Δυναμικες Δομες Δεικτοδοτησης Κειμενων για Συστηματα Αποθηκευσης Δεδομενων

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Γεώργιο Μαργαρίτη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Μάρτιος 2008

Αφιερωση

Στους αγαπημένους μου γονείς.

Ευχαριστιες

Πρωτίστως θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου κ. Στέργιο Αναστασιάδη. Τον ευχαριστώ ιδιαίτερα για τις πολύτιμες συμβουλές που μου έδωσε και για την αμέριστη στήριξη και βοήθεια στην προσπάθειά μου να ολοκληρώσω τη διατριβή μου, καθώς και για την οικονομική υποστήριξη της παρούσας έρευνας. Χωρίς κάποια από τα προηγούμενα, η περάτωση της παρούσας εργασίας θα ήταν αρκετά δυσκολότερη.

Επίσης θα ήθελα να ευχαριστήσω την οικογένειά μου για την ηθική, ψυχολογική και οικονομική υποστήριξη που μου παρείχαν όλα αυτά τα χρόνια. Όντας τόσα χρόνια μακρυά τους, ελάχιστα από αυτά μπόρεσα να τους ανταποδώσω.

Τέλος θα ήθελα να ευχαριστήσω όλους τους φίλους και γνωστούς, για τις επιστημονικές και "μη-επιστημονικές" συζητήσεις και αναλύσεις που κάναμε όλα αυτά τα χρόνια, και οι οποίες βοήθησαν να περάσουν αυτά τα χρόνια ευχάριστα.

Περιεχομενα

Ευρετηριο Σχηματων

Ευρετηριο Πινακων

Περιληψη

Γεώργιος Μαργαρίτης του Δημητρίου και της Παναγιώτας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Μάρτιος 2008. Δυναμικές Δομές Δεικτοδότησης Κειμένων για Συστήματα Αποθήκευσης Δεδομένων

Επιβλέποντας: Στέργιος Β. Αναστασιάδης.

Στην παρούσα εργασία μελετούμε το πρόβλημα της δημιουργίας ενός συστήματος δεικτοδότησης για δυναμικές συλλογές κειμένων, όπου συνεχώς προστίθενται νέα κείμενα ή τροποποιούνται τα υπάρχοντα. Στόχος μας είναι η αποδοτική υποστήριξη αναζήτησης αρχείων σε συστήματα αποθήκευσης δεδομένων. Βασικό χαρακτηριστικό τέτοιων συστημάτων είναι η ύπαρξη ενός ευρετηρίου το οποίο πρέπει να ενημερώνεται συνεχώς προκειμένου να μπορεί να χρησιμοποιηθεί για την αναζήτηση κειμένων που περιέχουν συγκεκριμένες λέξεις. Προκειμένου να επιτύχουν καλή απόδοση αναζήτησης τα υπάρχοντα συστήματα βασίζονται σε πολύπλοκες δομές δεικτοδότησης που είναι αποθηκευμένες συνεχόμενες στο δίσκο.

Στην εργασία μας προτείνουμε ένα νέο σύστημα δεικτοδότησης που χρησιμοποιεί μπλοκς προεπιλεγμένου μεγέθους και μια νέα μέθοδο επιλεκτικής ενημέρωσης της δομής σχεδιασμένη για πιο αποδοτική προσπέλαση των συσκευών αποθήκευσης. Η επιλογή αυτή επιτρέπει το σχεδιασμό απλούστερων αλγορίθμων ενημέρωσης του ευρετηρίου.

Υλοποιώντας την προτεινόμενη αρχιτεκτονική και εκτελώντας έναν αριθμό πειραμάτων με μεγάλες συλλογές κειμένων της τάξεως των 100Gb συμπεραίνουμε πως το σύστημά μας παρέχει χρόνους δεικτοδότησης και αναζήτησης κειμένων συγκρίσιμους με γνωστά συστήματα δεικτοδότησης.

Extended Abstract in English

Margaritis, George. MSc , Computer Science Department, University of Ioannina, Greece.
March, 2008. Dynamic Inverted Files for Online Index Maintanance.

Thesis Supervisor: Stergios V. Anastasiadis.

Dropping prices in storage space capacity combined with a constant increase in the accumulated amount of digitized content create an unprecedented need for full-text search automation in most types of text collections. For example, in addition to traditioncal article archives and hyperlinked documents over the web, we observe an enormous necessity for text search automation in highly dynamic storage environments such as desktop and distributed file systems. Existing content search technology was originally developed for the needs of the information retrieval applications. Thus, the early availability of news articles in digital text format provided the basis for designing systems that can store and index large amounts of text repositories. Both the storage management and the indexing methods used in those early systems were mainly customized for the needs of append-only updates.

Justified from the need for fast query processing, the indexing structure that maps each searched term to the documents that contain the term is assumed to be contiguously stored on the disk. To a large extent, this fundamental decision dominates the structure of the entire search engine by enforcing the indexing information to be constantly relocated to guarantee contiguity as new articles are added to the system [7]. In fact, alternative approaches that organize indexing information in blocks were not previously adequately evaluated, since they restricted the block size to a few tens of kilobytes [26, 2]. Practically, the complexity of the text indexing issue combined with the large number of configuration parameters keep it a challenging problem to design architectures that successfully support both searches and updates online in an efficient way.

We develop an inverted-file management architecture that is amenable to continuous updates without significant relocation overhead. We follow the well-known approach of

separating postings lists into short and long depending on their total number of postings in memory or on disk. We use fixed-size blocks for storing both short and long postings lists on disk with the only difference that blocks of long lists are used exclusively by a single term. As is confirmed from our experiments, the amount of time spent for flushing short lists to disk is factors higher in comparison to the flushing time for long lists. We use this observation to favor flushing of long postings lists and thus improve the efficiency of disk accesses in the Proteus algorithm that we propose. Based on our ideas, we developed an index construction prototype that we compare with a public open-source search engine. Using a standard dataset, we find that appropriate choice of the block size and other configuration parameters can lead to index building and posting list retrieval performance that is comparable to that of highly optimized indexing architectures.

