra) αναγνώσεις, υιοθετώντας την πολιτική αποθήκευσης (commit policy).

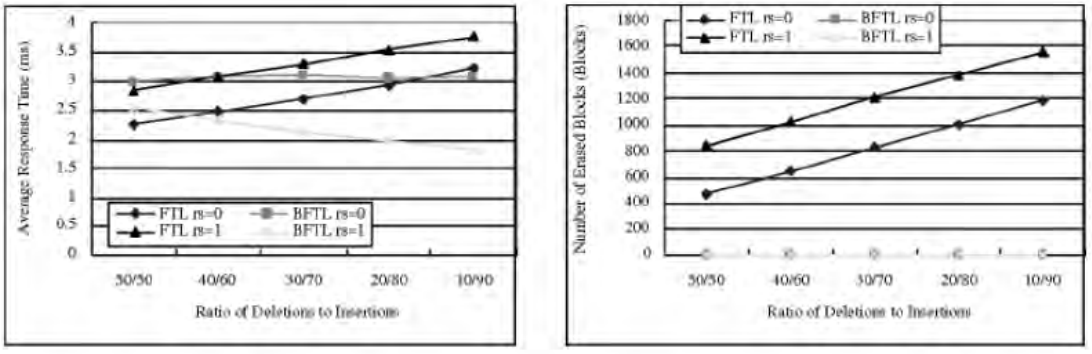
Όσον αφορά τη διαδικασία συλλογής απορριμμάτων, στην Εικόνα 4.5.2.1(δ) διακρίνεται ότι το BFTL υπερτερεί ολοκληρωτικά εις βάρος του FTL. Σε όλα τα πειράματα που διεξήχθησαν, ο μηχανισμός συλλογής απορριμμάτων δεν ενεργοποιήθηκε καθόλου. Ως αποτέλεσμα, το BFTL μπορεί να εγγυηθεί μεγαλύτερη διάρκεια ζωής (lifetime) της μνήμης Flash.

Στην Εικόνα 4.5.2.2, παρουσιάζεται ο φόρτος εργασίας που εισάγεται στο σύστημα, εξαιτίας της συμπίεσης (compaction) του πίνακα

μετάφρασης κόμβων. Ειδικότερα, το πλήθος εμφάνισης της λειτουργίας συμπίεσης, μειώνεται όταν οι τιμές των εισαγόμενων κλειδιών, ακολουθούν αύξουσα σειρά. Αντίθετα, το BFTL συμπιέζει συχνότερα τον πίνακα μετάφρασης κόμβων όταν οι τιμές των εισαγόμενων κλειδιών παράγονται τυχαία, αφού οι μονάδες ευρετηρίου ενός κόμβου, διασκορπίζονται τυχαία στους διάφορους τομείς. Συνεπώς, το μέγεθος των λιστών μεγαλώνει ταχύτατα προκαλώντας το φαινόμενο της συμπίεσης συχνότερα.

4.5.3. Απόδοση Διατήρησης B-δέντρων

Σε αυτό τον τομέα, παρουσιάζονται τα αποτελέσματα των πειραμάτων σχετικά με την απόδοση του FTL και του BFTL, κατά τη διατήρηση των B-δέντρων. Ειδικότερα, κατά τη διάρκεια των πειραμάτων που διεξήχθησαν, πραγματοποιήθηκαν εισαγωγές, διαγραφές και τροποποιήσεις εγγραφών. Για παράδειγμα, ο λόγος 30/70 δηλώνει ότι το 30% των συνολικών πράξεων ήταν διαγραφές (deletions) και το υπόλοιπο 70% των συνολικών πράξεων ήταν εισαγωγές (insertions). Σε κάθε επανάληψη των πειραμάτων, πραγματοποιήθηκαν 30.000 πράξεις επί των B-δέντρων, με το λόγο *εισαγωγές / διαγραφές* να κυμαίνεται μεταξύ 50/50, 40/60, 30/70, 20/80 και 10/90. Για τα πειράματα, η τιμή του rs ήταν ή 0 ή 1.



(α) Μέσοι χρόνοι απόκρισης κάτω από διάφορους λόγους διαγραφών/εισαγωγών.

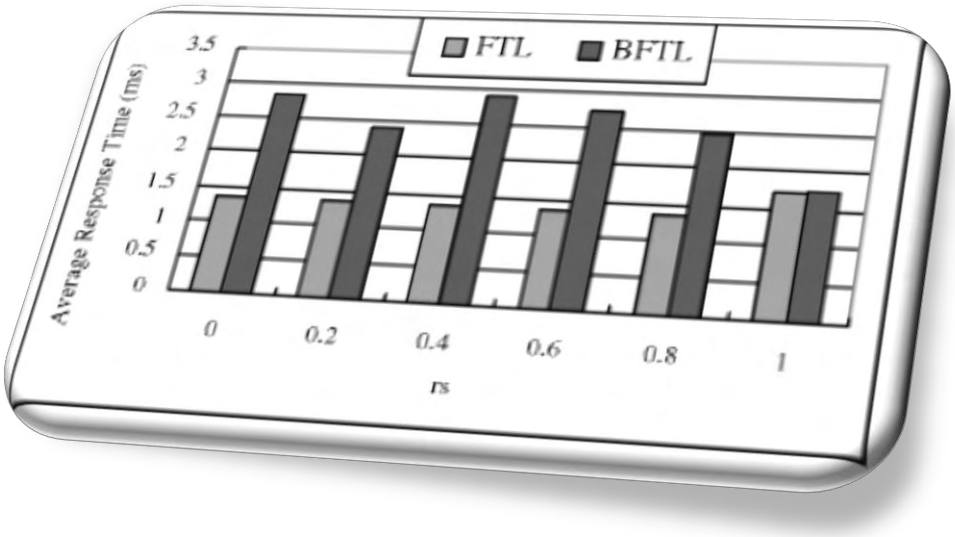
(β) Αριθμός σβησμένων μπλοκ κάτω από διάφορους λόγους διαγραφών/εισαγωγών.

Εικόνα 4.5.3 [1]

Στην Εικόνα 4.5.3, διακρίνονται τα αποτελέσματα των πειραμάτων κατά τη διατήρηση B-δέντρων. Πιο αναλυτικά, οι μέσοι χρόνοι απόκρισης, δείχνουν ότι το FTL είναι πιο αποδοτικό από το BFTL, όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 20/80, με rs=0. Από τη στιγμή που τα εισαγόμενα κλειδιά έχουν ήδη παραχθεί τυχαία και στην πορεία, διαγράφονται επίσης τυχαία, όλο και περισσότερες μονάδες ευρετηρίου ενώνονται μεταξύ τους στον πίνακα μετάφρασης κόμβων με αποτέλεσμα η επίσκεψη ενός κόμβου να καθίσταται πλέον μη αποδοτική. Στην περίπτωση τώρα, που ο λόγος διαγραφές/εισαγωγές ξεπερνά την αναλογία 20/80, με rs=0, το BFTL είναι πιο αποδοτικό από το FTL, επειδή ο αριθμός των εισαγωγών αυξήθηκε με αποτέλεσμα το FTL να γράψει περισσότερες σελίδες. Όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 10/90, με rs=1, η απόδοση του BFTL βελτιώνεται ριζικά. Η αιτία, έγκειται στο γεγονός ότι το BFTL γράφει λιγότερες σελίδες, όταν οι τιμές των εισαγόμενων κλειδιών ακολουθούν αύξουσα σειρά. Συνεπώς, το BFTL έχει καλύτερη απόδοση από το FTL, όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 10/90 με, με rs=1. Τέλος, στην Εικόνα 4.5.3(β) διακρίνεται ο αριθμός των σβησμένων μπλοκ στα πειράματα. Οι δραστηριότητες της διαδικασίας συλλογής απορριμμάτων, έχουν μειωθεί αισθητά, από το BFTL. Ειδικότερα, δεν έχουν ακόμη ξεκινήσει, σε όλη τη διάρκεια των πειραμάτων περί του BFTL.

4.5.4. Αποτελέσματα Πειραμάτων σχετικά με την απόδοση του FTL και BFTL κατά την αναζήτηση στοιχείων σε B-δέντρα

Αρχικά, τα B-δέντρα, δημιουργήθηκαν εισάγοντας 30.000 καταγραφές με διαφορετική τιμή κάθε φορά, του λόγου rs. Σε κάθε επανάληψη των πειραμάτων, πραγματοποιήθηκαν 3000 αναζητήσεις κλειδιών με διαφορετικές τιμές, ενώ μετρήθηκαν οι μέσοι χρόνοι απόκρισης των αναζητήσεων.



Εικόνα 4.5.4 [1]

Όπως φαίνεται και από Εικόνα 4.5.4, από τη στιγμή που οι μονάδες ευρετηρίου των κόμβων του B-δέντρου μπορούν να διασκορπίζονται στη μνήμη Flash, εξαιτίας της πολιτικής αποθήκευσης (commit policy), το BFTL χρειάζεται να διαβάσει περισσότερους τομείς για την κατασκευή των κόμβων. Συνεπώς, στο BFTL, οι λειτουργίες αναζήτησης στοιχείων, διαρκούν περισσότερο χρόνο, από ότι στο FTL. Ορίζεται επίσης, ένα άνω όριο (C) στο μέγεθος των λιστών του πίνακα μετάφρασης κόμβων, ώστε να αποφευχθεί το πρόβλημα των μεγάλων χρόνων αναζήτησης στο BFTL. Στα πειράματα, των οποίων τα αποτελέσματα φαίνονται στο σχήμα 8, το C παίρνει την τιμή 4, με αποτέλεσμα οι μέσοι χρόνοι απόκρισης των αναζητήσεων στο BFTL να είναι διπλάσιοι αυτών του FTL. Ας μην παραλειφθεί το γεγονός όμως, ότι το BFTL μείωσε δραστικά τον αριθμό των εγγραφών σε επίπεδο σελίδας καθώς και τον αριθμό των σβησμένων μπλοκ (erased block number).

Βιβλιογραφία

[1]http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.3814&rep=rep1&type=pdf

[2]http://lambda.csail.mit.edu/~chet/papers/others/l/log-structured/rtcsa03_btreeflash.pdf

Κεφάλαιο 5

5.1 Εισαγωγή στα AS B-δέντρα

Εφόσον οι διαδοχικές λειτουργίες εισόδου/εξόδου μπορούν να συγχωνευτούν και να μετατραπούν σε μία λειτουργία εισόδου/εξόδου ενός μεγάλου κομματιού εξ'αυτών, η οποία θα καλυφθεί από τα πολλαπλά πακέτα μνήμης Flash μέσα στον δίσκο SSD, δίνεται η δυνατότητα παράλληλης μεταφοράς δεδομένων στα πολλαπλά πακέτα μνήμης Flash την ίδια χρονική στιγμή. Επιπλέον, διαδοχικές λειτουργίες εισόδου/εξόδου ενισχύουν την απόδοση εκκαθάρισης του στρώματος FTL βελτιώνοντας έτσι την μακρόχρονη επίδοση των δίσκων SSD. [1]

Ωστόσο, η δομή των B^+-δέντρων, η οποία είναι ένα αντιπροσωπευτικό ευρετήριο συστημάτων διαχείρισης βάσεων δεδομένων, παράγει ακατάπαυστα λειτουργίες εισόδου/εξόδου, με τυχαία σειρά, όταν οι δομές των κόμβων τους είναι ενημερωμένες. Επομένως, η συμβατική δομή των B+ -δέντρων είναι ασύμφορη για χρήση στους δίσκους SSD. [2,3]

Σε αυτό το σημείο λοιπόν προτείνεται το ***AS B-δέντρο*** (Always Sequential B-tree), το οποίο σε κάθε λειτουργία ανανέωσης γράφει τους τροποποιημένους κόμβους στο τέλος των αρχείων του. Το LBA του κόμβου που υπήρχε στα B^+-δέντρα αντικαθίσταται τώρα από την ταυτότητα κόμβου (NodeID). Ο ανανεωμένος κόμβος τοποθετείται πάντα σε ένα νέο LBA που μόλις του έχει ανατεθεί, στο τέλος του αρχείου. Κάθε LBA συνδέεται με το αντίστοιχο NodeID, με τη βοήθεια του πίνακα αντιστοίχησης διευθύνσεων (Address Mapping Table) που βρίσκεται στην κύρια μνήμη. Με αυτό τον τρόπο, το AS B-δέντρο, κάνει τις εγγραφές των ανανεωμένων κόμβων βάσει των συνεχόμενων LBA προκειμένου να εκμεταλλευτεί όπως αναφέρθηκε παραπάνω, την παραλληλοποίηση που προσφέρουν οι δίσκοι SSD.

5.2 AS B-δέντρα VS LA-δέντρα, BFTL

Ως τώρα διάφορα ευρετήρια μνήμης Flash έχουν προταθεί, όπως το BFTL [5] και το LA- δέντρο [6]. Αυτά τα ευρετήρια μοιράζονται την ίδια βασική ιδέα, να

ελαχιστοποιήσουν τις λειτουργίες εγγραφής στους δίσκους SSD υποβαθμίζοντας την επίδοση αναζήτησης συγκριτικά με αυτή των B^+ -δέντρων.

Σε αντίθεση, τα AS B-δέντρα επικεντρώνονται στο να εκμεταλλευτούν τα οφέλη των διαδοχικών λειτουργιών εισόδου/εξόδου στους δίσκους SSD αντικαθιστώντας τις τυχαίες εγγραφές με τις διαδοχικές εγγραφές, χωρίς όμως να υποβαθμίζουν την επίδοση αναζήτησης συγκριτικά με αυτήν των B^+-δέντρων.

5.3 Ανάλυση AS B-δέντρου

Ένα AS B-δέντρο λοιπόν, αποτελείται από την αρχική δομή του ***B^+-δέντρου***, έναν ***Sequential Writer (SW)***, έναν ***Write Buffer (WB)***, έναν ***Address Mapping Table (AMT - πίνακας αντιστοίχησης διευθύνσεων*** και ένα ***Node Validation Manager (NVM)***. Στην παρακάτω εικόνα (Εικόνα 4), παρουσιάζεται η αρχιτεκτονική ενός AS B- δέντρου.