Κεφαλαιο 1

Εισαγωγη

1.1 Δεικτοδότηση συλλογών κειμένων

Η μείωση των τιμών των συσκευών αποθήκευσης, σε συνδυασμό με μία σταθερή αύξηση του μεγέθους των δεδομένων που είναι συγκεντρωμένα σε ψηφιακή μορφή, δημιουργούν μία πρωτοφανή ανάγκη για υποστήριξη αναζήτησης αρχείων βάσει του περιεχομένου τους (full-text search). Για παράδειγμα, παρατηρούμε μία αυξημένη ανάγκη για υποστήριξη αναζήτησης αρχείων σε αρκετά δυναμικά περιβάλλοντα, όπως οι επιτραπέζιοι υπολογιστές ή τα κατανεμημένα συστήματα αρχείων.

Η αναζήτηση κειμένων βασίζεται στη χρήση ενός ευρετηρίου το οποίο στην απλούστερη περίπτωση αντιστοιχίζει κάθε όρο στα κείμενα τα οποία τον περιέχουν. Η πιο αποτελεσματική δομή για την αποτίμηση ερωτημάτων αναζήτησης είναι το *ανεστραμμένο αρχείο* [29]: μία συλλογή από λίστες, μία για κάθε όρο, οι οποίες περιέχουν όλα τα κείμενα στα οποία εμφανίζεται ο εκάστοτε όρος. Οι λίστες αυτές καλούνται *ανεστραμμένες λίστες*.

Η υπάρχουσα τεχνολογία αναζήτησης αναπτύχθηκε αρχικά για την υποστήριξη συλλογών σε βιβλιοθήκες και σε περιβάλλοντα τα οποία χαρακτηρίζονταν από έναν χαμηλό ρυθμό ενημέρωσης των συλλογών κειμένων. Τόσο η διαχείριση του αποθηκευτικού χώρου όσο

και οι μέθοδοι δεικτοδότησης που είχαν αναπτυχθεί υποστήριζαν κυρίως την δεικτοδότηση στατικών συλλογών.

Σε δυναμικές συλλογές κειμένων οι οποίες ενημερώνονται συνεχώς (νέα κείμενα προστίθενται ή υπάρχοντα κείμενα τροποποιούνται/διαγράφονται), όπως στην περίπτωση του διαδικτύου, η διατήρηση του ευρετηρίου σε συνεπή κατάσταση με την συλλογή κειμένων είναι ένα αρκετά δύσκολο εγχείρημα, καθώς η ενημέρωση του ευρετηρίου απαιτεί πρωτίστως το σύστημα αναζήτησης να είναι πλήρως λειτουργικό καθώς νέα κείμενα προστίθενται ή τροποποιούνται, ενώ παράλληλα οι χρόνοι αναζήτησης πρέπει να διατηρούνται χαμηλοί.

Βασιζόμενοι στην ανάγκη για γρήγορη αναζήτηση κειμένων, κάθε ανεστραμμένη λίστα η οποία περιέχει τα κείμενα στα οποία εμφανίζεται ένας όρος απαιτείται να είναι αποθηκευμένη *συνεχόμενα* στο δίσκο. Σε ένα πολύ μεγάλο βαθμό, η απαίτηση αυτή καθορίζει την δομή και την οργάνωση του συστήματος αναζήτησης, αναγκάζοντας τις ανεστραμμένες λίστες να μετακινούνται συνεχώς σε νέες θέσεις στο δίσκο καθώς προστίθενται νέα κείμενα προκειμένου να είναι εγγυημένη η φυσική συνέχεια των λιστών αυτών στο δίσκο [7].

Καθώς το μέγεθος των συλλογών κειμένων γίνεται πάρα πολύ μεγάλο, γίνεται εξαιρετικά δύσκολο να διατηρούμε τις ανεστραμμένες λίστες συνεχόμενα αποθηκευμένες στο δίσκο. Για τον σκοπό αυτό έχουν προταθεί διάφορες μέθοδοι [26, 2, 13, 5, 7] οι οποίες, με σκοπό την βελτίωση της απόδοσης δεικτοδότησης, διατηρούν τις ανεστραμμένες λίστες κατακερματισμένες σε διάφορα τμήμα στο δίσκο.

Η πολυπλοκότητα ενός συστήματος δεικτοδότησης δυναμικών συλλογών κειμένων, σε συνδυασμό με το μέγεθος των συλλογών και τον αριθμό των παραμέτρων που πρέπει να ληφθούν υπόψιν, μετατρέπουν θέμα της μελέτης και σχεδίασης μιας κατάλληλης αρχιτεκτονικής που να υποστηρίζει αποδοτικά τόσο την δεικτοδότηση κειμένων όσο και την αναζήτηση κειμένων σε ένα απαιτητικό και δύσκολο πρόβλημα.

1.2 Αντικείμενο μεταπτυχιακής εργασίας

Στην παρούσα εργασία θεωρούμε το πρόβλημα της δημιουργίας ενός συστήματος δεικτοδότησης δυναμικών συλλογών κειμένων. Μελετάμε κυρίως το πρόβλημα της διατήρησης του ευρετηρίου συνεχώς ενημερωμένο, καθώς νέα κείμενα προστίθενται στη συλλογή. Για το σκοπό αυτό, προτείνουμε μία αρχιτεκτονική για την δημιουργία και ενημέρωση του ανεστραμμένου αρχείου, η οποία υποστηρίζει την αποδοτική ενημέρωση του ευρετηρίου σε συνδυασμό με την διατήρηση των χρόνων αναζήτησης κειμένων σε χαμηλά επίπεδα.