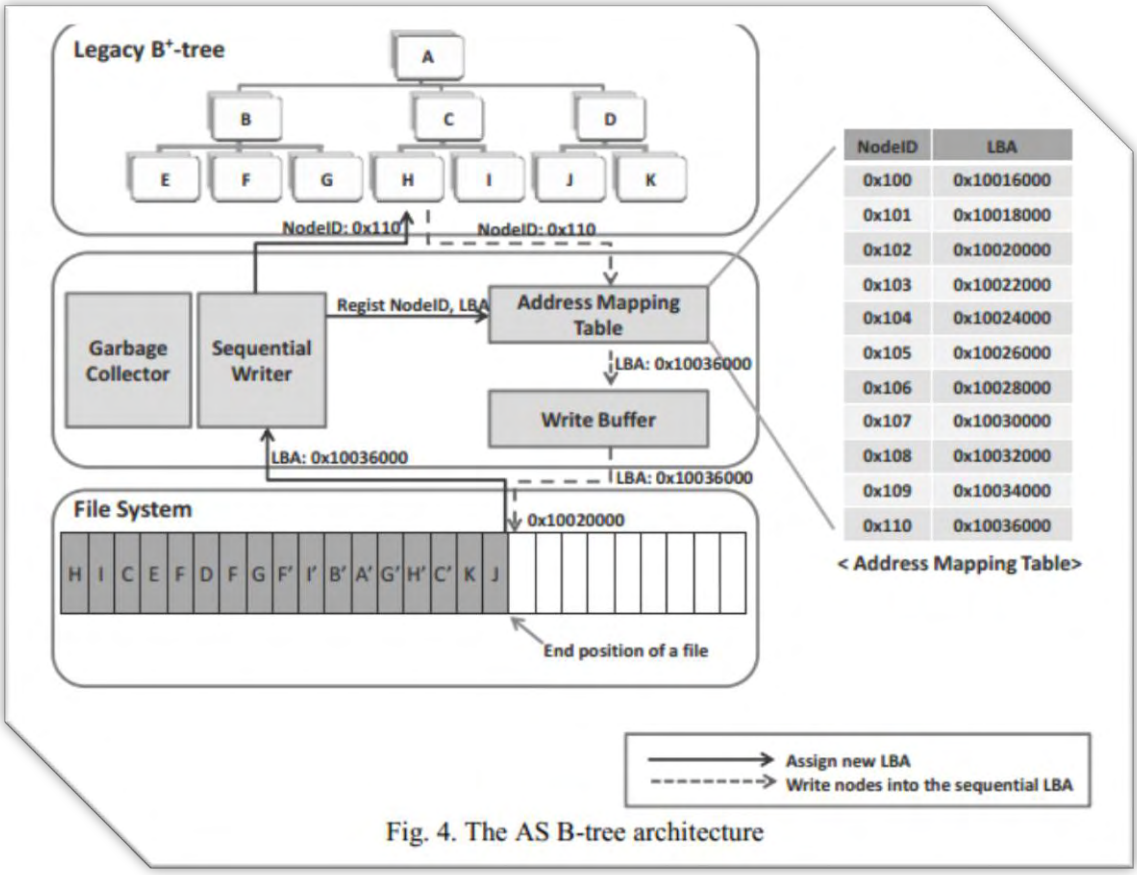


Fig. 4. The AS B-tree architecture

[4]

Ας αρχίσουμε να αναλύουμε λοιπόν ένα προς ένα τα κομμάτια από τα οποία αποτελείται αυτή η δομή:

5.3.1. Sequential Writer - SW

Προκειμένου να χρησιμοποιηθεί αποδοτικά η εσωτερική παραλληλοποίηση των δίσκων SSD, όλοι οι τροποποιημένοι κόμβοι θα γραφούν σε διαδοχικά block λογικών διευθύνσεων (Logical Block Addresses - LBA). Ο κόμβος που τροποποιήθηκε από μια λειτουργία εισαγωγής/διαγραφής/ανανέωσης, θα γραφεί στο τέλος του αρχείου, αντί να επανεγγραφεί (out of place/update). Αυτή η διαδικασία αποθήκευσης δεδομένων, αναγκάζει όλους τους τροποποιημένους κόμβους να γραφούν διαδοχικά.

```
Procedure: Insert(K, NodeID)
Input: K(inserted key), NodeID(node ID number)
begin
1:    LBA ← MapTbl[NodeID]; // MapTbl[]:   table that maps NodeID to LBA;
2:    Node ← read from LBA; // Node: data of a node.
3:    InsertKey(K, Node);
4:    LBA ← get the end position of the file;
5:    MapTbl[NodeID] ← LBA;
6:    write Node to LBA;
7:    Function InsertKey(K, Node)
8:        //Node.KEYS: the array of the keys in the Node
9:        I ← search the location that should be inserted in the Node.KEYS;
10:       Node.KEYS[i] ← K;
end
```

Algorithm 1. Assignment of the sequential LBA

[4]

Ο αλγόριθμος 1, παρουσιάζει μια λεπτομερή εξήγηση αυτής της διαδικασίας. Όταν ένας κόμβος έχει τροποποιηθεί, γράφεται σε ένα νέο LBA (σειρές 4-6) αντί να γραφεί στο LBA στο οποίο ήταν πριν τοποθετημένο. Η νέα διεύθυνση τοποθετείται στο τέλος του αρχείου. Το AS B- δέντρο έχει πρόσβαση σε κάθε κόμβο του, βάση της μοναδικής ταυτότητάς του (unique node ID number – NodeID), αντί των λογικών διευθύνσεων τους. Στον αλγόριθμο 1, NodeID και LBA υποδηλώνουν την ταυτότητα του κόμβου και τη λογική διεύθυνση των block δεδομένων αντίστοιχα. Ακόμα και αν ο τροποποιημένος κόμβος είναι νέο-εγγεγραμμένος στο τέλος του αρχείου, το αρχικό NodeID του κόμβου, δεν έχει αλλάξει. Για να αποκτήσουμε πρόσβαση σε έναν κόμβο του κληροδοτημένου B^+- δέντρου χρησιμοποιώντας το NodeID του, χρησιμοποιείται ο πίνακας αντιστοίχησης διευθύνσεων (Address Mapping Table - AMT), ο οποίος μετατρέπει ένα NodeID στο αντίστοιχο LBA (σειρές 1-2).

5.3.2. Write Buffer - WB

Όταν μία εγγραφή ευρετηρίου εισάγεται σε ένα B^+- δέντρο, διαβάζονται τόσοι κόμβοι όσοι και το ύψος του δέντρου, προκειμένου να τοποθετηθεί ο κόμβος φύλλο που περιέχει την εγγραφή ευρετηρίου.

Για παράδειγμα, σε ένα B^+- δέντρο του οποίου το ύψος είναι τρία, ακολουθούν τα εξής βήματα για την εισαγωγή μιας εγγραφής ευρετηρίου: Αρχικά διαβάζεται ο κόμβος-ρίζα. Έπειτα διαβάζεται ένας εσωτερικός κόμβος. Έπειτα διαβάζεται ένας κόμβος-φύλλο και τελικά ο τροποποιημένος κόμβος φύλλο εγγράφεται. Αυτό υποδεικνύει ότι κάθε λειτουργία εισαγωγής παράγει μια λειτουργία εγγραφής για κάθε δύο λειτουργίες ανάγνωσης (λειτουργία διαβάζω-προτού-γράψω – read-before-write operation) [7,8]. Εάν η βάση δεδομένων κάνει χρήση ενός buffer στην κρυφή μνήμη (buffer cache), τότε κάποιες λειτουργίες εγγραφής και ανάγνωσης μπορεί να μη συμβούν. Εάν ο ζητούμενος κόμβος είναι στον buffer, τότε είτε κάποιο βήμα της λειτουργίας εγγραφής είτε κάποιο βήμα της λειτουργίας ανάγνωσης, θα

παραληφθεί. Αυτή η λειτουργία read-before-write, συμβαίνει ακόμα και αν ο buffer είναι γεμάτος.

Εάν ο κόμβος δεν είναι στον buffer, τότε αυτές οι λειτουργίες θα συμβούν και έπειτα ο διαβασμένος ή ο εγγεγραμμένος κόμβος θα αποθηκευτεί στον buffer. Πριν την αποθήκευση αυτή όμως, ένας κόμβος που έχει ήδη επιλεχθεί ως κόμβος-θύμα βάσει της εκάστοτε πολιτικής αντικατάστασης του buffer πρέπει να αποθηκευτεί στον δίσκο.

Σύμφωνα με κάποια αποτελέσματα πειραμάτων για έλεγχο των δίσκων SSD, μια λειτουργία read-before-write αλλοιώνει την επίδοση τους [9]. Ως εκ τούτου, ακόμα και αν οι τροποποιημένοι κόμβοι έχουν γραφεί σε διαδοχικές LBA, δεν υπάρχει όφελος από την εσωτερική παραλληλοποίηση των δίσκων SSD λόγω αυτή της λειτουργίας.

Ακόμα όμως και αν οι τροποποιημένοι κόμβοι, είναι εγγεγραμμένοι σε διαδοχικά LBA, λόγω της λειτουργίας read-before-write, δε καθίσταται εκμεταλλεύσιμη η παραλληλοποίηση των SSD.

Ο write buffer, χρησιμοποιείται για να επιλύσει αυτό το πρόβλημα, όπως φαίνεται και στην εικόνα 5. Αρχικά, ο τροποποιημένος κόμβος τοποθετείται προσωρινά στον write buffer, ο οποίος με τη σειρά του περιέχεται στην κύρια μνήμη, όχι απευθείας στον δίσκο SSD. Έπειτα, αν ο write buffer είναι γεμάτος, οι κόμβοι μετακινούνται στις αντίστοιχες LBA.

Όταν ο write buffer δεν χρησιμοποιείται, οι λειτουργίες ανάγνωσης που βρίσκονται ανάμεσα στις λειτουργίες εγγραφής διακόπτουν τις λειτουργίες εγγραφής, αναστέλλοντας έτσι τη βελτίωση επίδοσης των λειτουργιών εισαγωγής.

Όταν ο write buffer χρησιμοποιείται, τότε μόνο οι λειτουργίες ανάγνωσης πραγματοποιούνται. Οι λειτουργίες εγγραφής καθυστερούν τοποθετώντας τους τροποποιημένους κόμβους στον write buffer της κύριας μνήμης. Με αυτόν τον τρόπο, μπορούν να πραγματοποιηθούν αλλεπάλληλες λειτουργίες εγγραφών αμέσως μετά

από αλλεπάλληλες λειτουργίες ανάγνωσης, χωρίς έτσι να υποβαθμίζεται η επίδοση των δίσκων SSD.

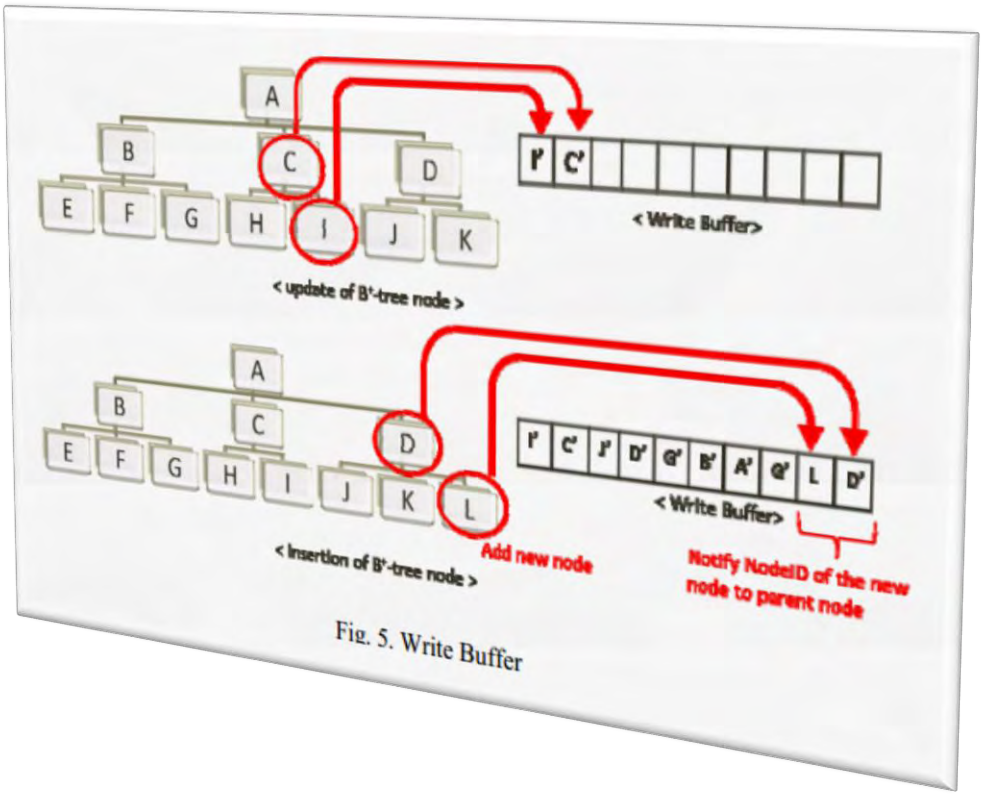


Fig. 5. Write Buffer

[4]

Αλγόριθμος 2

```
Procedure: WriteNode(NodeID, Node)
Input: NodeID(node ID number), Node(data of the modified node)
begin
1:    bFound = FALSE;
2:    i=0;
3:    for i < WBN    do     // WBN: the number of nodes in Write Buffer
4:        if(WBA[i] = NodeID)   then     // WBA[]: NodeIDs of nodes in Write Buffer
5:          copy Node to WBD[i];     // WBD[]:data of nodes in Write Buffer
6:                bFound = TRUE;
7:        endif
8:        i++;
9:    endfor
10:   if(bFound = FALSE)   then
11:       i ← get an index of the empty buffer in Write Buffer;
12:       WBA[i] ← NodeID;
13:       copy Node to WBD[i];
14:       WBN++;
15:       if (WBN = MS) then                  // MS: the maximum size of Write Buffer
16:          i=0, WBN=0;
17:          for i < MS    do
18:              LBA ← MapTbl[WBA[i]];
19:              write WBD[i] to LBA;
20:              i++;
21:          endfor
22:       endif
23:   endif
end
```

Algorithm 2. Writing to a node with Write Buffer

[4]

Ο αλγόριθμος 2, περιγράφει τον τρόπο με τον οποίο οι τροποποιημένοι κόμβοι και ένας νέο-ανατιθέμενος κόμβος στο αντίστοιχο LBA, τοποθετούνται στον write buffer. Η δομή ενός κόμβου που είναι αποθηκευμένος στον write buffer αποτελείται από τη διεύθυνση του write buffer (Write Buffer Address - WBA) και από τα δεδομένα του write

buffer (Write Buffer Data - WBD), όπου το WBA είναι το NodeID του κόμβου που πρόκειται να αποθηκευτεί και το WBD τα δεδομένα αυτού.

Εάν ο write buffer έχει το WBA του κόμβου που έχει ζητηθεί, τότε ο τροποποιημένος κόμβος χρειάζεται να αντικατασταθεί από το αντίστοιχο WBD (σειρές 3-9). Σε αυτή την περίπτωση, ο write buffer λειτουργεί με εκείνον στην κρυφή μνήμη.

Εάν δεν υπάρχει άλλος κόμβος με το ίδιο WBA, τότε ο write buffer αναθέτει στον κόμβο ένα buffer block και τοποθετεί το WBA και το WBD του κόμβου μέσα στο buffer block (σειρές 11-14). Εάν ο write buffer είναι γεμάτος, τότε το WBD του κάθε κόμβου μέσα στον write buffer γράφεται στο αντίστοιχο WBA (σειρές 15-22).

Αλγόριθμος 3

```
Procedure: ReadNode(NodeID)
Input: NodeID(node ID number)
Output: Node(data of the read node)
begin
1:    bFound = FALSE;
2:    i=0;
3:    for    i < WBN   do    // WBN: the number of nodes in Write Buffer
4:        if(WBA[i] = NodeID)   then    // WBA[]: NodeIDs of nodes in Write Buffer
5:            copy WBD[i] to Node;    // WBD[]:data of nodes in Write Buffer
6:            bFound = TRUE;
7:        endif
8:        i++;
9:    endfor
10:   if(bFound = FALSE)   then
11:       LBA ← MapTbl[NodeID];
12:       Node ← read from LBA;
13:   endif
14:   return (Node);
End
```

Algorithm 3. Reading the node with Write Buffer

[4]

Ο αλγόριθμος 3, περιγράφει τον τρόπο με τον οποίο επεξεργάζονται από τον write buffer, οι λειτουργίες ανάγνωσης. Αρχικά, ο write buffer υπόκειται σε μία ανάλυση προκειμένου να αποφασιστεί εάν περιέχει ή όχι τον κόμβο που έχει ζητηθεί να αναγνωστεί. Έπειτα, εάν ο κόμβος υπάρχει, τότε θα διαβαστεί απευθείας από τον write buffer στην κύρια μνήμη (σειρές 3-9). Εάν όμως ο κόμβος δεν υπάρχει στον write buffer, παρέχεται το αντίστοιχο LBA του κόμβου από τον πίνακα αντιστοιχήσεων (AMT), χρησιμοποιώντας το δοσμένο NodeID. Έχοντας το αντίστοιχο LBA, το AS B- δέντρο μπορεί τώρα να διαβάσει τον ζητούμενο κόμβο από τη μνήμη Flash του δίσκου SSD.

5.3.3. Address Mapping Table (AMT - Πίνακας Αντιστοιχήσεων Διευθύνσεων)

Για να τοποθετηθούν οι τροποποιημένοι κόμβοι σε διαδοχικές λογικές διευθύνσεις (LBA's), χρησιμοποιούνται εκτός από τα LBA, τα NodeID's. Η εικόνα 6, δείχνει τη διαδικασία κατά την οποία ο πίνακας αντιστοιχήσεων διευθύνσεων μετατρέπει ένα NodeID στο αντίστοιχο LBA. Εάν ο write buffer δεν έχει άλλο διαθέσιμο χώρο προς αποθήκευση τροποποιημένων κόμβων, τότε κάθε κόμβος του write buffer τοποθετείται στη δική του LBA, σύμφωνα με τον πίνακα αντιστοιχήσεων διευθύνσεων. Επειδή κάθε κόμβος στον write buffer έχει διαδοχική LBA, εγγράφονται τόσοι κόμβοι όσοι και το μέγεθος του write buffer, σε διαδοχικές LBA ενός flashSSD. Όταν έπειτα το B^+-δέντρο διαβάζεται, δημιουργείται ένας πίνακας αντιστοίχησης διευθύνσεων με τα NodeIDs και LBAs από όλους τους κόμβους, διαβάζοντάς τους από τους δίσκους SSD.

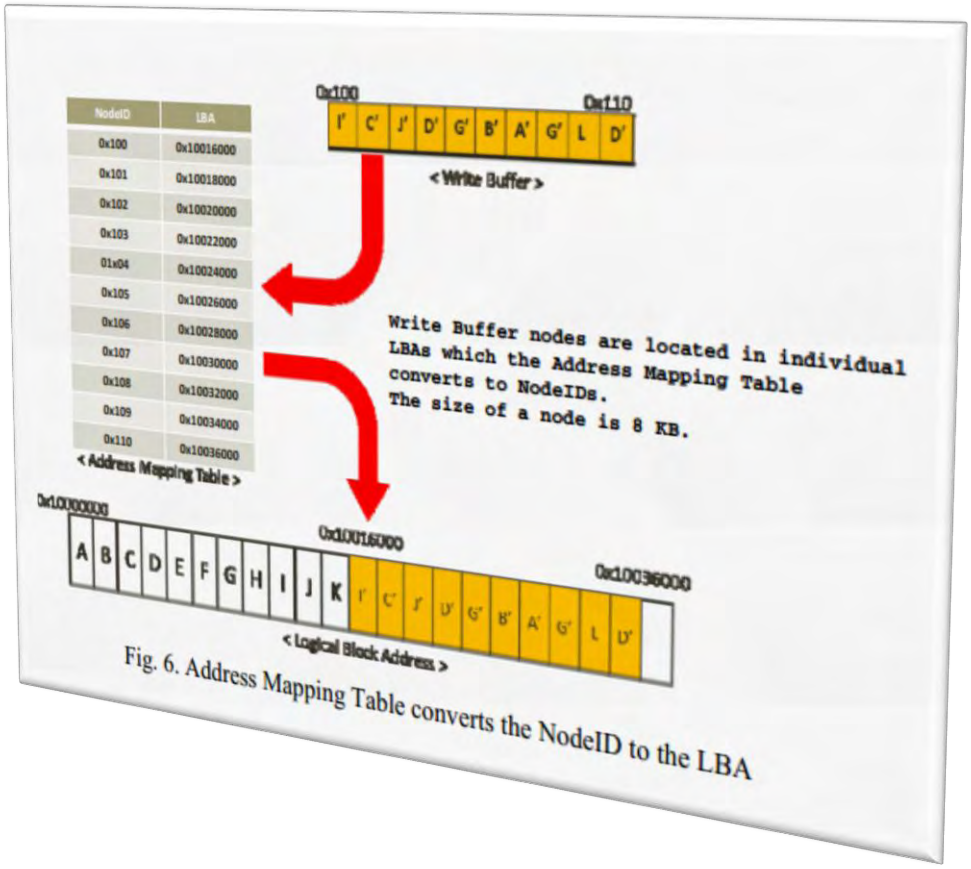


Fig. 6. Address Mapping Table converts the NodeID to the LBA

Σχήμα 6: Οι κόμβοι του write buffer, είναι τοποθετημένοι σε ξεχωριστά LBA's, το οποία και μετατρέπονται στα αντίστοιχα NodeIDs βάσει του Πίνακα Αντιστοίχησης Διευθύνσεων. [4]

5.3.4. Node Validation Manager - NVM

Εφόσον ένας τροποποιημένος κόμβος δεν τοποθετείται στην προηγούμενη διεύθυνση του(LBA) αλλά στο τέλος του αρχείου, είναι σαφές ότι κάθε τροποποίηση αυξάνει το μέγεθός του. Όπως διακρίνεται και στην Εικόνα 7, οι περισσότεροι από τους έγκυρους κόμβους, τοποθετούνται στο τέλος του αρχείου. Ο αριθμός των έγκυρων κόμβων στα παλιά αρχεία τα οποία έχουν μικρούς αριθμούς περιγραφής (25~29) είναι μικρός, ενώ εκείνων στα καινούργια αρχεία τα οποία έχουν υψηλούς αριθμούς περιγραφής (30~34) είναι υψηλός. Αυτό σημαίνει ότι έγκυροι κόμβοι, σπάνια υπάρχουν στην αρχή ή στη μέση του αρχείου. Εάν η διανομή των τιμών των εγγραφών που έγιναν με εισαγωγή δεν είναι ομοιόμορφη, τότε οι έγκυροι κόμβοι είναι διεσπαρμένοι σε μια ευρύτερη περιοχή του SSD. Για αποδοτικότητα χώρου, η περιοχή στην

οποία υπάρχουν μη έγκυροι κόμβοι χρειάζεται να αναδιοργανωθεί. Ένα AS B-δέντρο είναι αποθηκευμένο σε διάφορα αρχεία, καθένα από τα οποία έχει 64MB χώρο αποθήκευσης.

Ο πίνακας αντιστοίχησης διευθύνσεων περιέχει τα LBAs των έγκυρων κόμβων, για αυτό και είμαστε σε θέση να ξέρουμε αν εάν κάθε LBA ‘’δείχνει’’ σε έγκυρο κόμβο. Διατηρώντας μια αρίθμηση των έγκυρων κόμβων σε κάθε αρχείο, καθίσταται δυνατόν να εντοπίσουμε τα αρχεία που περιέχουν μεγάλο αριθμό μη έγκυρων κόμβων και επομένως μπορούν να ανακυκλωθούν.

Εάν ο αριθμός των έγκυρων κόμβων σε ένα αρχείο είναι μικρότερος από T% των συνολικών κόμβων του αρχείου, τότε εκτελείται η διαδικασία της ανακύκλωσης.

Αρχικά, προκειμένου να μεταφερθούν οι έγκυροι κόμβοι σε νέες τοποθεσίες διαβάζονται από το αρχείο και επανεγγράφονται στο τέλος του νεότερου αρχείου.

Στα πειράματα που διεξήχθησαν, το T τέθηκε ίσο με 5%. Μετά την επανεγγραφή, το αρχείο διεγράφη και ο πίνακας αντιστοίχησης διευθύνσεων ανανεώθηκε με τα νέα LBA των μεταφερμένων κόμβων.

Εφαρμόζοντας αυτή την διαδικασία, είναι εμφανής η μείωση του συνολικού μεγέθους του αρχείου του AS B-δέντρου.

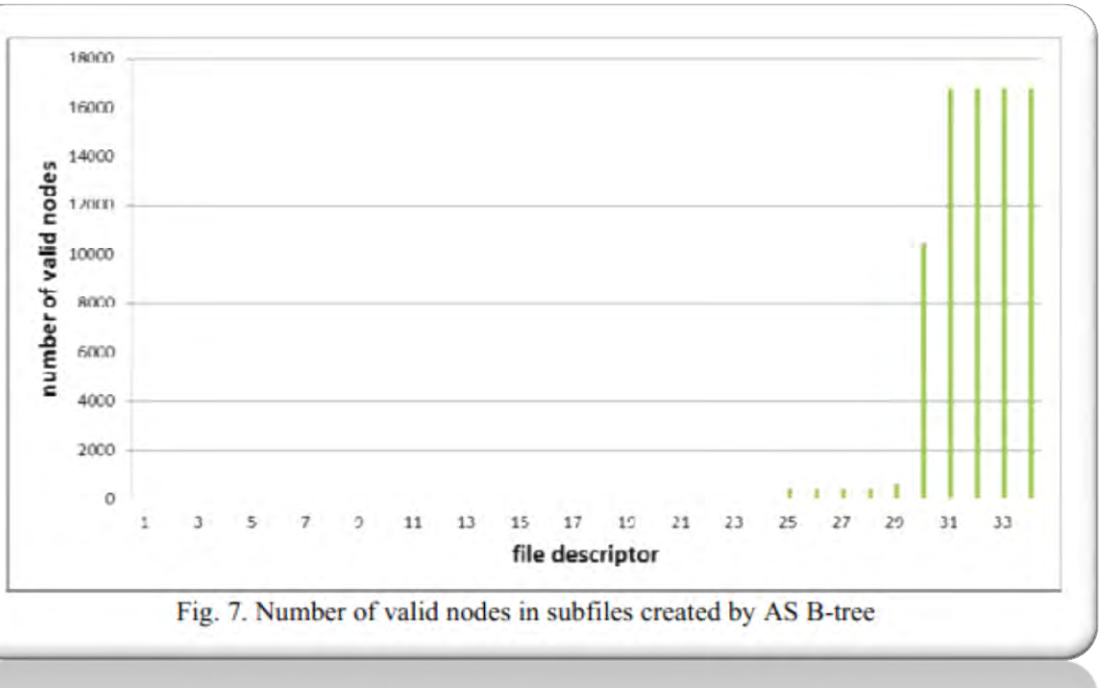


Fig. 7. Number of valid nodes in subfiles created by AS B-tree

[4]

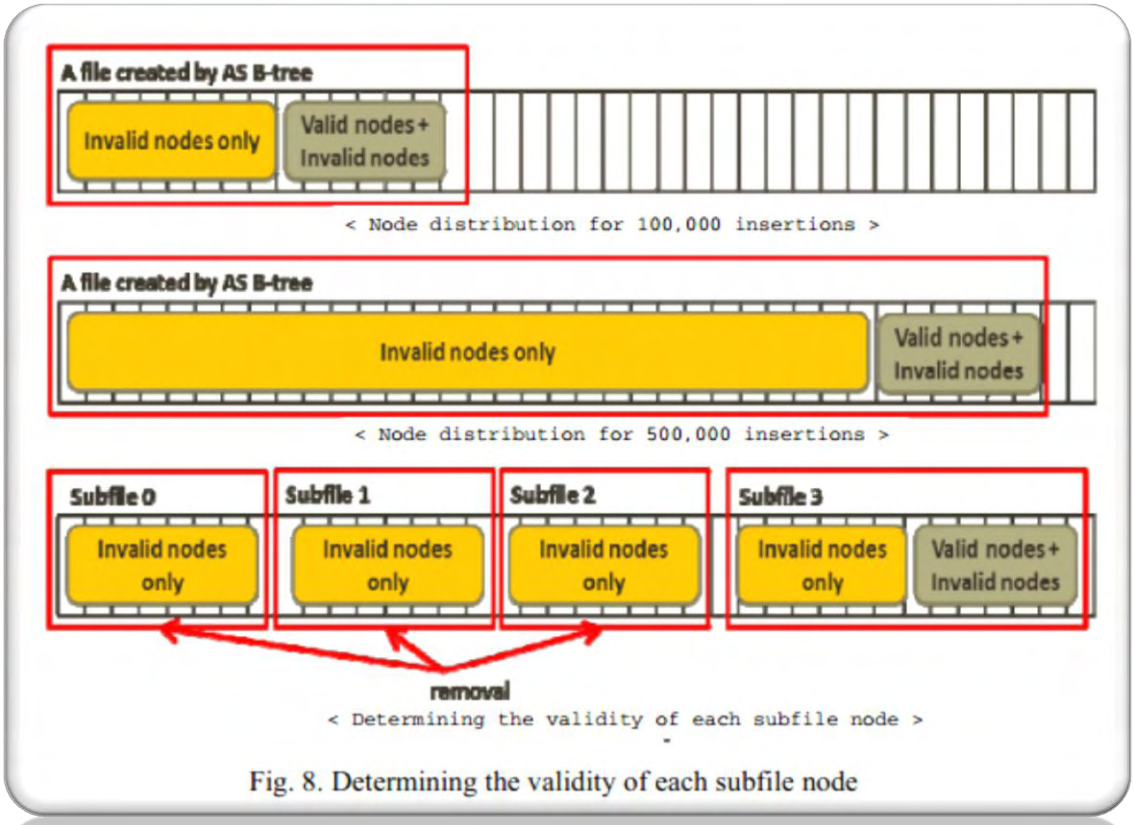


Fig. 8. Determining the validity of each subfile node

[4]

5.3.5. Απόδοση Πειραμάτων

Στην Εικόνα 9 (Fig 9), φαίνονται τα αποτελέσματα των λειτουργιών εισαγωγής. Η επίδοση των AS B-δέντρων ήταν πολύ καλύτερη συγκριτικά με τις υπόλοιπες δομές ευρετηρίων. Όπως διακρίνεται και από το σχήμα, η επίδοση του AS B-δέντρου ενισχύονταν καθώς το F (ορίζεται ως ο μέγιστος αριθμός κλειδιών σε έναν κόμβο) και το μέγεθος του write buffer αυξάνονταν. Πιο συγκεκριμένα, το κόστος ανάγνωσης μειώνονται σε κάθε λειτουργία εισαγωγής, ως αποτέλεσμα της μείωσης του ύψους του B^+-δέντρου κατά τη διάρκεια αύξησης του F. Ένα AS B-δέντρο με F ίσο με 256 και μέγεθος write buffer ίσο με 32,

παρουσιάζει 21% καλύτερη επίδοση συγκριτικά με αυτή ενός B^+-δέντρου και 57% υψηλότερη ταχύτητα συγκριτικά με το BFTL.

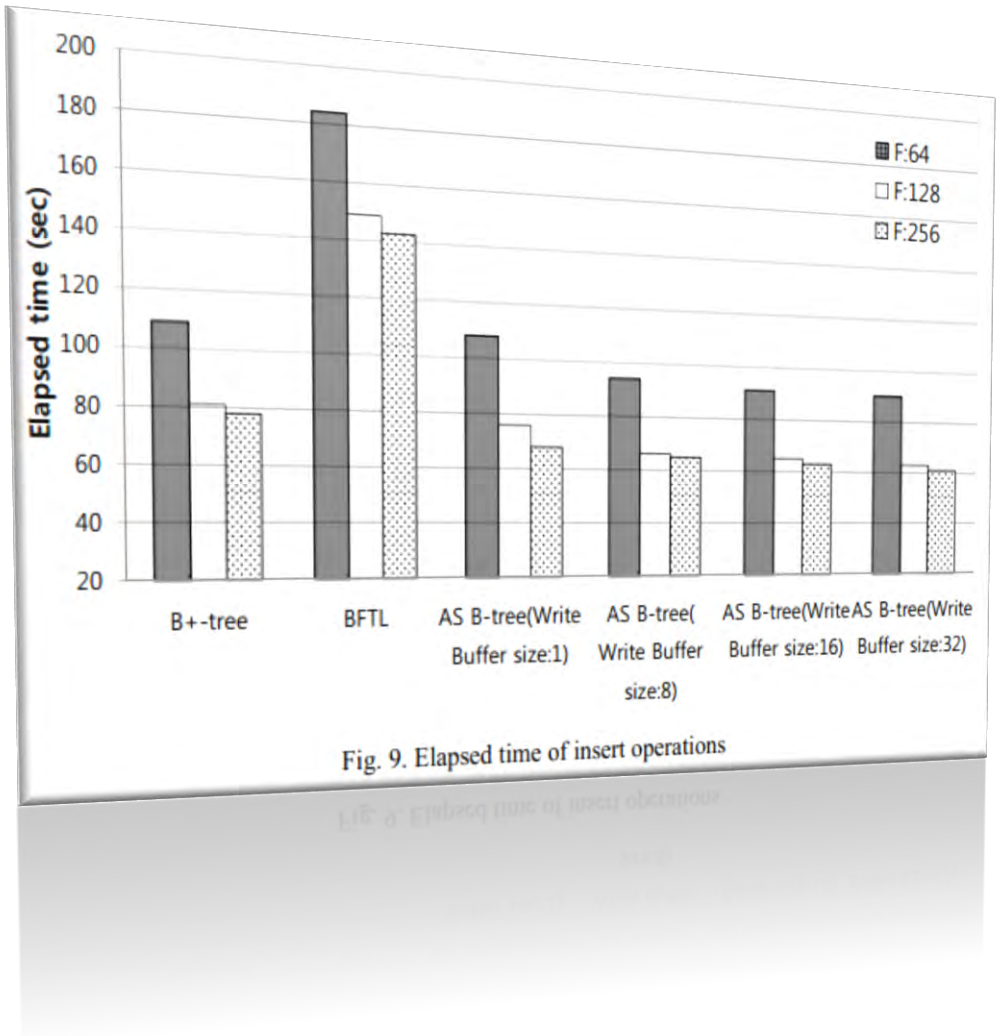


Fig. 9. Elapsed time of insert operations

[4]

Ομοίως, εφαρμόζοντας τις ίδιες συνθήκες με αυτές της εικόνας 9 (για τις λειτουργίες εισαγωγής), μετρήθηκε ο παρερχόμενος χρόνος για 100.000 αναζητήσεις σε ένα AS B-δέντρο, B^+-δέντρο και BFTL όπως φαίνεται στην Εικόνα 10 (Fig 10). Τα αποτελέσματα έδειξαν ότι το AS B-δέντρο, απαιτούσε τον ίδιο χρόνο με αυτόν ενός B^+-δέντρου και λιγότερο χρόνο σε σχέση με το BFTL.