Το σύστημα δεικτοδότησης που προτείνουμε βασίζεται στο διαχωρισμό των ανεστραμμένων λιστών ανάλογα με το μέγεθος τους σε *μικρές* και *μεγάλες* και στη χρήση διαφορετικών αλγορίθμων ενημέρωσης για κάθε κατηγορία. Τόσο οι μικρές όσο και οι μεγάλες λίστες αποθηκεύονται σε μπλοκς προεπιλεγμένου μεγέθους, με τη μόνη διαφορά πως αποθηκεύουμε πολλές μικρές λίστες σε ένα μπλοκ, ενώ κάθε μεγάλη λίστα αποθηκεύεται σε έναν αριθμό από αποκλειστικά δικά της μπλοκς. Ανά πάσα στιγμή διατηρούμε την πληροφορία του ποιες λίστες βρίσκονται αποθηκευμένες σε κάθε μπλοκ του συστήματος και πόσος ελεύθερος χώρος υπάρχει σε αυτό. Χρησιμοποιώντας τις πληροφορίες αυτές και τον προτεινόμενο αλγόριθμο συγχώνευσης, κάθε φορά που εξαντλείται η μνήμη επιλέγουμε να συγχωνεύσουμε με το δίσκο ένα υποσύνολο των λιστών που εξάχθηκαν από τα νέα κείμενα και βρίσκονται στη μνήμη, και συγκεκριμένα επιλέγουμε εκείνες τις λίστες οι οποίες προσφέρουν το μέγιστο κέρδος με κριτήριο την ελαχιστοποίηση των μετακινήσεων του δίσκου και την απελευθέρωση του μεγαλύτερου ποσοστού μνήμης.

1.3 Διάρθρωση μεταπτυχιακής εργασίας

Στο Κεφάλαιο 1 γίνεται μία σύντομη εισαγωγή στο πρόβλημα και τις δυσκολίες που ενέχει η δημιουργία και διατήρηση ενός ευρετηρίου για δυναμικές συλλογές. Στο Κεφάλαιο 2 αναφέρεται η βασική ορολογία καθώς και ο τρόπος με τον οποίον αποτιμάται ένα ερώτημα αναζήτησης δεδομένου ενός ευρετηρίου, ενώ γίνεται αναφορά στις μεθόδους συμπίεσης του ευρετηρίου. Στο Κεφάλαιο 3 αναφέρονται οι υπάρχουσες μέθοδοι δεικτοδότησης συλλογών κειμένων. Στο Κεφάλαιο 4 περιγράφεται το σύστημα δεικτοδότησης που προτείνουμε. Η αρχιτεκτονική του συστήματος καθώς και οι διάφορες δομές που χρησιμοποιεί αναλύονται, ενώ περιγράφονται οι αλγόριθμοι που χρησιμοποιούνται για την δημιουργία και ενημέρωση του ευρετηρίου. Στο Κεφάλαιο 5 περιγράφεται η υλοποίηση του συστήματος μας. Αρχικά περιγράφουμε το σύστημα το οποίο χρησιμοποιήσαμε για την υλοποίηση, ενώ στη συνέχεια αναφέρονται διάφορα θέματα υλοποίησης, καθώς και κάποιες τεχνικές και δομές για την αποδοτική υλοποίηση των αλγορίθμων της μεθόδου. Στο Κεφάλαιο 6 παρουσιάζονται και αναλύονται τα πειραματικά αποτελέσματα που αφορούν την απόδοση του συστήματος κατά την δεικτοδότηση και αναζήτηση κειμένων. Τέλος, στο Κεφάλαιο 7 συνοψίζονται τα συμπεράσματα και παρουσιάζονται μελλοντικές επεκτάσεις της παρούσας έρευνας.

Κεφαλαιο 2

Θεωρητικο υποβαθρο

2.1 Βασική ορολογία

Δεδομένης μιας συλλογής κειμένων, συχνά υπάρχει η ανάγκη εύρεσης όλων των κειμένων που περιέχουν κάποιες συγκεκριμένες λέξεις. Η αποτίμηση τέτοιων ερωτημάτων αναζήτησης βασίζεται στη χρήση ενός *ευρετηρίου*, μιας δομής δεδομένων η οποία στην απλούστερη της μορφή αντιστοιχίζει κάθε όρο t στα κείμενα τα οποία περιέχουν τον t. Όπως προκύπτει από διάφορες μελέτες [31], η πιο αποτελεσματική δομή για την αποτίμηση ερωτημάτων αναζήτησης είναι το *ανεστραμμένο αρχείο*: μία συλλογή από λίστες, μία για κάθε όρο, οι οποίες περιέχουν τα αναγνωριστικά των κειμένων που περιέχουν τον εκάστοτε όρο. Άλλες δομές που έχουν προταθεί, όπως τα *αρχεία υπογραφών* (signature files) και οι *πίνακες επιθεμάτων* (suffix arrays), δεν είναι τόσο αποδοτικές για γενικού σκοπού αποτίμηση ερωτημάτων.

Γενικά, ένα ανεστραμμένο ευρετήριο αποτελείται από δύο βασικές δομές, το *λεξικό* και το *ανεστραμμένο αρχείο*.

Το λεξικό είναι μία δομή αναζήτησης η οποία αντιστοιχεί κάθε όρο στην ανεστραμμένη του λίστα. Συγκεκριμένα, για κάθε όρο t διατηρούμε στο λεξικό:

- έναν μετρητή f_t που περιέχει το πλήθος των κειμένων που περιέχουν τον t (*συχνότητα εμφάνισης* του t)
- έναν δείκτη στην αρχή της ανεστραμμένης λίστας του t στο δίσκο

Μελέτες σχετικά με την αποτελεσματικότητα της ανάκτησης των λιστών συμπεραίνουν πως όλοι οι όροι θα πρέπει να δεικτοδοτούνται, ακόμα και οι αριθμοί. Ακόμα και οι πολύ συχνοί όροι, όπως "the", "of", "and", οι οποίοι φαίνονται μικρής σημασίας, έχουν ένα πολύ σημαντικό ρόλο στην αποτίμηση ερωτημάτων φράσεων, στα οποία θα αναφερθούμε στη συνέχεια.