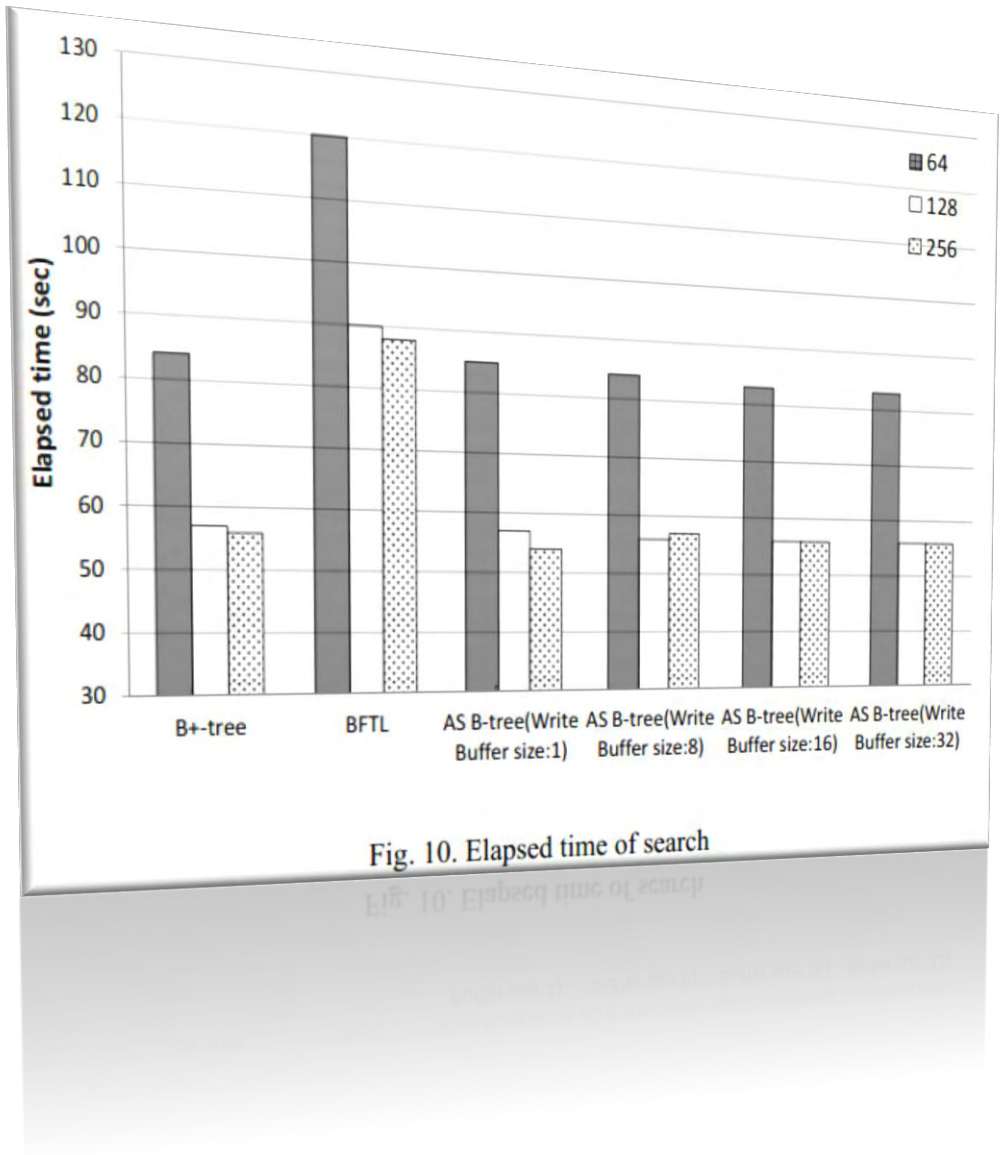


Fig. 10. Elapsed time of search

[4]

Τέλος, στην Εικόνα 11 (Fig 11), παρουσιάζονται τα αποτελέσματα για τις αντίστοιχες 100.000 λειτουργίες ανανέωσης. Το AS B-δέντρο απαιτούσε τον μικρότερο χρόνο και το BFTL τον μεγαλύτερο. Το AS B-δέντρο με το F να ισούται με 256 και το μέγεθος του write buffer να ισούται με 32, πέτυχε 21% καλύτερη επίδοση συγκριτικά με αυτή του B^+-δέντρου (του οποίου ο write buffer είχε επίσης μέγεθος 32) και 32% καλύτερη επίδοση συγκριτικά με αυτή του BFTL.

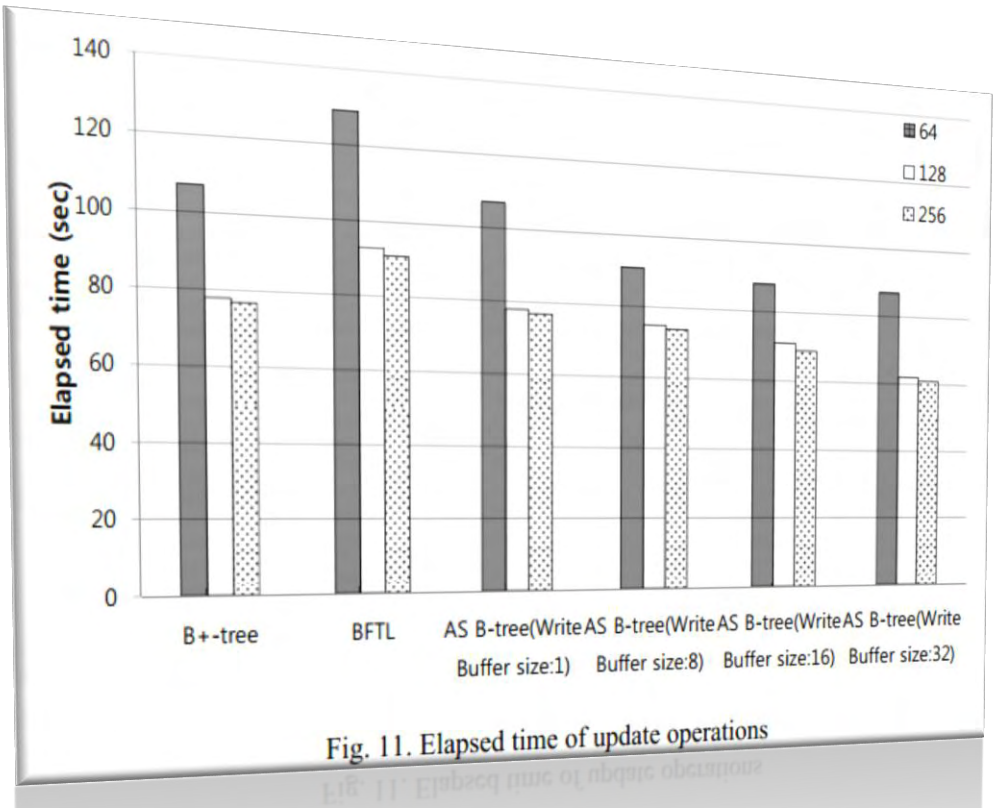


Fig. 11. Elapsed time of update operations

[4]

Βιβλιογραφία

[1] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber, “A Design for High-Performance Flash Disks,” ACM SIGOPS Operating Systems Review, Vol. 41, 2007, pp. 88–93.

[2] Feng Chen, David A. Koufaty, and Xiaodong Zhang, “Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives,” SIGMETRICS '09 Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, 2009, pp. 181–192.

[3] Li-Pin Chang,Tei-Wei Kuo, “Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation,” ACM Transactions on Storage (TOS), Vol. 1, 2005, pp. 381–418.

[4] http://journal.iis.sinica.edu.tw/paper/1/110517-4.pdf?cd=69E4DA0F006F476C5

[5] Bayer, R. and McCreight, “Organization and Maintenance of Large Ordered Indices,”ACTA Informatica, Vol. 1, 1972, pp. 173–189.

[6] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D, “Davis, Mark Manasse and Rzna. Design tradeoffs for SSD performance,” USENIX 2008 Annual Technical Conference on Annual Technical Conference, 2008, pp. 57–70.

[7] D. Kang, D. Jung, J. U. Kang, and J. S. Kim, “μ-Tree : An Ordered Index Structure for NAND Flash Memory,” Proceedings of the 7th ACM & IEEE international conference on Embedded software, 2007.

[8] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and Walid A. Najjar, “Microhash: An Efficient Index Structure for Flash-based Sensor Devices,” In Proc. USENIX Conference on File and Storage Technologies (FAST), 2005.

[9] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, Sang-Woo Kim, “A Case for Flash Memory SSD in Enterprise Database Applications,” SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, pp. 1075–1086

Κεφάλαιο 6

6.1 Εισαγωγή στα FlashB - δέντρα

Σε αυτό το κεφάλαιο, γίνεται αρχικά μια ανάλυση για τους 2 μηχανισμούς που είναι κατάλληλοι για τις περιπτώσεις στις οποίες ο φόρτος εργασίας αποτελείται κυρίως από λειτουργίες εγγραφής και λειτουργίες ανάγνωσης αντίστοιχα. Επισημαίνονται επίσης, τα μειονεκτήματα των δύο αυτών μηχανισμών και προτείνεται ως λύση το καινοτόμο σχήμα B-δέντρου, FlashB-δέντρο, το οποίο και τείνει να μηδενίσει τον αριθμό αναδιοργανώσεων του δέντρου. Υιοθετείται έπειτα ο *online transition algorithm* με σκοπό την αξιοποίηση της υπεροχής των δύο αντιπροσωπευτικών μηχανισμών. Στη συνέχεια παρουσιάζονται κάποιες μετρήσεις πάνω σε διαφορετικά περιβάλλοντα εργασίας και με τυχαία σειρά λειτουργιών εισαγωγής, διαγραφής κοκ

Λόγω των ιδιαίτερων χαρακτηριστικών των δίσκων SSD, τα συστήματα και οι εφαρμογές που βασίζονται στους δίσκους HDD και εφαρμόζονται εξ' ολοκλήρου σε αυτούς, δε μπορούν να επωφεληθούν πλήρως από τα θετικά χαρακτηριστικά τους. Επιπρόσθετα, οι συχνές αλλαγές του B-δέντρου, επιφέρουν μείωση της επίδοσης των SSD, μειώνοντας τη διάρκεια ζωής τους. Με μία λέξη τα B-δέντρα είναι πολύ δύσκολο να επιφέρουν τη μέγιστη απόδοση τους όταν εφαρμόζονται στους δίσκους SSD. Αποτελεί πλέον επιτακτική ανάγκη, η ανάπτυξη μιας δομής ευρετηρίου B-δέντρου βασισμένη σε δίσκους SSD.

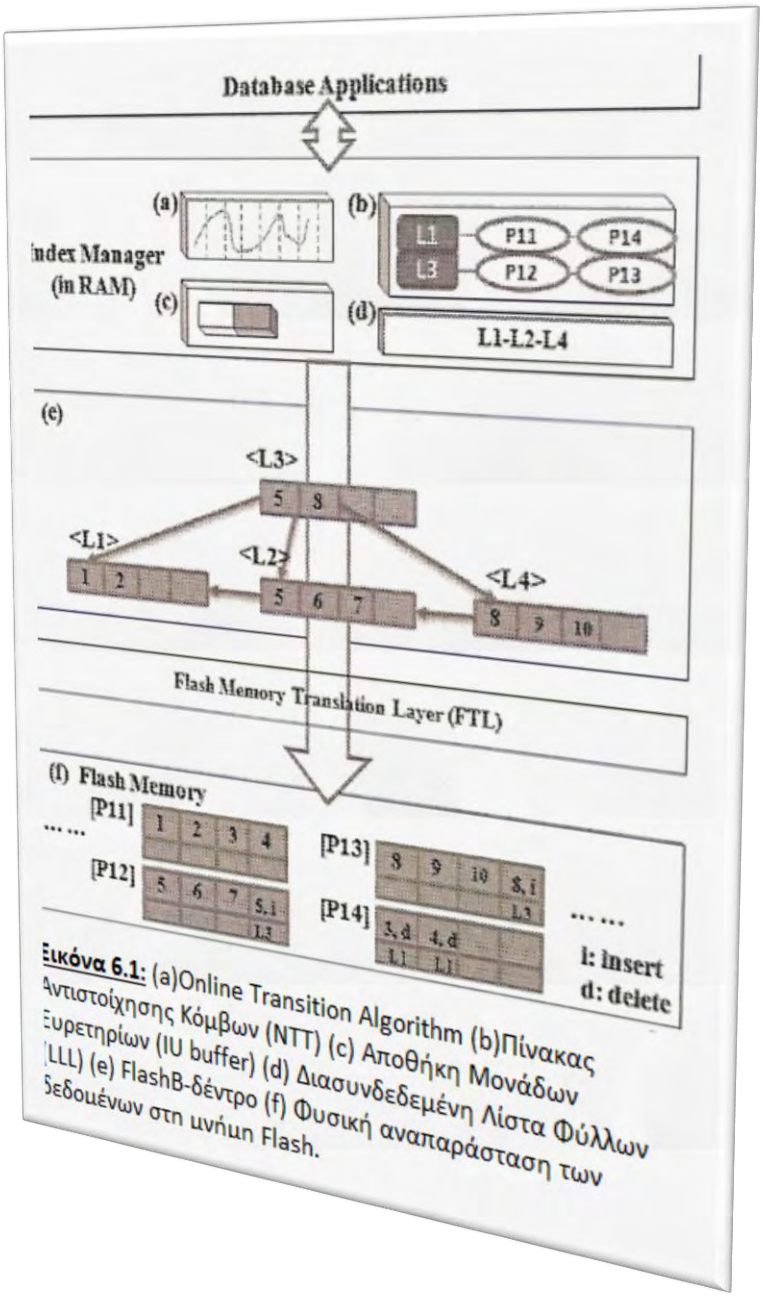
Πρώτη απόπειρα για τη λύση του προβλήματος αυτού, αποτελεί η πρόταση BFTL (κομμάτι του λογισμικού), για την αποτελεσματική χρήση του B-δέντρου στη μνήμη Flash. Ομαδοποιεί τις εγγραφές ευρετηρίου οι οποίες ανήκουν σε διαφορετικούς λογικούς κόμβους B-δέντρων σε μία φυσική σελίδα. Ως αποτέλεσμα, ο αριθμός των σελίδων που εγγράφονται/ανανεώνονται μειώνεται αισθητά.

Από την άλλη πλευρά, η πρόταση IBSF [2] (σε αντίθεση με το BFTL), εισάγει την συνηθισμένη προσέγγιση, η οποία απλά αποθηκεύει τις εγγραφές ευρετηρίου σε συνεχόμενες θέσεις στην φυσική περιοχή.

Οι λειτουργίες ανάγνωσης αποτελούν τον κυριότερο λόγο για τον οποίο μπορεί κάποιος να χρησιμοποιήσει αυτόν τον μηχανισμό.

Στην Εικόνα 6.1, φαίνεται η αρχιτεκτονική του συστήματος, αποτελούμενη από 2 βασικά στοιχεία: ***μια δομή FlashB-δέντρου*** (FlashB-tree Structure) που υλοποιεί τη βέλτιστη δομή ευρετηρίου και έναν ***Index Manager***, που υλοποιεί σχήματα διαχείρισης ευρετηρίων. Με τη σειρά του, στον διαχειριστή ευρετηρίων συμπεριλαμβάνονται: ένας ***online αλγόριθμος μετάβασης*** (Online Transition Algorithm – OTA), ένας ***πίνακας αντιστοίχησης κόμβων*** (Node Mapping Table - NMT), ένας ***Index Unit Buffer – IUB*** και μια ***λίστα συνένωσης φύλλων*** (Linked Leaf List - LLL).