Η δεύτερη βασική δομή ενός ανεστραμμένου ευρετηρίου είναι το ανεστραμμένο αρχείο, το οποίο αποτελείται από ένα σύνολο ανεστραμμένων λιστών. Για κάθε όρο t διατηρούμε στην αντίστοιχη λίστα του στο ανεστραμμένο αρχείο:

- τα αναγνωριστικά d_i των κειμένων που περιέχουν τον όρο t, τα οποία αναπαρίστανται σαν σειριακοί αριθμοί (μονοτονικά αυξανόμενοι ακέραιοι αριθμοί)
- για κάθε κείμενο d_i, την αντίστοιχη συχνότητα εμφάνισης $f_{d_i,t}$ του όρου t στο κείμενο d_i

Οι ανεστραμμένες λίστες αναπαρίστανται σαν ακολουθίες ζευγών $\langle d, f_{d,t} \rangle$ τα οποία καλούνται *εμφανίσεις* (postings), γι' αυτό και μία εναλλακτική ονομασία των ανεστραμμένων λιστών είναι *λίστες εμφανίσεων* (postings lists). Ένα ευρετήριο που περιέχει τέτοιας μορφής ανεστραμμένες λίστες καλείται ευρετήριο *επιπέδου κειμένων* (document-level index), λόγω του γεγονότος πως δεν διατηρεί πληροφορίες σχετικά με τις θέσεις στις οποίες εμφανίζεται ένας όρος μέσα στα κείμενα, παρά μόνο τα κείμενα στα οποία εμφανίζεται. Στην περίπτωση που οι ανεστραμμένες λίστες περιέχουν πλειάδες της μορφής $\langle d;\ f_{d,t};\ pos_1, pos_2, ..., pos_{f_{d,t}} \rangle$, όπου pos_i οι θέσεις εμφάνισης του όρου t μέσα στο κείμενο d, τότε το ευρετήριο καλείται ευρετήριο *επιπέδου λέξεων* (word-level index).

```
1)  The old night keeper keeps the keep in the town
2)  In the big old house in the big old gown
3)  The house in the town had the big old keep
4)  Where the old night keeper never did sleep
5)  The night keeper keeps the keep in the night
6)  And keeps in the dark and sleeps in the light
```

Σχήμα 2.1: Απλή συλλογή κειμένων. Αποτελείται από έξι κείμενα.

Μαζί με έναν πίνακα που περιέχει τα *βάρη* των κειμένων (στην απλούστερη περίπτωση, περιέχει για κάθε κείμενο το μέγεθος του σε λέξεις ή σε bytes) και ο οποίος αποθηκεύεται ξεχωριστά, οι δομές αυτές παρέχουν όλες τις απαραίτητες πληροφορίες για την αποτίμηση ενός ερωτήματος αναζήτησης. Μια απλή συλλογή κειμένων που αποτελείται από έξι κείμενα (μία γραμμή το καθένα) παρουσιάζεται στο Σχήμα 2.1, ενώ το αντίστοιχο ανεστραμμένο ευρετήριο της συλλογής παρουσιάζεται στο Σχήμα 2.2.

Σε ένα πλήρες σύστημα δεικτοδότησης αρχείων υπάρχουν αρκετές ακόμα δομές, όπως ο πίνακας που αντιστοιχίζει τα αναγνωριστικά κειμένων στις θέσεις των κειμένων στο δίσκο (ή απλά στα ονόματα των αρχείων) καθώς και διάφορες άλλες δομές που χρησιμοποιούνται κατά την δεικτοδότηση των κειμένων.

Γενικά, μπορούμε να θεωρήσουμε πως η διαδικασία δεικτοδότησης μιας συλλογής αποτελείται από δύο σαφώς διαχωρισμένες διαδικασίες: την διαδικασία *λεκτικής ανάλυσης* (parsing process) των κειμένων, η οποία αναλύει ένα κείμενο σε ένα σύνολο από εμφανίσεις $\langle t, d, f_{d,t} \rangle$, και την διαδικασία *δημιουργίας του ανεστραμμένου αρχείου*, η οποία παίρνει σαν είσοδο μία ροή από εμφανίσεις από την διαδικασία λεκτικής ανάλυσης και δημιουργεί το λεξικό και το ανεστραμμένο αρχείο στο δίσκο. Η διαδικασία δημιουργίας του ανεστραμμένου ευρετηρίου καλείται συχνά και διαδικασία *αντιστροφή* της συλλογής.

2.2 Αναζήτηση Κειμένων

2.2.1 Ορισμός αναζήτησης

Στις παραδοσιακές σχεσιακές βάσεις δεδομένων ένα ερώτημα αναζήτησης μεταφράζεται συνήθως ως "*βρες όλες τις εγγραφές που περιέχουν την τιμή X στο πεδίο A*". Τέτοιου είδους αναζητήσεις είναι αρκετά σπάνιες στις αναζητήσεις σε συλλογές κειμένων. Παρά το γεγονός πως σε κάποιες συλλογές μπορεί να υπάρχουν για κάθε κείμενο πεδία όπως "συγγραφέας", "τίτλος" ή κατηγορίες στις οποίες ανήκει το κείμενο, αυτά τα πεδία αυτά δεν χρησιμοποιούνται συχνά και σίγουρα δεν είναι τόσο χρήσιμα όσο είναι τα πεδία και τα κλειδιά σε μία σχεσιακή βάση.