Πιο αναλυτικά ο Index Unit Manager μπορεί να αποθηκεύσει δεδομένα, ισοδύναμα μιας σελίδας. Όπως και στην περίπτωση των FlashDB (Παράγραφος 3.1), έτσι και στα FlashB-δέντρα οι καταστάσεις των κόμβων επιλέγονται δυναμικά. Κάθε κόμβος οργανώνεται ανεξάρτητα, βάσει του φόρτου εργασίας που του έχει ανατεθεί. Στην Εικόνα 6.1 (b), παρουσιάζονται κάποιοι κόμβοι σε BFTL, ενώ κάποιοι άλλοι σε κατάσταση IBSF. Στην Εικόνα 6.1 (f), παρουσιάζεται η φυσική αναπαράσταση των δεδομένων που είναι αποθηκευμένα στη μνήμη Flash. Όλες οι εγγραφές κλειδιών των κόμβων L2 και L4, είναι αποθηκευμένες στους τομείς P12 και P13 αντίστοιχα. Ενώ οι κόμβοι L1 και L3 βρίσκονται σε BFTL, οι αντίστοιχες εγγραφές ημερολογίου είναι διασκορπισμένες σε διάφορους τομείς. Στην Εικόνα 6.1 (e) παρουσιάζεται ένα FlashB-δέντρο, στο οποίο όπως μπορεί κανείς να διακρίνει, σε κάθε κόμβο-φύλλο, ο πρώτος δείκτης ‘’δείχνει’’ στον κόμβο-φύλλο από αριστερά (ή σε NULL στην περίπτωση απουσίας του τελευταίου).Το FlashB-δέντρο λοιπόν, αποθηκεύει στη μνήμη RAM όλες τις δυνατές πληροφορίες των διπλανών κόμβων του κόμβου-γονέα. Οι κόμβοι-φύλλα του B^{+}-δέντρου, είναι συνδεδεμένοι προκειμένου να παρέχουν ταξινομημένη πρόσβαση στην αναζήτηση κλειδιών. Όπως διακρίνεται και από την Εικόνα 6.1 (e), οι κόμβοι-φύλλα ενός FlashB-δέντρου, είναι συνδεδεμένοι αντίστροφα. Συνεπώς, η λίστα συνένωσης φύλλων (LLL) σχεδιάστηκε για να διατηρήσει σε τάξη την πληροφορία των κόμβων φύλλων, προκειμένου να ελαττωθεί στο μέγιστο ο χρόνος σειριακής αναζήτησης.



Εικόνα 6.1: (a)Online Transition Algorithm (b)Πίνακας Αντιστοίχησης Κόμβων (NTT) (c) Αποθήκη Μονάδων Ευρετηρίων (IU buffer) (d) Διασυνδεδεμένη Λίστα Φύλλων (LLL) (e) FlashB-δέντρο (f) Φυσική αναπαράσταση των δεδομένων στη μνήμη Flash.

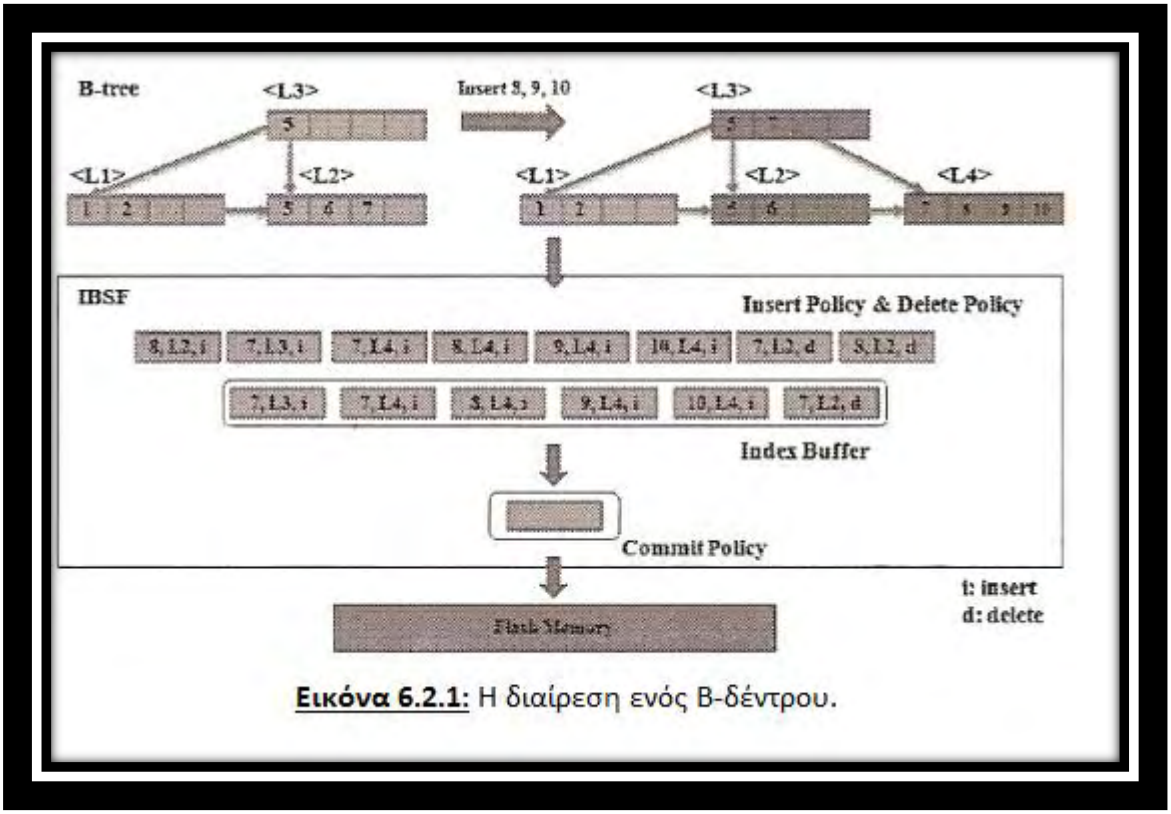
[1]

6.2 Αρχές Σχεδίασης

Η κύρια αιτία της δημιουργίας των FlashB-δέντρων, είναι η αποφυγή του υψηλού κόστους των λειτουργιών διάσπασης, συγχώνευσης και περιστροφής. Προκειμένου να επιτευχθεί αυτός ο στόχος, υϊοθετούνται 2 στρατηγικές:

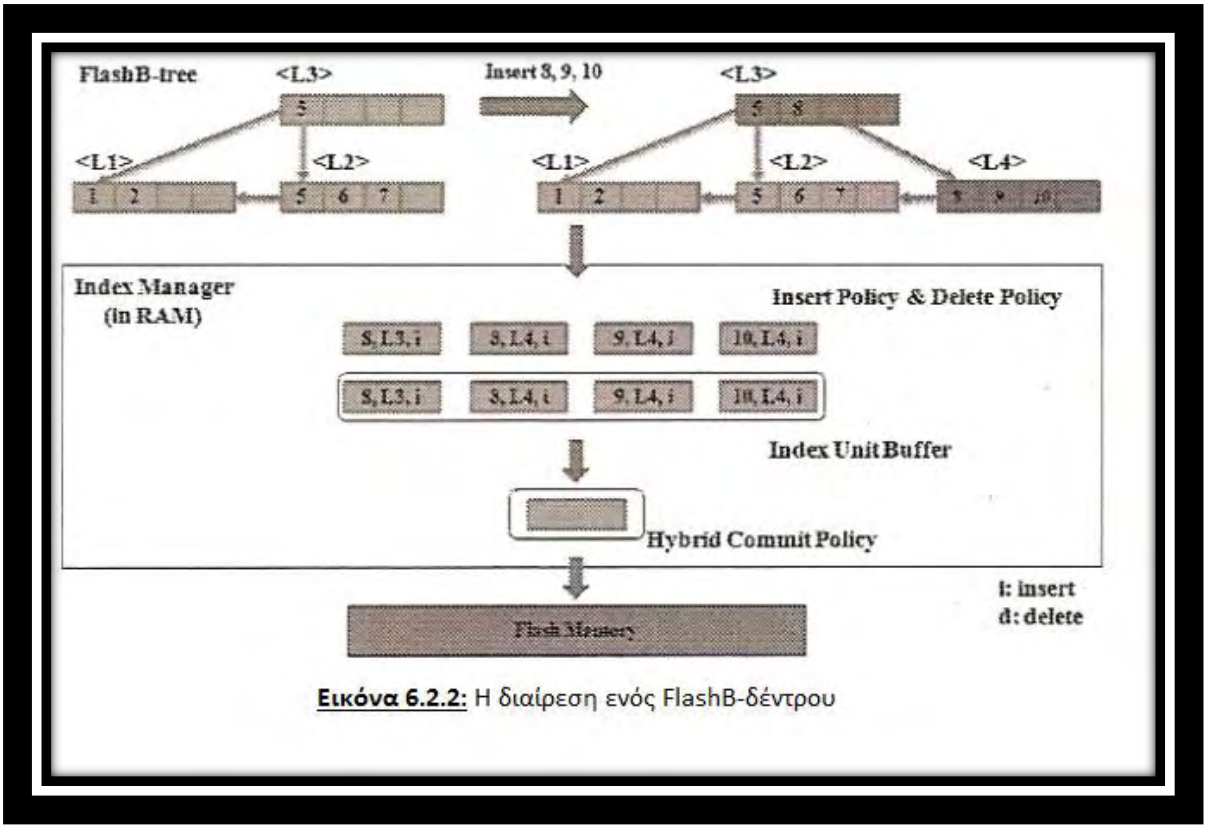
6.2.1 Πολιτική Διαίρεσης

Στην εφαρμογή των B-δέντρων (Εικόνα 6.2.1), μετά τη διαίρεση, κάθε κόμβος έχει κλειδιά αντίστοιχα, του μισού της τάξης του δέντρου. Μια ανανέωση ενός κόμβου-παιδιού, απαιτεί και ανανέωση του κόμβου-γονέα καθώς και του κόμβου-αδελφού, διαδικασία που εκτείνεται έως τη ρίζα του δέντρου. Η πραγματοποίηση αυτού του βήματος, χειροτερεύει όταν δουλεύουμε με δίσκους SSD. Εισάγεται λοιπόν η *Modify-Two-Node* στρατηγική, πάνω στη διάσπαση των FlashB-δέντρων. Στόχος αυτής της στρατηγικής είναι να τροποποιούνται οι κόμβοι (παιδί και δεξιός αδελφός), χωρίς να παραβιάζεται ο αριστερός κόμβος-αδελφός. Στην περίπτωση δεξιόπλευρης αύξησης, το FlashB-δέντρο, θα καταναλώσει πολύ λιγότερες νέες σελίδες και θα εγγράψει/ανανεώσει πολύ λιγότερους κόμβους. Στις Εικόνες 6.2.1 και 6.2.1.1 γίνεται μια σύγκριση ανάμεσα στα 2 σενάρια διάσπασης (της μεθόδου διαίρεσης που χρησιμοποιούσαμε έως τώρα και της Modify-Two-Node στρατηγικής διάσπασης).



Εικόνα 6.2.1: Η διαίρεση ενός B-δέντρου.

[1]



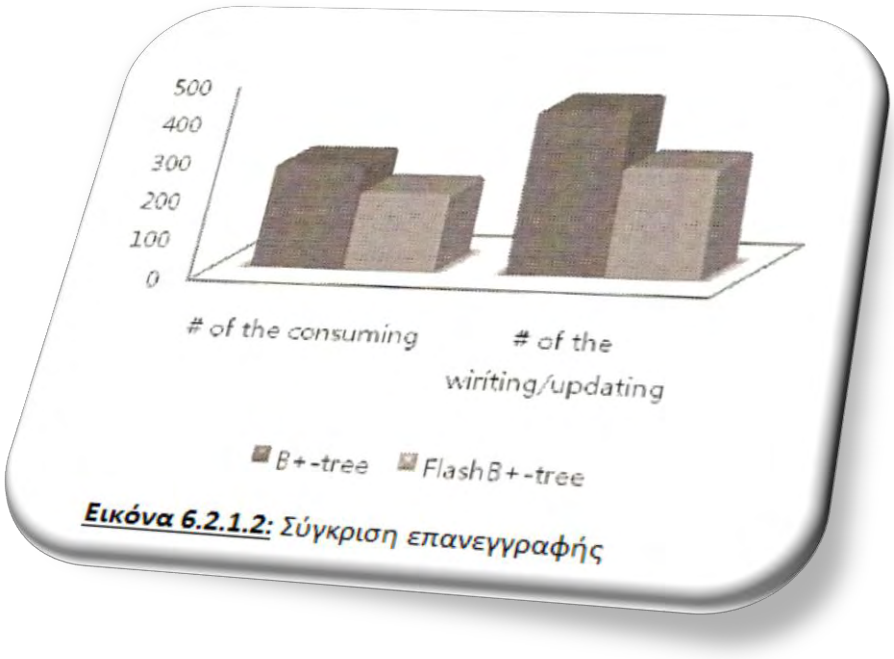
Εικόνα 6.2.2: Η διαίρεση ενός FlashB-δέντρου

[1]

Όπως φαίνεται στην Εικόνα 6.2.1, η IBSF έχει μια περιοχή προσωρινής αποθήκευσης στη μνήμη RAM για τις νέο-δημιουργούμενες αιτήσεις, οι οποίες αντιμετωπίζονται ως μονάδες ευρετηρίου (Index Units-IU). Στο παράδειγμα που παρουσιάζεται, όταν η τιμή 10 εισάγεται στο B-δέντρο μετά τις τιμές 8 και 9, δημιουργούνται οι [8,L2,i][7,L3,i][7,L4,i][8,L4,i][9,L4,i][10,L4,i][7,L2,d][8,L2,d] μονάδες ευρετηρίου (Index Units-IU), όπου [n,L_n,OPT] συμβολίζουν ότι η τιμή κλειδιού n πρέπει να εισαχθεί/διαγραφεί στον/από τον λογικό κόμβο L_n του B-δέντρου. Σύμφωνα με την πολιτική εισαγωγής/διαγραφής, η IBSF εξαλείφει τις πλεονάζουσες μονάδες ευρετηρίου και αποθηκεύει τις συμπαγείς του είδους τους στην περιοχή προσωρινής αποθήκευσης. Για παράδειγμα, η IBSF θεωρεί τις [8,L2,i] και [8,L4,d] μονάδες ευρετηρίου περιττές. Όταν η περιοχή προσωρινής αποθήκευσης γεμίσει, η IBSF αναλαμβάνει να ‘’τρέξει’’ τις μονάδες ευρετηρίου στον δίσκο SSD. Διαλέγει την πρώτη μονάδα ευρετηρίου από την περιοχή προσωρινής αποθήκευσης και έπειτα συλλέγει όλες τις μονάδες ευρετηρίου που σχετίζονται με οποιονδήποτε τρόπο με την πρώτη.

Πιο αναλυτικά, όταν οι τιμές 8,9 και 10 εισάγονται, το δέντρο υφίσταται διάσπαση κατά την εισαγωγή του 9. Η δομή ευρετηρίου IBSF, χρειάζεται 3 εγγραφές στον δίσκο SSD, αφού τροποποιεί τους κόμβους L2,L3 και L4.

Στην περίπτωση εισαγωγής τριών νέων τιμών στο FlashB-δέντρο, δημιουργούνται τέσσερις μονάδες ευρετηρίου (IU's). Σύμφωνα με την *Modify-Two-Node* πολιτική, εάν κατά τη διάρκεια εισαγωγής κλειδιού το δέντρο υποστεί διάσπαση, τότε θα χρειαστούν 2 λειτουργίες εγγραφής αφού αποφεύγεται η τροποποίηση του αριστερού κόμβου, διαδικασία που αναλύεται στην Εικόνα 6.2.1.2.