Ο συνήθης τρόπος αναζήτησης σε συλλογές κειμένων είναι η υποβολή ενός ερωτήματος που αποτελείται από κάποιες λέξεις, η επιστροφή από το σύστημα ενός αριθμού από προτεινόμενα κείμενα, ταξινομημένα συνήθως ως προς την σχετικότητα τους με τις λέξεις του ερωτήματος, και τέλος η αξιολόγηση των επιστρεφόμενων αποτελεσμάτων από τον χρήστη,

t	f_t	Inverted list for t
and	1	$\langle 6,2\rangle$
big	2	$\langle 2,2\rangle$ $\langle 3,1\rangle$
dark	1	$\langle 6,1\rangle$
did	1	$\langle 4,1\rangle$
gown	1	$\langle 2,1\rangle$
had	1	$\langle 3,1\rangle$
house	2	$\langle 2,1\rangle$ $\langle 3,1\rangle$
in	5	$\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$ $\langle 6,2\rangle$
keep	3	$\langle 1,1\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$
keeper	3	$\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,1\rangle$
keeps	3	$\langle 1,1\rangle$ $\langle 5,1\rangle$ $\langle 6,1\rangle$
light	1	$\langle 6,1\rangle$
never	1	$\langle 4,1\rangle$
night	3	$\langle 1,1\rangle$ $\langle 4,1\rangle$ $\langle 5,2\rangle$
old	4	$\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 4,1\rangle$
sleep	1	$\langle 4,1\rangle$
sleeps	1	$\langle 6,1\rangle$
the	6	$\langle 1,3\rangle$ $\langle 2,2\rangle$ $\langle 3,3\rangle$ $\langle 4,1\rangle$ $\langle 5,3\rangle$ $\langle 6,2\rangle$
town	2	$\langle 1,1\rangle$ $\langle 3,1\rangle$
where	1	$\langle 4,1\rangle$

d	1	2	3	4	5	6
W_d	11.4	13.5	11.4	8.0	11.3	12.6

Σχήμα 2.2: Ανεστραμμένο ευρετήριο για την συλλογή του Σχήματος 2.1.

ο οποίος ελέγχει τα αποτελέσματα που του επιστράφηκαν και προσπελαύνει κάποια από τα έγγραφα τα οποία θεωρεί σχετικά με αυτό που αναζητά. Αν τα αποτελέσματα απέχουν πολύ από την πληροφορία που αναζητά ο χρήστης, ο χρήστης επαναπροσδιορίζει το ερώτημα προσθέτοντας ή τροποποιώντας κάποιες από τις λέξεις του και υποβάλει ξανά το ερώτημα στο σύστημα.

Πέρα από τον τρόπο αναζήτησης πληροφορίας, μια άλλη βασική διαφορά με τις σχεσιακές βάσεις είναι η έννοια της *ικανοποίησης* ενός ερωτήματος. Μία εγγραφή ενός πίνακα ικανοποιεί ένα ερώτημα SQL αν ικανοποιεί μία λογική συνθήκη. Ένα έγγραφο ικανοποιεί ένα ερώτημα αναζήτησης αν ο χρήστης το θεωρεί *σχετικό*. Η έννοια της "σχετικότητας" ενός κειμένου με ένα ερώτημα είναι αρκετά ασαφής, καθώς ένα κείμενο μπορεί να είναι σχετικό με ένα ερώτημα χωρίς να περιέχει κανέναν από τους όρους αναζήτησης, ενώ από την άλλη ένα κείμενο που περιέχει όλους τους όρους αναζήτησης μπορεί να μην είναι

σχετικό με το ερώτημα. Αυτή η ασάφεια εισάγει την έννοια της *αποτελεσματικότητας* του συστήματος: ένα σύστημα αναζήτησης θεωρείται *αποτελεσματικό* αν ένα καλό ποσοστό των πρώτων r αποτελεσμάτων που επιστρέφονται είναι σχετικά με το ερώτημα αναζήτησης. Ο συχνότερος τρόπος μέτρησης της αποτελεσματικότητας ενός συστήματος αναζήτησης είναι η μέτρηση της *ακρίβειας* (precision) και της *ανάκλησης* (recall), του ποσοστού δηλαδή των επιστρεφόμενων κειμένων που είναι σχετικά με το ερώτημα, και αντίστοιχα, του ποσοστού των σχετικών κειμένων που υπάρχουν στη συλλογή και επιστρέφονται από το σύστημα.

Η συχνότερη μορφή ενός ερωτήματος αναζήτησης είναι απλά ένα σύνολο από λέξεις. Υποβάλλοντας ο χρήστης ένα σύνολο από λέξεις ουσιαστικά ζητά από το σύστημα να εντοπίσει τα κείμενα που είναι σχετικά με αυτές τις λέξεις. Υπάρχουν διάφορες παραλλαγές της βασικής αυτής μορφής ερωτήματος, όπως είναι τα *ερωτήματα φράσεων* (phrase queries) όπου ο χρήστης αναζητά τα κείμενα που περιέχουν μία συγκεκριμένη φράση, ή τα *λογικά ερωτήματα* (boolean queries) στα οποία χρησιμοποιούνται διάφοροι λογικοί τελεστές μεταξύ των όρων (`AND`, `OR`, `NOT` κτλ), με την βοήθεια των οποίων μπορούμε να ορίσουμε για παράδειγμα πως αναζητούμε όλα τα κείμενα που περιέχουν οπωσδήποτε την λέξη Α και την λέξη Β, αλλά όχι την λέξη Γ.