Εικόνα 6.2.1.2: Σύγκριση επανεγγραφής

[1]

6.2.2 Πολιτική Συνένωσης/Περιστροφής

Η λειτουργία διαγραφής αφαιρεί την εγγραφή κλειδιού από τον κόμβο φύλλο και έπειτα πραγματοποιούνται οι απαραίτητες αλλαγές προκειμένου να μην αλλάξουν οι ιδιότητες του B-δέντρου. Κατά τη σχεδίαση των B-δέντρων, στην περίπτωση όπου ο αριθμός των κλειδιών πέσει κάτω από το μισό της τάξης του δέντρου, χρειάζεται ανακατανομή.

Το FlashB-δέντρο, χρησιμοποιεί την ***Lazy-Coalescence*** στρατηγική με στόχο την διατήρηση της λειτουργίας συγχώνευσης/περιστροφής στο ελάχιστο δυνατόν. Όταν ο αριθμός των κλειδιών σε έναν κόμβο δεν έχει ξεπεραστεί κατά το μισό της διάταξης του δέντρου, τότε δε χρειάζεται η διαδικασία συγχώνευσης μεταξύ των 2 κόμβων, ακόμα και αν ο ένας έχει γίνει πολύ μικρός μετά τη διαγραφή. Επιπλέον, δεν υπάρχει περιστροφή μεταξύ 2 κόμβων στην περίπτωση όπου ο ένας έχει γίνει πολύ μικρός μετά τη διαγραφή αλλά ο γείτονας του είναι πολύ μεγάλος ώστε να προσαρτηθούν σε αυτόν οι εγγραφές του πρώτου (χωρίς δηλαδή να υπερβεί το μέγιστο αριθμό κλειδιών).

6.3 Σχεδίαση του Online Transition Algorithm – OTA

Όπως έχει αναφερθεί πολλάκις, ούτε το IBSF αλλά ούτε και το BFTL δε μπορούν να χρησιμοποιηθούν παράλληλα με όλες τις συνθήκες εργασίας. Σε αυτό το σημείο λοιπόν, περιγράφεται ο αλγόριθμος online μεταφοράς (online transition algorithm) των FlashB-δέντρων για τους δίσκους SSD, με σκοπό να προσαρμόσει τις 2 παραπάνω μεθόδους σε μία που να δουλεύει γ κάθε φόρτο εργασίας.

Για καλύτερη κατανόηση, ορίζονται σε αυτό το σημείο κάποιες παράμετροι που χρησιμοποιούνται παρακάτω:

Παράμετροι	**Περιγραφή**
$\mathbf{I}$	Πρόκειται για μια συσσωρευμένη ποσότητα που βασίζεται στη φόρτωση λειτουργιών εγγραφής/ανάγνωσης.
$\mathbf{I_C}$	Το κόστος για την αλλαγή από Non-Clustered Mode σε Clustered Mode.
$\mathbf{I_N}$	Το κόστος για την αλλαγή από Clustered Mode σε Non-Clustered Mode.
$\mathbf{C_R}$	Το κόστος λειτουργίας ανάγνωσης

C_W	Το κόστος λειτουργίας εγγραφής.

Το σύστημα θα έπρεπε να δουλεύει με εντολές όπως ‘’set clustered/non-clustered’’ για επιλογή κατάστασης. Αλλά σε αυτή την σχεδίαση που παρουσιάζεται, θα εστιάσουμε στο πώς το σύστημα θα αποφασίζει την καλύτερη κατάσταση βάσει του εκάστοτε φόρτου εργασίας.

Ας υποθέσουμε ότι η δομή ευρετηρίου χρησιμοποιεί αρχικά Clustered Mode. Τότε, η τιμή της συσσωρευμένης ποσότητας I_{t0} είναι I_C , με την περιοχή προσωρινής αποθήκευσης (Reservation Buffer - RB) και τον πίνακα αντιστοίχησης κόμβων (Node Mapping Table - NMT) να μην έχουν τεθεί ακόμη σε εφαρμογή στην κατάσταση αυτή.

Συμβολίζουμε το κόστος λειτουργίας εγγραφής με C_W και το κόστος λειτουργίας ανάγνωσης με C_R. Οι τιμές διαμορφώνονται ως εξής:

- Όταν πραγματοποιείται λειτουργία ανάγνωσης → $I_{t'} = I_t - C_R$
- Όταν πραγματοποιείται λειτουργία εγγραφής → $I_{t'} = I_t + C_W$

Εφόσον το I_t φράσσεται από το το I_C και από το I_N ($I_C < I_t < I_N$), όταν το $I_{t'}$ ισούται με το I_N ($I_{t'} = I_N$), το σύστημα μετατρέπεται σε Non-Clustered Mode και δεν αλλάζει κατάσταση προτού μειωθεί η τιμή του $I_{t'}$. Σε αυτό το σημείο είναι που τίθενται σε εφαρμογή ο πίνακας αντιστοίχησης κόμβων (NMT) και η περιοχή προσωρινής αποθήκευσης (RB). Όταν η τιμή του $I_{t'}$ μειωθεί και γίνει ίση με το το I_C το σύστημα μετατρέπεται ξανά στην κατάσταση Clustered Mode.

6.4 Αξιολόγηση Επίδοσης

Για την αξιολόγηση της επίδοσης των FlashB-δέντρων, εφαρμόστηκαν και εξετάστηκαν σε τρία διαφορετικά περιβάλλοντα εργασίας, προκειμένου να πιστοποιηθεί η πρακτικότητα της εφαρμογής τους.

Πιο συγκεκριμένα, η επίδοση των λειτουργιών αναζήτησης/διαγραφής/εισαγωγής μπορεί να αντικατοπτριστεί στον αριθμό των input/output λειτουργιών στη μνήμη Flash. Μια λειτουργία διαγραφής κατατάσσεται στις λειτουργίες εγγραφής, διότι η διαδικασία που ακολουθείται είναι το κλειδί της αντίστοιχης εγγραφής να μαρκάρεται ως μη αποτελεσματικό (ατελέσφορο). Ο αριθμός των input/output λειτουργιών μιας εισαγωγής είναι μεγαλύτερος ή ίσος από μια λειτουργία εγγραφής και αντίστοιχα ο αριθμός των input/output μιας ανανέωσης κλειδιού ή δείκτη (pointer) είναι μεγαλύτερος ή ίσος του αριθμού των λειτουργιών εγγραφής. Και ορίζεται υψίστης σημασίας μια λειτουργία εισαγωγής ή ανανέωσης, αφού μπορεί να προκαλέσει διάσπαση του δέντρου και η διάσπαση με τη σειρά της να προκαλέσει λειτουργίες εγγραφής και διαγραφής.

Το κόστος ανάγνωσης n εισαγόμενων καταγραφών ορίζεται ως:

$$\begin{cases} B-tree(R) = n * h * Cost(R) \\ BFTL(R) = n * \sum_{i=1}^{h} Ti * Cost(R) \\ IBSF(R) = n * h * Cost(R) \\ \frac{h'}{h} IBSF(R) < FlashB(R) < BFTL(R) \end{cases}$$

Σχέση 6.4 [1]

Όπου η παράμετρος O_{read} συμβολίζει το κόστος λειτουργίας ανάγνωσης στους δίσκους SSD, η παράμετρος h συμβολίζει το ύψος του FlashB-δέντρου και τέλος η παράμετρος T_X υποδηλώνει τον αριθμό των σελίδων που περιέχουν μονάδες ευρετηρίου για τον κόμβο x.

Στην κλασσική σχεδίαση του B-δέντρου οι λειτουργίες διαίρεσης, συγχώνευσης και περιστροφής απαιτούν 3 λειτουργίες εγγραφής, αφού σε κάθε περίπτωση τροποποιούν τον κόμβο-γονέα και τους 2 κόμβους-αδέλφια (γειτονικοί κόμβοι). Στα FlashB-δέντρα όμως, όταν καλούνται οι 3 παραπάνω λειτουργίες, λόγω της Modify-Two-Node στρατηγικής, διασφαλίζεται ο ελάχιστος αριθμός λειτουργιών εγγραφής.

Εάν ο αριθμός των λειτουργιών αυτών είναι y, τότε εισάγοντας n καταγραφές, ο συνολικός αριθμός λειτουργιών εγγραφής είναι:

$$\begin{cases} B-tree(W) = \{n + (y * 3)\} * Cost(W) \\ BFTL(W) = \frac{k}{l_{max}}\{n + (y * 3)\} * Cost(W) \\ IBSF(W) = \frac{1}{p}\{n + (y * 3)\} * Cost(W) \\ FlashB(W) = \{n + (y * 2)\} * Cost(W) \end{cases}$$

Σχέση 6.4.1 [1]

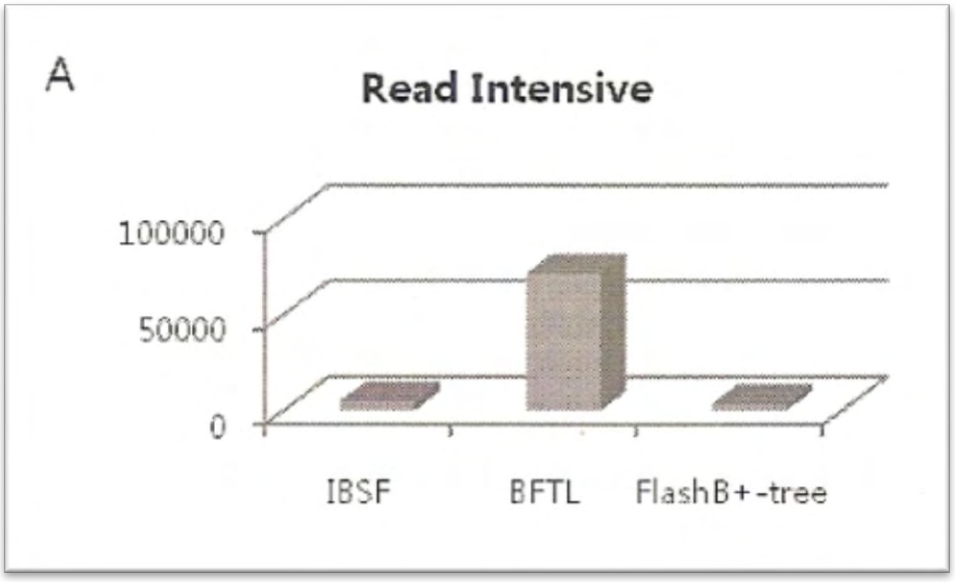
Όπου οι παράμετροι συμβολίζουν τα εξής:

- Η παράμετρος O_{write} συμβολίζει το κόστος της λειτουργίας ανάγνωσης στους δίσκους SSD.
- Η παράμετρος I_{max} συμβολίζει τον μέγιστο αριθμό μονάδων ευρετηρίων στην περιοχή προσωρινής αποθήκευσης.
- Και τέλος η παράμετρος k συμβολίζει τον αριθμό των σελίδων ενός B-δέντρου.

Τα απαιτούμενα δεδομένα κρατούνται προσωρινά στην περιοχή προσωρινής αποθήκευσης έως ότου γεμίσει. Τότε ομαδοποιούνται από την εκάστοτε πολιτική αποθήκευσης. Στην περίπτωση της δομής BFTL, εάν ο αριθμός των σελίδων είναι k και ο μέγιστος αριθμός των μονάδων ευρετηρίου στην περιοχή προσωρινής αποθήκευσης είναι I_{max} , τότε απαιτούνται k λειτουργίες εγγραφής οποτεδήποτε ‘’τρέξουν’’ στον δίσκο SSD I_{max} μονάδες ευρετηρίου. Όσον αφορά τη δομή IBSF, είναι προφανές ότι πολύ δύσκολα θα μειώσει τον αριθμό των λειτουργιών εγγραφής αφού θα ‘’τρέξει’’ στο δίσκο πολύ περισσότερες σελίδες απ’ οτι η BFTL.

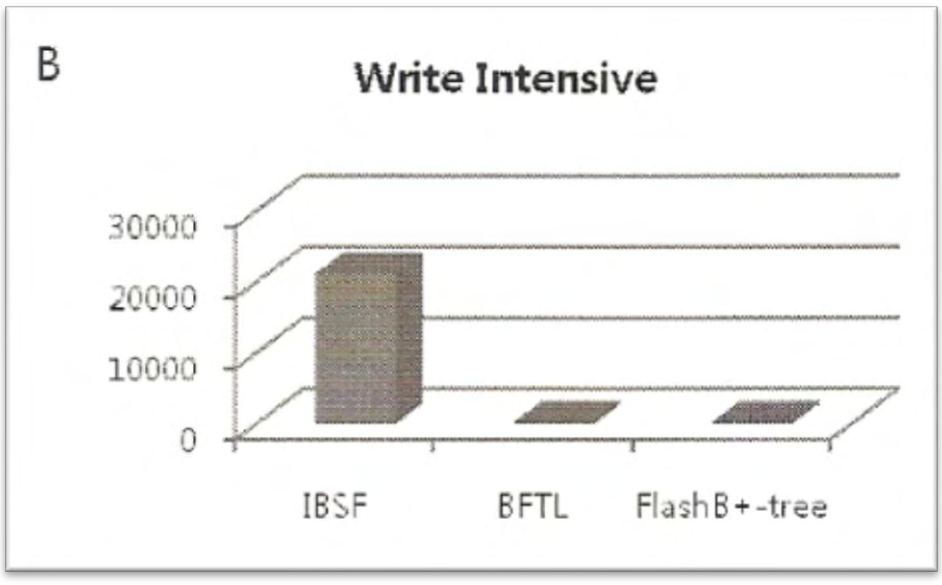
6.5 Αποτελέσματα Πειραμάτων

Στην Εικόνα 6.5 (Α) παρουσιάζεται η περίπτωση όπου το φόρτο εργασίας αποτελείται κυρίως από λειτουργίες ανάγνωσης. Δε διακρίνεται ξεκάθαρα η διαφορά μεταξύ του FlashB-δέντρου και της IBSF, ενώ είναι προφανές ότι η δομή BFTL έχει την χειρότερη επίδοση. Συνεπώς, FlashB-δέντρο και δομή IBSF προσφέρουν τα ίδια όταν πρόκειται για πλειοψηφία λειτουργιών, λειτουργίες ανάγνωσης.



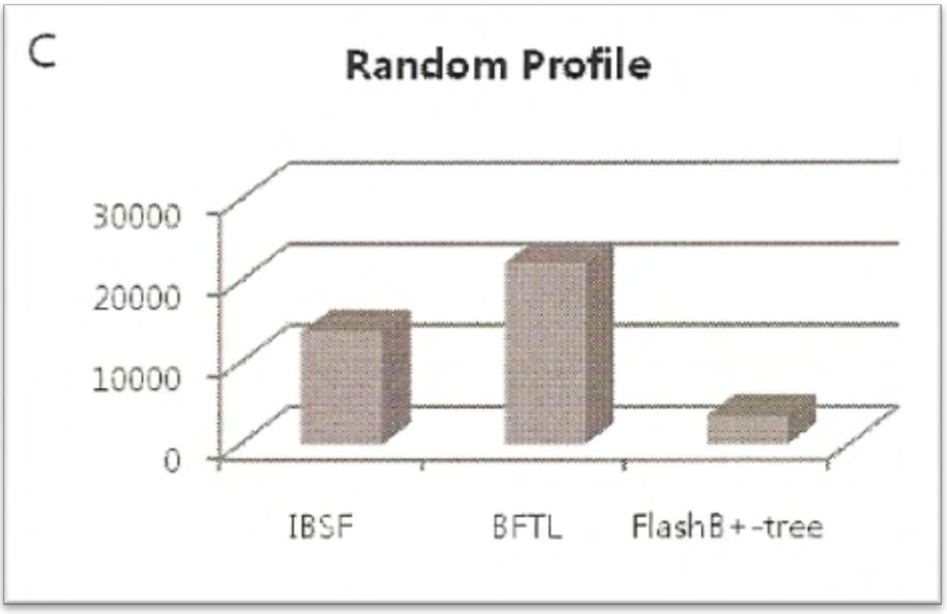
Εικόνα 6.5 (A) [1]

Στην Εικόνα 6.5 (B) φαίνονται τα αποτελέσματα για τις περιπτώσεις όπου το φόρτο εργασίας αποτελείται κυρίως από λειτουργίες εγγραφής. Είναι ξεκάθαρο, ότι το FlashB-δέντρο υπερτερεί έναντι των άλλων, απαιτώντας μικρότερο αριθμό διαιρέσεων και συγχωνεύσεων.



Εικόνα 6.5 (B) [1]

Στην Εικόνα 6.5 (C) παρουσιάζονται τα αποτελέσματα ενός τυχαίου προφίλ φόρτου εργασίας.



Εικόνα 6.5 (C) [1]

Βιβλιογραφία

[1] Jin, Rize, Se Jin Kwon, and Tae-Sun Chung. "FlashB-tree: a novel B-tree index scheme for solid state drives." *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. ACM, 2011.

[2] Hyun-Seob Lee, Dong-Ho Lee, ‘’An efficient index buffer management scheme for implementing a B-tree on NAND flash memory’’, Data Knowl. Eng. 69(9), pp. 901-916, 2010.

[3] Karl M. J. Lofgren, Robert D. Norman, Gregory B. Thelin, and Anil Gupta, ‘’Wear leveling technique for flash EEPROM systems’’, 2003, United States Patent, no.6,594,183.

ΚΕΦΑΛΑΙΟ 7

7.1 Εισαγωγή στα LA-δέντρα

Τα LA-δέντρα (*Lazy – Adaptive trees*) αποτελούν σήμερα μία καινοτομία στις δομές ευρετηρίου και δημιουργήθηκαν με σκοπό να βελτιώσουν την επίδοση των συστημάτων, ελαχιστοποιώντας όσο είναι εφικτό την πρόσβαση στη μνήμη Flash. Ένα LA-δέντρο ‘’ανταλλάσσει’’ το κόστος ανάγνωσης/εγγραφής κόμβων, εκτελώντας τις λειτουργίες ανανέωσης βάσει των buffer που υπάρχουν σε κάθε επίπεδο. Επιπλέον με τη χρήση ενός online αλγορίθμου (βέλτιστα αποτελέσματα σε εκτέλεση σειριακής NAND μνήμης) προσαρμόζει δυναμικά το μέγεθος των buffer.

7.2 Σχεδιασμός ενός LA-δέντρου

Ο σχεδιασμός ενός LA-δέντρου, έχει τα ακόλουθα χαρακτηριστικά:

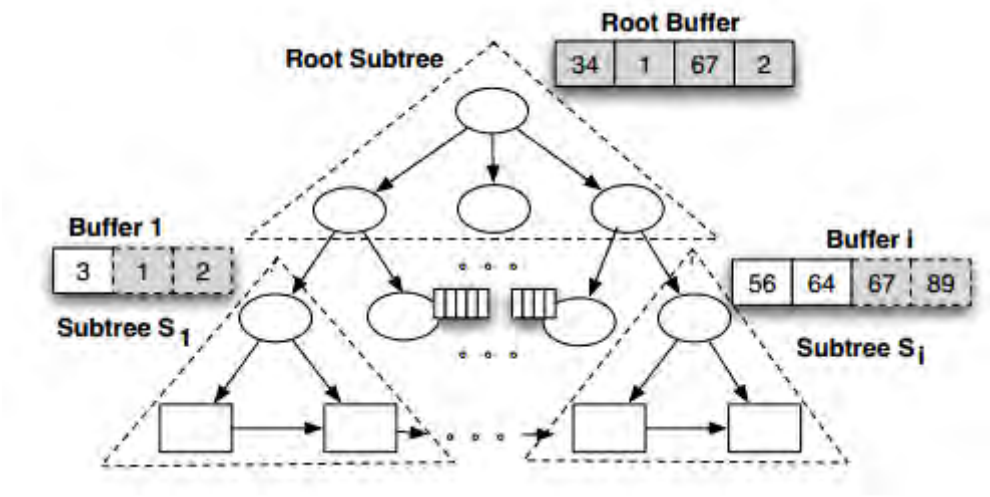
7.2.1 *Cascaded Buffers*

Ένα LA-δέντρο προσαρμόζει buffers στους κόμβους των διάφορων επιπέδων του δέντρου. Κάθε buffer περιέχει λειτουργίες ανανέωσης που πρέπει να εφαρμοστούν στους κόμβους του επιπέδου και στους ‘’απογόνους’’ αυτών. Την κατάλληλη στιγμή, όλα τα στοιχεία που υπάρχουν σε κάθε buffer στέλνονται σε ομαδοποιημένη μορφή στους buffer του επόμενου επιπέδου. Επομένως, σε ένα LA-δέντρο, όλες οι ανανεώσεις γίνονται ταυτόχρονα ενώ κατά τη διάρκεια τους, παρέχεται πρόσβαση από τον κόμβο-ρίζα μέχρι τους κόμβους-φύλλα (τελευταίοι κόμβοι του δέντρου). Με αυτόν τον τρόπο, βελτιστοποιεί και την ανάγνωση αλλά και την εγγραφή των κόμβων κατά τη διάρκεια των λειτουργιών ανανέωσης.

7.2.2 *Adaptive Buffering*

Ενώ μια τέτοια προσέγγιση (lazy-approach) είναι αποτελεσματική για τις λειτουργίες ανανέωσης, παραμένει μη αποτελεσματική για τις λειτουργίες αναζήτησης, όπου πρέπει να παραχθεί για τον χρήστη, μια άμεση απάντηση. Η εκτέλεση αυτών των άμεσης-απάντησης αναζητήσεων σε ένα lazy-δέντρο, προϋποθέτει σάρωση μεγάλης χωρητικότητας buffer που βρίσκονται στη μνήμη Flash. Όπως είναι εμφανές, κάτι τέτοιο είναι απαγορευτικά ακριβό. Το LA-δέντρο, διορθώνει αυτό το μειονέκτημα, χρησιμοποιώντας έναν online αλγόριθμο που ‘’έξυπνα’’ προσαρμόζει το μέγεθος του buffer βάσει του φόρτου εργασίας. [2,3]

Μία ειδοποιός διαφορά των LA-δέντρων από τα B^+-δέντρα, είναι ότι τα πρώτα προσκολλούν έναν buffer στους κόμβους κάθε K-ιοστού επιπέδου της διεύθυνσης αναζήτησης, ξεκινώντας από τη ρίζα. Ο buffer χρησιμοποιείται για να ομαδοποιεί τις λειτουργίες ανανέωσης (εισαγωγή και διαγραφή) που πρέπει να εφαρμοστούν στον κόμβο αυτόν και στους ‘’απογόνους’’ του. Στην Εικόνα 7.1 απεικονίζονται οι buffers που τοποθετούνται σε κάθε επίπεδο ξεκινώντας από τη ρίζα (K=2).



Εικόνα 7.2.2: Ένα LA-δέντρο, με κόμβους φύλλα, με υποδέντρα και με κλιμακωτούς buffer (cascaded buffers) σε κάθε αλληλοδιάδοχο επίπεδο. Τα σκιασμένα τετράγωνα, δείχνουν το αποτέλεσμα αδειάσματος του buffer του κόμβου-ρίζα, στους buffers των επόμενων επιπέδων. [1]

Σε ένα LA-δέντρο, ο όρος υποδέντρο (subtree) χρησιμοποιείται για να αναπαραστήσει ένα κομμάτι του δέντρου το οποίο ξεκινά με ένα προσκολλημένο σε κάποιον κόμβο buffer και τελειώνει πάνω από του επόμενου επιπέδου προσκολλημένους στους κόμβους buffers, ώστε το υποδέντρο να έχει <u>ακριβώς</u> K επίπεδα. Χρησιμοποιείται επίσης η μεταβλητή B_i για να συμβολίσει το μέγεθος του buffer (μετράται σε εγγραφές) του i-οστού υποδέντρου. Το B_i δεν μπορεί να υπερβαίνει το προκαθορισμένο όριο U (επιλέγεται να είναι ένα κομμάτι της μνήμης M) αν και συχνά έχει μικρότερη τιμή αφού καθορίζεται δυναμικά βάσει του τωρινού φόρτου εργασίας που φαίνεται από το υποδέντρο. Στον πίνακα 1 συνοψίζονται οι επεξηγήσεις των συμβόλων που χρησιμοποιούνται παρακάτω.

Parameter	Symbol	Value
Memory size	M	given by application
Node size (in num. of elements)	**F**	variable
Num. of elements in the tree	N	variable
Height of the tree	H	$\approx \lceil \log_F \frac{N}{F} \rceil + 1$
Subtree height for buffering	**K**	variable
Effective buffer size (in num. of elements)	B_i	$\leq U < M$, variable, subtree-specific

Πίνακας 1 [1]

7.3 Περιγραφή των βασικών λειτουργιών σε ένα LA-δέντρο

7.3.1 Λειτουργίες Εισαγωγής και Διαγραφής

Μία λειτουργία ανανέωσης σε ένα LA-δέντρο πραγματοποιείται με καθυστέρηση: η νέα καταχώριση προς εισαγωγή ή διαγραφή τοποθετείται στον buffer του κόμβου-ρίζα. Τότε η AppendToBuffer συνάρτηση, ελέγχει αν ο buffer χρειάζεται να αδειάσει καλώντας τον αλγόριθμο ADAPT. Εάν η απόφαση είναι ναι (γεγονός που συμβαίνει σπάνια), με το άδειασμα του buffer μεταφέρονται όλες οι καταχωρίσεις, σε ομαδοποιημένη μορφή, στους buffers των κατώτερων επιπέδων του δέντρου. Η διαδικασία αυτή μπορεί να συμβαίνει αναδρομικά έως ότου οι καταχωρίσεις τελικά φτάσουν στους κόμβους-φύλλα και περιγράφεται στην Εικόνα 7.3.1

(a) Routine Insert(R, E)

Input: Root of the tree R, Entry E = (k: key value, *rid*: record id)

1. Let BUF = buffer of the root node, invoke AppendToBuffer(BUF, E, i).

(b) Routine Delete(R, E)

Input: Root of the tree R, Entry E = (k: key value, *rid*: record id)

1. Let BUF = buffer of the root node, invoke AppendToBuffer(BUF, E, d).

Εικόνα 7.3.1 [1]

7.3.2 Λειτουργίες Αναζήτησης

Όταν ένα LA-δέντρο λάβει μια αίτηση αναζήτησης, διατρέχει όλο το δέντρο από πάνω προς τα κάτω (Εικόνα 7.3.2). Εν αντιθέσει με τα B^+-δέντρα, αν ένας κόμβος που βρίσκεται στο μονοπάτι κόμβος-ρίζα –> κόμβος-φύλλο, έχει έναν προσκολλημένο buffer, το LA-δέντρο σαρώνει επίσης και τον buffer για ανανεώσεις οι οποίες ακόμη δεν έχουν φτάσει στους κόμβους-φύλλα.

(c) Routine Search(R, q)

Input : Root of the tree R, Predicate q
Output: Set of entries A that satisfy q

1. Top down search in the tree. For each non-leaf node on the path, if the node has a buffer BUF, invoke SearchBuffer(BUF, q) $\rightarrow A$.
2. At the leaf node N, invoke SearchNode(N) $\rightarrow A$.
3. If q is a range predicate, for each next leaf node N, SearchNode(N) $\rightarrow A$, for each buffer BUF at an ancestor node of N, if BUF has not been scanned, SearchBuffer(BUF, q) $\rightarrow A$.
4. If A contains deletion entries, apply deletions to collapse content of A.

Εικόνα 7.3.2 [1]

Ένα μοναδικό χαρακτηριστικό των LA-δέντρων είναι ότι κατά τη διάρκεια σάρωσης των buffer (Εικόνα 7.3.2.1), ο αλγόριθμος ADAPT ελέγχει εάν το άδειασμα αυτών θα προσφέρει βελτίωση της επίδοσης, βάσει του μέχρι τώρα φόρτου εργασίας. Στην περίπτωση που το άδειασμα του buffer κριθεί ωφέλιμο, όλες οι καταχωρίσεις του, μεταφέρονται στο επόμενο επίπεδο. Αλλιώς, γίνεται σάρωση του buffer μήπως βρεθούν αποτελέσματα στην αναζήτηση. Αυτή η διαδικασία επαναλαμβάνεται έως ότου εξεταστούν οι κόμβοι-φύλλα, όπου το πιθανότερο είναι εκεί να βρίσκονται τα αποτελέσματα των αναζητήσεων.

(b) Routine SearchBuffer(BUF, q)

Input: Buffer BUF, Predicate q
Output: Set of entries A that satisfy q

1. If ADAPT decides to empty the buffer, invoke EmptyBuffer(BUF) and piggyback the search for q. Else, directly search BUF for q. Return result.

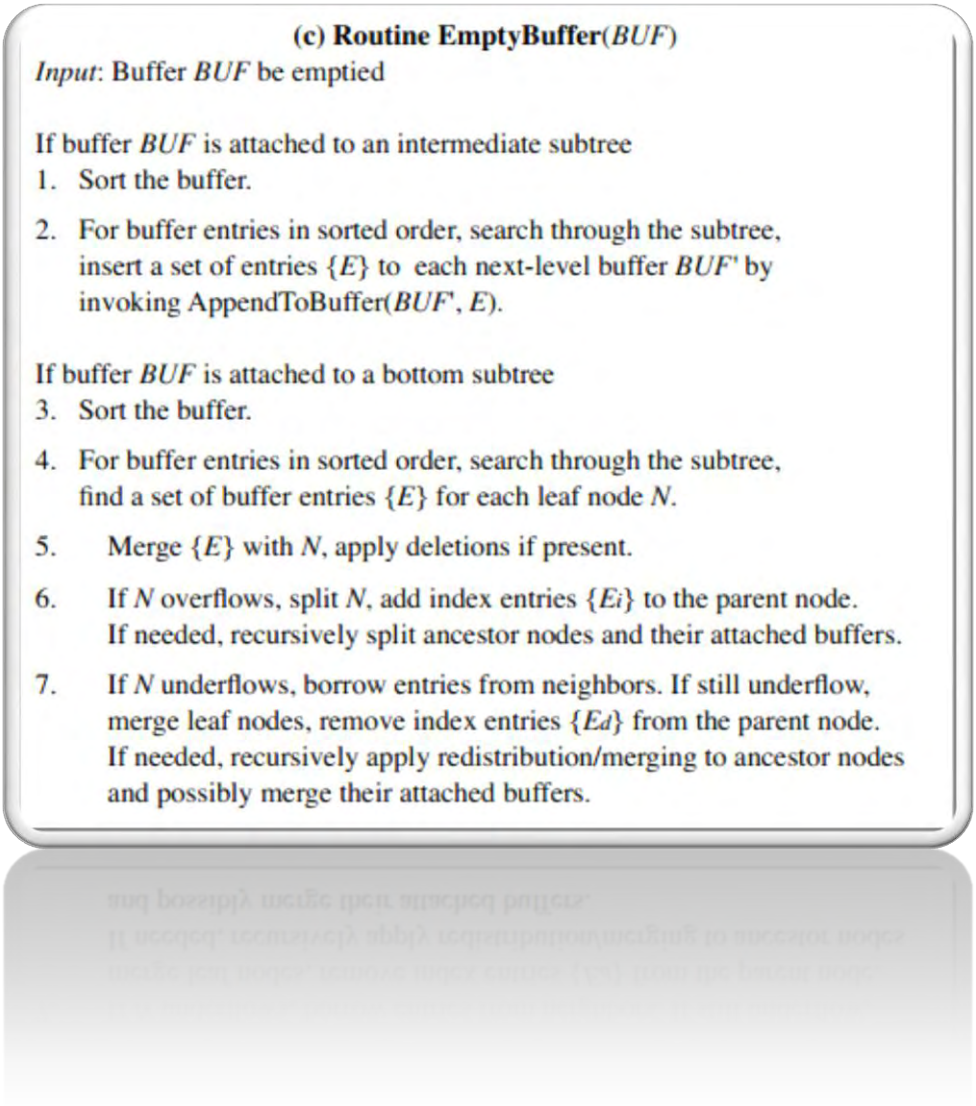
Εικόνα 7.3.2.1 [1]

Για ένα εύρος αναζητήσεων, το LA-δέντρο μπορεί διαδοχικά να αποκτήσει πρόσβαση και σε άλλους κόμβους-φύλλα. Κατά τη διάρκεια των σαρώσεων κάθε κόμβου-φύλλου, το LA-δέντρο επίσης πραγματοποιεί

αναζητήσεις και στους buffer που είναι προσκολλημένοι στους ‘’προγόνους’’ αυτών, στην περίπτωση που αυτοί οι κόμβοι-φύλλα (‘’πρόγονοι’’) δεν έχουν εξεταστεί στο παρελθόν. Το γεγονός αυτό αποτελεί ακόμη μία διαφορά μεταξύ των LA-δέντρων και των B^+-δέντρων. Όμως, το κόστος αναζήτησης των επιπρόσθετων αυτών buffer, είναι περιορισμένο. Πιο αναλυτικά, οι επαναλαμβανόμενες σαρώσεις των buffer αυτών σύντομα θα έχουν ως αποτέλεσμα, ο αλγόριθμος ADAPT να αποφασίσει το ‘’άδειασμα’’ τους.