2.2.2 Θέματα αναζήτησης

Κατά την δημιουργία ενός συστήματος δεικτοδότησης και αναζήτησης κειμένων υπάρχουν κάποια θέματα που αφορούν την μέθοδο εξαγωγής των λέξεων (λεκτική ανάλυση) για τα οποία πρέπει να ληφθούν διάφορες αποφάσεις κατά τον σχεδιασμό του συστήματος, καθώς επηρεάζουν τόσο την διαδικασία δεικτοδότησης όσο την διαδικασία αναζήτησης. Τα σημαντικότερα από αυτά τα θέματα είναι *η μετατροπή ή όχι των κεφαλαίων γραμμάτων σε πεζά* (case folding), η *λεξιλογική αποκοπή των λέξεων* (stemming) η οποία αφαιρεί βάσει γραμματικών κανόνων τις καταλήξεις, τις πτώσεις και άλλες γραμματικές παραλλαγές των λέξεων κρατώντας μόνο την ρίζα τους, και τέλος η χρήση *απαγορευμένων λέξεων* (stop words) οι οποίες αγνοούνται κατά την διαδικασία της δεικτοδότησης (συνήθως είναι πολύ συχνοί όροι οι οποίοι θεωρητικά έχουν μικρή αξία, όπως οι όροι “the”, “of”, “and” κτλ).

Η μετατροπή κεφαλαίων γραμμάτων σε πεζά έχει μία σχετικά μικρή επίδραση στο σύστημα. Η λεξιλογική αποκοπή επηρεάζει αρκετά τόσο το μέγεθος του ευρετηρίου, όσο και τα αποτελέσματα που επιστρέφονται από ένα ερώτημα αναζήτησης. Για παράδειγμα, αν χρησιμοποιείται λεξιλογική αποκοπή τότε κάθε μία από τις λέξεις “indexed”, “indexing”, “indexes” θα εξαχθεί από την διαδικασία λεκτικής ανάλυσης απλά ως “index” (η

ίδια μετατροπή θα συμβεί και στις λέξεις ενός ερωτήματος). Αυτό απευθείας σημαίνει πως το λεξικό του ευρετηρίου θα περιέχει αρκετά λιγότερους όρους και άρα θα είναι αρκετά μικρότερο, εφόσον δεν θα περιέχει όλες τις παραλλαγές μιας λέξης. Από την μεριά της διαδικασίας αναζήτησης αυτό συνεπάγεται πως αν ένας χρήστης αναζητήσει τις λέξεις "text indexing" τότε θα του επιστραφούν και κείμενα που περιέχουν τις λέξεις "text index", "text indexed", "text indexing", "text indexes" κτλ.

Ο σημαντικότερος λόγος για την χρήση απαγορευμένων λέξεων είναι η μείωση του μεγέθους του ανεστραμμένου αρχείου. Οι λέξεις αυτές είναι συνήθως λέξεις που συναντώνται πολύ συχνά στα κείμενα και έχουν πάρα πολύ μεγάλες ανεστραμμένες λίστες, με αποτέλεσμα να αποτελούν το μεγαλύτερο τμήμα του ανεστραμμένου αρχείου. Αν οι λέξεις αυτές αγνοηθούν κατά την δεικτοδότηση τότε το ανεστραμμένο αρχείο θα είναι τουλάχιστον μία τάξη μεγέθους μικρότερο, αλλά από την άλλη προφανώς θα χάσουμε κάποια πληροφορία, κάνοντας την διαδικασία αναζήτησης λιγότερο αποτελεσματική.

Η επιλογή της μετατροπής κεφαλαίων γραμμάτων σε πεζά καθώς και της λεξιλογικής αποκοπής είναι σχεδόν δεδομένη για όλες τις σύγχρονες μηχανές αναζήτησης, ενώ οι απαγορευμένες λέξεις συνήθως δεν χρησιμοποιούνται, καθώς από τη μία οι συχνές λέξεις είναι πάρα πολύ χρήσιμες για την αποτίμηση ερωτημάτων φράσεων, ενώ παράλληλα, με τις τρέχουσες μεθόδους συμπίεσης οι λίστες των όρων αυτών πιάνουν αρκετά μικρότερο χώρο στο δίσκο (το γεγονός πως οι όροι αυτοί συναντώνται πολύ συχνά έχει σαν αποτέλεσμα να μπορούμε να κωδικοποιήσουμε κάθε εμφάνιση ενός τέτοιου όρου σε ένα κείμενο με έναν πολύ μικρό αριθμό από bytes, συνήθως με μόνο 1 byte).

Άλλα λιγότερο σημαντικά θέματα είναι η δεικτοδότηση ή όχι των περιεχομένων των ετικετών (tags) των HTML και XML αρχείων και το αν οι λέξεις που χωρίζονται με παύλα θεωρούνται μία ή δύο λέξεις.

2.2.3 Αποτίμηση λογικών ερωτημάτων

Κατά την αποτίμηση ενός λογικού ερωτήματος το σύστημα καλείται να επιστρέψει όλα τα κείμενα τα οποία ικανοποιούν μία λογική συνθήκη. Τα ερωτήματα αυτά αποτελούνται από ένα σύνολο λέξεων μεταξύ των οποίων υπάρχουν διάφοροι λογικοί τελεστές, όπως οι `AND`, `OR` και `NOT`. Για παράδειγμα, το λογικό ερώτημα "cat `AND` food `OR` dog" μεταφράζεται ως "βρες όλα τα κείμενα που περιέχουν τις λέξεις cat και food, ή την λέξη dog". Στα λογικά ερωτήματα δεν εκτελούμε κανενός είδους κατάταξη των αποτελεσμάτων: επιστρέφουμε όλα τα κείμενα που ικανοποιούν την λογική συνθήκη του ερωτήματος.