7.4 Λειτουργίες εκκαθάρισης των buffer

Ο ADAPT είναι ένας online αλγόριθμος που αποτελεί τμήμα της κεντρικής υπηρεσίας πληροφοριών (central intelligence) των LA-δέντρων. Λαμβάνει ως είσοδο μία ακολουθία αιτήσεων από ανανεώσεις και αναζητήσεις για κάθε buffer και ελέγχει το μέγεθός του, αποφασίζοντας πότε πρέπει να τον αδειάσει. Οι αιτήσεις ανανέωσης, δεν πυροδοτούν το άδειασμα ενός buffer, εκτός και αν ο buffer ξεπεράσει το προκαθορισμένο όριο U. Για τις αναζητήσεις, ο ADAPT καταγράφει το κόστος σαρώσεων των buffer αλλά και το κατ’ εκτίμηση κόστος αδειάσματος αυτών. Έπειτα, υπολογίζει αφενός σε εκείνο το σημείο, το ενός-χρόνου κόστος (one-time cost) άδειασμα του buffer και αφετέρου τα κέρδη που οι αναζητήσεις θα επωφεληθούν, εξαιτίας του αδειάσματος. Εάν τα κέρδη ξεπερνούν το κόστος αδειάσματος, τότε ο αλγόριθμος ADAPT αποφασίζει να αδειάσει το buffer.



(c) Routine EmptyBuffer(*BUF*)

Input: Buffer *BUF* be emptied

If buffer *BUF* is attached to an intermediate subtree

1. Sort the buffer.
2. For buffer entries in sorted order, search through the subtree, insert a set of entries $\{E\}$ to each next-level buffer *BUF'* by invoking AppendToBuffer(*BUF'*, *E*).

If buffer *BUF* is attached to a bottom subtree

3. Sort the buffer.
4. For buffer entries in sorted order, search through the subtree, find a set of buffer entries $\{E\}$ for each leaf node *N*.
5. Merge $\{E\}$ with *N*, apply deletions if present.
6. If *N* overflows, split *N*, add index entries $\{E_i\}$ to the parent node. If needed, recursively split ancestor nodes and their attached buffers.
7. If *N* underflows, borrow entries from neighbors. If still underflow, merge leaf nodes, remove index entries $\{E_d\}$ from the parent node. If needed, recursively apply redistribution/merging to ancestor nodes and possibly merge their attached buffers.

Εικόνα 7.4 [1]

Ο αλγόριθμος ADAPT προκειμένου να αποφασίσει άδειασμα ή όχι ενός buffer, ‘’καλεί’’ την ρουτίνα αδειάσματος buffer (Routine EmptyBuffer). Το άδειασμα ενός buffer σε ένα ενδιάμεσο υποδέντρο, αποτελείται από τα εξής βήματα (βήματα 1-2, Εικόνα 7.4): ταξινομεί τα δεδομένα του buffer και διανέμει όλες τις καταχωρίσεις του, σε ταξινομημένη σειρά, στους buffers των επόμενων επιπέδων, αφού ολοκληρώσει την αναζήτηση του υποδέντρου. Για παράδειγμα, στην Εικόνα 1 απεικονίζεται η διαδικασία αδειάσματος buffer του κόμβου-ρίζα. Ο buffer αρχικά διαβάζεται στη μνήμη, ταξινομείται και έπειτα τα κλειδιά του διανέμονται στους buffers των επόμενων επιπέδων.

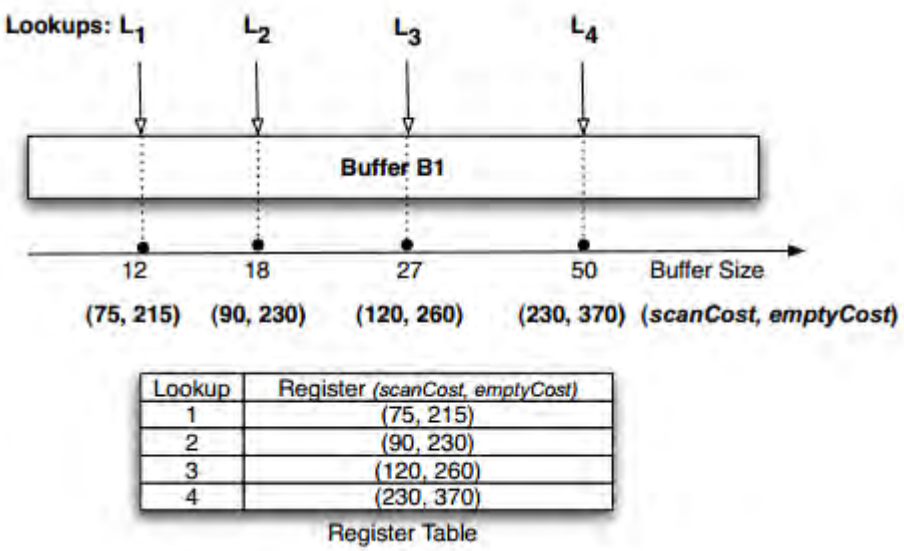
Το άδειασμα ενός buffer σε ένα υποδέντρο κόμβων-φύλλων (βήματα 3-7, Εικόνα 7.4) επίσης ξεκινά με την ταξινόμηση των δεδομένων του buffer. Έπειτα οι καταχωρίσεις του buffer διανέμονται σε ταξινομημένη σειρά στους κόμβους-φύλλα. Οι κόμβοι-φύλλα επεξεργάζονται ένας-ένας κάθε φορά, από τα αριστερά προς τα δεξιά: για κάθε κόμβο-φύλλο N, οι λαμβανόμενες καταχωρίσεις του buffer συγχωνεύονται με το N εκτελώντας ότι οι καταχωρίσεις αυτές καθορίζουν (εισαγωγές, διαγραφές). Εάν υπάρξει υπερχείλιση του N, τότε διαιρείται σε δύο ή περισσότερους κόμβους. Μια τέτοια διαίρεση μπορεί να διευρυνθεί και στους κόμβους-προγόνους (οι μη μηδενικής χωρητικότητας buffers των κόμβων αυτών διαιρούνται επίσης, αντίστοιχα). Λόγω των εισαγωγών/διαγραφών, το N μπορεί να μείνει χωρίς καμία εγγραφή. Σε αυτήν την περίπτωση όμως, το N αρχικά δανείζεται κάποιες καταχωρίσεις από κάποιον γειτονικό κόμβο. Έπειτα και εάν χρειαστεί συγχωνεύεται με κάποιον γειτονικό κόμβο, ακριβώς όπως γίνεται και με τα B^+-δέντρα. Στις σπάνιες αυτές περιπτώσεις συγχωνεύσεων, η προσαρμογή των κόμβων-γονέων και των κόμβων-προγόνων επιτυγχάνεται όπως στα B^+-δέντρα με τη διαφορά ότι επιτυγχάνεται και συγχώνευση των αντίστοιχων buffer.

Τέλος, είναι σημαντικό να αναφερθεί ότι το μονοπάτι από τον κόμβο-ρίζα έως τους κόμβους-φύλλα, που ακολουθείται σε κάθε λειτουργία ανανέωσης σε ένα LA-δέντρο, είναι όμοιο με αυτό ενός B^+-δέντρο. Η μόνη διαφορά τώρα, είναι ο αυξανόμενος ‘’χρόνος διαδρομής’’ (travel time), εξαιτίας της σάρωσης των buffer στους ενδιάμεσους κόμβους κατά μήκος του μονοπατιού.

7.5 Ο Αλγόριθμος ADAPT

Ο αλγόριθμος ADAPT αποφασίζει με δυναμικό τρόπο το μέγεθος των buffer βάσει του θεωρητικού φόρτου εργασίας. Τέτοιες αποφάσεις, λαμβάνονται για κάθε buffer ξεχωριστά, αφού διαφορετικά υποδέντρα μπορεί να έχουν διαφορετικό φόρτο εργασίας, ανάλογα με την εκάστοτε κατανομή κλειδιών. Καθώς μια ακολουθία αιτήσεων από ανανεώσεις και αναζητήσεις λαμβάνεται στον buffer, ο αλγόριθμος ADAPT αποφασίζει αν θα αδειάσει τον buffer ώστε το κόστος ολόκληρης της ακολουθίας των λειτουργιών, να ελαχιστοποιηθεί. Ακολουθεί στη συνέχεια, μια σύντομη περιγραφή της κεντρικής ιδέας του αλγορίθμου ADAPT, με τη βοήθεια της Εικόνας 7.5.

Στην Εικόνα 7.5, διακρίνεται ένας buffer που επεξεργάζεται μια ακολουθία από λειτουργίες αναζήτησης. Κάθε αναζήτηση, η οποία συμβολίζεται με L_i, ''βλέπει'' ένα συγκεκριμένο μέγεθος buffer b_i και επομένως ''πληρώνει'' ένα συγκεκριμένο κόστος σάρωσης του buffer, s_i.



Lookup	Register *(scanCost, emptyCost)*
1	(75, 215)
2	(90, 230)
3	(120, 260)
4	(230, 370)

Register Table

Εικόνα 7.5: Για την ακολουθία των τεσσάρων αναζητήσεων του παραδείγματος, ο αλγόριθμος ADAPT αδειάζει τον buffer στην τέταρτη αναζήτηση, L4. [1]

Για παράδειγμα, το κόστος σάρωσης της αναζήτησης L_1 είναι 75 ενώ της αναζήτησης L_2 είναι 90. Για κάθε L_i, ο αλγόριθμος ADAPT αποφασίζει εκείνη τη στιγμή αν θα αδειάσει τον buffer ή όχι. Βασίζει την απόφαση του στο κατ' εκτίμηση κόστος αδειάσματος του buffer σε κάθε αναζήτηση (συμβολίζεται με e_i και αυξάνεται με το μέγεθος του buffer s_i). Για παράδειγμα, το κατ' εκτίμηση κόστος αδειάσματος στην αναζήτηση L1 είναι 215 και στην αναζήτηση L2 είναι 230.

Πώς λοιπόν, θα υπολόγιζε κανείς το όφελος (αν υπάρχει) ενός πιθανού αδειάσματος του buffer στην αναζήτηση L1 της Εικόνας 4?

Αρχικά, κάθε μία από τις τρείς αναζητήσεις που ακολουθούν την L1, αποθηκεύουν την τιμή s_1. Επομένως το όφελος αδειάσματος του buffer στην

αναζήτηση L1, το οποίο συμβολίζεται με p_1, δίνεται από τη σχέση $p_1 = 3*s_1 = 225$. Το όφελος αυτό, ξεπερνά το κατ' εκτίμηση κόστος αδειάσματος b_1 (= 215) με αποτέλεσμα να είναι προτιμότερο το άδειασμα του buffer στην αναζήτηση L_1. Γενικεύοντας, θα ήταν πάντα ωφελιμότερο να αδειάζει ο buffer στην αναζήτηση L_i, αν το μελλοντικό p_i, ξεπερνά το κατ' εκτίμηση κόστος αδειάσματος e_i. Παρόλα αυτά, εφόσον ο ADAPT είναι ένας online αλγόριθμος, δε μπορεί να έχει πληροφορία για το μελλοντικό φόρτο εργασίας και επομένως δε μπορεί να προβλέψει το μελλοντικό p_i.

Συνεπώς, η διαδικασία που ακολουθείται είναι η εξής: έστω η αναζήτηση L_j. Οι εκ των προτέρων υπολογισμοί μιας προηγούμενης αναζήτησης L_i ($i < j$) είναι οι πιθανοί υπολογισμοί που παρήχθησαν κατά το άδειασμα του buffer στην αναζήτηση Li. Υπάρχουν (j-i) αναζητήσεις μεταξύ Li και Lj και καθεμία από αυτές αποθηκεύει το κόστος σάρωσης του buffer s_i. Επομένως, οι εκ των προτέρων υπολογισμοί των Li, οι οποίοι συμβολίζονται με sav(i,j) δίνονται από τον τύπο: $sav(i,j) = (j-i)*s_i$. Ο αλγόριθμος ADAPT αδειάζει τον buffer στην αναζήτηση Lj στην περίπτωση όπου οι υπολογισμοί εκ των προτέρων οποιασδήποτε προηγούμενης αναζήτησης Li, ξεπερνά το αντίστοιχο κατ' εκτίμηση κόστος αδειάσματος e_i. Με λίγα λόγια, ο αλγόριθμος ADAPT αδειάζει τον buffer στην αναζήτηση Lj εάν $\exists_{i<j} sav(i,j) > e_i$. Προκειμένου να το υπολογίσει αυτό, ο αλγόριθμος ADAPT διατηρεί μια δομή ''book-keeping'' τη λεγόμενη Register R_i για κάθε μία από τις παλιότερες αναζητήσεις Li. Όπως είναι εμφανές και στην Εικόνα 4, το R_i έχει δύο πεδία: το πεδίο scanCost που αποθηκεύει το κόστος σάρωσης s_i και το πεδίο emptyCost που αποθηκεύει το κατ' εκτίμηση κόστος αδειάσματος e_i.

Συνεπώς σε αυτό το παράδειγμα (Εικόνα 7.5), ο αλγόριθμος ADAPT πρώτα αδειάζει τον buffer στην αναζήτηση L4, αφού οι εκ των προτέρων υπολογισμοί της αναζήτησης L1 (που δίνονται από τον τύπο sav(1,4) = (4-1)*R1.scanCost = 225), ξεπερνούν το αντίστοιχο κόστος αδειάσματος του buffer, R1.emptyCost = 215.

Ο ολοκληρωμένος αλγόριθμος ADAPT είναι ο εξής:

Algorithm 1 ADAPT(operationType)

```
1: if (operationType = lookup) then
2:    j ← j + 1 {j is the lookup index}
3:    {Check if any old lookup triggers emptying}
4:    for all lookups L_i such that (i < j) do
5:       savings ← (j − i) · R_i.scanCost {sav(i, j)}
6:       if (savings > R_i.emptyCost) then
7:          Return EMPTY
8:       end if
9:    end for
10:   Create new R_j = (emptyCost, scanCost)
11: else if (bufferSize > U) then
12:    Return EMPTY
13: end if
14: Return NOT_EMPTY
```

[1]

Για τον χειρισμό των λειτουργιών αναζήτησης, ο αλγόριθμος ADAPT εκτελεί δύο βήματα (σειρές 2-10): Αρχικά ελέγχει εάν οι εκ των προτέρων υπολογισμοί οποιασδήποτε παλιότερης αναζήτησης Li υπερβαίνει το κατ' εκτίμηση κόστος αδειάσματος του buffer. Έπειτα, ο αλγόριθμος ADAPT δημιουργεί μια νέα καταχώριση (register) για να αποθηκεύσει το κατ' εκτίμηση κόστος αδειάσματος και το κόστος σάρωσης για την αναζήτηση που βρίσκεται σε εξέλιξη. Η διαχείριση των λειτουργιών ανανέωσης είναι πολύ πιο εύκολη: Ο αλγόριθμος ADAPT αδειάζει τον buffer όταν υπερβεί το προκαθορισμένο όριο U (σειρά 11).

Βιβλιογραφία:

[1]http://people.cs.umass.edu/~dganesan/papers/VLDB09-LATree.pdf

[2] L. Arge, K. Hinrichs et al. Efficient bulk operations on dynamic R-trees. In Algorithmica 33(1):104-128, 2002.

[3] A. Birrell, M. Isard et al. A Design for High Performance Flash Disks. In SIGOPS Operating system review, 41(2), 2007.

[4] L. Arge. The buffer tree: A Technique for Designing Batched External Data Structures. In Algorithmica 37(1):1-24, 2003.

[5] http://nvmw.ucsd.edu/2010/documents/Diao_Yanlei.pdf