Τα πιο συχνά λογικά ερωτήματα είναι τα ερωτήματα *σύζευξης*, στα οποία ο χρήστης αναζητά ένα σύνολο κειμένων που περιέχουν όλες τις λέξεις που ορίζει. Τα ερωτήματα αυτά είναι της μορφής "$term_1$ AND $term_2$ AND ... AND $term_k$". Ο πιο απλός αλγόριθμος αποτίμησης ενός τέτοιου ερωτήματος συνίσταται αρχικά στην ανάκτηση των ανεστραμμένων λιστών όλων των όρων του ερωτήματος και έπειτα στον υπολογισμό της τομής αυτών των λιστών. Τα κείμενα που περιέχονται στην τομή των λιστών είναι αυτά που περιέχουν όλους τους όρους και αυτά που τελικά επιστρέφονται στο χρήστη.

Ένας πιο αποδοτικός αλγόριθμος παρουσιάζεται στο Σχήμα 2.3, όπου οι λίστες των όρων ανακτώνται με τη σειρά από τον όρο με τη μικρότερη συχνότητα εμφάνισης προς τον όρο με την μεγαλύτερη συχνότητα εμφάνισης. Αρχικά θέτουμε ως σύνολο υποψηφίων κειμένων C τα κείμενα που περιέχονται στην ανεστραμμένη λίστα του λιγότερο συχνού όρου, και κάθε φορά που ανακτούμε την λίστα ενός όρου t υπολογίζουμε την μέχρι στιγμής τομή των λιστών, διαγράφοντας από το σύνολο C όσα κείμενα δεν περιέχουν τον όρο t.

```
 1.  Find query term t with the smallest f_t
 2.  Read its inverted list I_t
 3.  Set C ← It    /* C is the list of candidate documents */
 4.  For each remaining term t with the smallest f_t
 5.     Read its inverted list I_t
 6.     For each d ∈ C
 7.        If d ∉ I_t
 8.           C ← C − {d}
 9.     If C = ∅
10.        Return /* there are no answers */
11.  Return C
```

Σχήμα 2.3: Αλγόριθμος αποτίμησης λογικού ερωτήματος.

Υπάρχουν πολλοί λόγοι που επιλέγουμε να ξεκινήσουμε από τους λιγότερο συχνούς όρους για να υπολογίσουμε την τομή των ανεστραμμένων λιστών, αφαιρώντας κάθε φορά από το σύνολο των υποψηφίων κειμένων όσα κείμενα δεν περιέχουν τον τρέχον όρο. Καταρχήν, με τον τρόπο αυτό ελαχιστοποιούμε την προσωρινή μνήμη που απαιτείται για την διατήρηση της λίστας των κειμένων που θα επιστρέψουμε. Επίσης, η αποτίμηση των ερωτημάτων γίνεται πιο γρήγορα, εφόσον για κάθε ανεστραμμένη λίστα που ανακτούμε εκτελούμε έναν αριθμό από λειτουργίες αναζήτησης, αντί για λειτουργίες συγχώνευσης. Τέλος, και ίσως πιο σημαντικό, προσπελάζοντας τις λίστες από τις λιγότερο συχνές προς τις περισσότερο συχνές μειώνουμε τον αριθμό των υποψήφιων κειμένων πολύ γρήγορα, πιθανόν ακόμα και στο μηδέν, τερματίζοντας έτσι την διαδικασία αναζήτησης νωρίτερα.

Για την αποτίμηση ερωτημάτων διάζευξης ("term1 OR term2") υπολογίζουμε την *ένωση*

των λιστών των όρων του ερωτήματος, αντί για την τομή τους. Ο αλγόριθμος του Σχήματος 2.3 μπορεί εύκολα να επεκταθεί ώστε να υπολογίζει και ερωτήματα διάζευξης.

2.2.4 Κατάταξη αποτελεσμάτων

Όλες οι σύγχρονες μηχανές αναζήτησης κατατάσσουν τα επιστρεφόμενα κείμενα, τοποθετώντας υψηλότερα τα κείμενα που θεωρούνται πιο σχετικά με το ερώτημα αναζήτησης. Ένα *μέτρο ομοιότητας*, χρησιμοποιείται για να μετρήσουμε την *εγγύτητα* ενός κειμένου με το ερώτημα. Η αρχή πάνω στην οποία βασίζεται το μέτρο ομοιότητας είναι πως όσο υψηλότερη τιμή έχει ένα κείμενο βάσει του μέτρου ομοιότητας, τόσο μεγαλύτερη είναι η εκτιμώμενη πιθανότητα να κριθεί ως "σχετικό" από τον χρήστη. Τελικά, τα r "πιο σχετικά κείμενα βάσει του μέτρου ομοιότητας" επιστρέφονται στο χρήστη σαν προτεινόμενες απαντήσεις.

Στην πιο απλή μορφή ερωτημάτων, όπου ο χρήστης υποβάλει στο σύστημα ένα σύνολο από λέξεις, το μέτρο ομοιότητας ορίζεται από διάφορα απλά στατιστικά. Στη συνέχεια θα αναφέρουμε κάποια μέτρα ομοιότητας για απλά ερωτήματα αναζήτησης (ο χρήστης υποβάλλει ένα σύνολο λέξεων), τα οποία μπορούν να επεκταθούν και για τις περιπτώσεις ερωτημάτων φράσεων.

Τα περισσότερα μέτρα ομοιότητας αποτελούν σύνθεση των παρακάτω βασικών στατιστικών τιμών:

- $f_{d,t}$: η συχνότητα εμφάνισης του όρου t στο κείμενο d
- $f_{q,t}$: η συχνότητα εμφάνισης του όρου t στο ερώτημα q
- f_t: το πλήθος των κειμένων που περιέχουν μία ή περισσότερες φορές τον όρο t
- F_t: το πλήθος των συνολικών εμφανίσεων του όρου t στη συλλογή κειμένων
- N: το πλήθος των κειμένων της συλλογής
- n: το πλήθος των διακριτών λέξεων της συλλογής που έχουν δεικτοδοτηθεί (πλήθος λέξεων που περιέχονται στο λεξικό)

Αυτές οι βασικές τιμές συνδυάζονται με τέτοιο τρόπο στα μέτρα ομοιότητας ώστε τα αποτελέσματα να ικανοποιούν τρεις γενικές συνθήκες μονοτονικότητας:

1. Λιγότερο βάρος δίνεται στους όρους που εμφανίζονται σε πολλά κείμενα.
2. Περισσότερο βάρος δίνεται στους όρους που εμφανίζονται πολλές φορές σε ένα κείμενο.
3. Λιγότερο βάρος δίνεται σε κείμενα που περιέχουν πολλούς όρους.

Μέτρο συνημιτόνου

Ένα πολύ συνηθισμένο μέτρο το οποίο αποδεικνύεται αρκετά αποτελεσματικό στην πράξη είναι το *μέτρο συνημιτόνου*, σύμφωνα με το οποίο υπολογίζουμε το συνημίτονο της γωνίας στον n-διάστατο χώρο μεταξύ ενός διανύσματος ερώτησης $\langle w_{q,t} \rangle$ και ενός διανύσματος κειμένου $\langle w_{d,t} \rangle$. Υπάρχουν πολλές παραλλαγές του μέτρου συνημιτόνου. Ένα παράδειγμα είναι το ακόλουθο:

$$w_{q,t} = \ln\left(1 + \tfrac{N}{f_t}\right) \qquad w_{d,t} = 1 + \ln f_{d,t}$$

$$W_d = \sqrt{\textstyle\sum_t w_{d,t}^2} \qquad W_q = \sqrt{\textstyle\sum_t w_{q,t}^2}$$

$$S_{q,d} = \frac{\sum_t w_{d,t} \cdot w_{q,t}}{W_d \cdot W_q}$$

Η ομοιότητα του ερωτήματος q με το κείμενο d εκφράζεται από τον όρο $S_{q,d}$. Ο όρος W_q μπορεί να παραληφθεί, καθώς είναι σταθερός για ένα δεδομένο ερώτημα και όλα τα κείμενα της συλλογής και δεν επηρεάζει την κατάταξη των αποτελεσμάτων.

Ο όρος $w_{q,t}$ συνήθως αντιπροσωπεύει την ιδιότητα συχνά αποκαλούμενη ως *αντίστροφη συχνότητα εγγράφου* του όρου (inverse document frequency), ή IDF, ενώ ο όρος $w_{d,t}$ αντιπροσωπεύει την *συχνότητα όρου* (term frequency), ή TF, για αυτό συχνά τα μέτρα αυτά ομοιότητας χαρακτηρίζονται με τον συμβολισμό TF×IDF.

Μέτρο Okapi

Διάφορα μέτρα ομοιότητας τα οποία βασίζονται απευθείας σε αρχές της στατιστικής έχουν αποδειχθεί πολύ πετυχημένα σε συστήματα αναζήτησης κειμένων, και χρησιμοποιούνται ευρέως σε μελέτες που δημοσιεύονται σε γνωστά συνέδρια σχετικά με τον τομέα της αναζήτησης κειμένων όπως το TREC[1]. Το κυριότερο από αυτά τα μέτρα είναι το μέτρο Okapi:

$$\begin{aligned} w_{q,t} &= \ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \cdot \frac{(k_3 + 1) \cdot f_{q,t}}{k_3 + f_{q,t}} \\ w_{d,t} &= \frac{(k_1 + 1) \cdot f_{d,t}}{K_d + f_{d,t}} \\ K_d &= k_1 \cdot \left((1 - b) + b \cdot \frac{W_d}{W_A}\right) \\ S_{q,d} &= \sum_{t \in q} w_{q,t} \cdot w_{d,t} \end{aligned}$$

[1] Text Retrieval Conference: http://trec.nist.gov

όπου τα k_1 και b είναι παράμετροι με προεπιλεγμένες τιμές 1.2 και 0.75 αντίστοιχα, το k_3 είναι μία παράμετρος στην οποία θέτουμε την τιμή ∞ έτσι ώστε η παράσταση $\frac{(k_3+1)\cdot f_{q,t}}{(k_3+f_{q,t})}$ να είναι ίση με $f_{q,t}$, και τα W_d και W_A είναι το μήκος του κειμένου και το μέσο μήκος κειμένου, εκφρασμένα σε οποιαδήποτε κατάλληλη μονάδα (συνήθως σε λέξεις ή bytes)

Σε όλα τα προαναφερθέντα μέτρα ομοιότητας ένα κείμενο μπορεί να έχει μεγάλη ομοιότητα με ένα ερώτημα αναζήτησης ακόμα και αν δεν περιέχει κάποιους από τους όρους του ερωτήματος.

2.2.5 Αλγόριθμος κατάταξης βάσει συχνοτήτων

Δεδομένου ενός μέτρου ομοιότητας, η κατάταξη των κειμένων βάσει της ομοιότητας τους με το ερώτημα είναι αρκετά απλή: υπολογίζουμε για κάθε κείμενο την ομοιότητα του με το ερώτημα και έπειτα επιστρέφουμε τα r κείμενα με την μεγαλύτερη ομοιότητα. Στο Σχήμα 2.4 παρουσιάζεται ένας απλοϊκός αλγόριθμος για τον υπολογισμό του μέτρου συνημιτόνου μεταξύ ενός ερωτήματος και όλων των κειμένων της συλλογής.

```
1.  Calculate w_{q,t} for each query term t in q
2.  For each document d in the collection
3.      S_{q,d} ← 0
4.      For each query term t
5.          Calculate or read w_{d,t}
6.          S_{q,d} ← S_{q,d} + w_{q,t} × w_{d,t}
7.      Calculate or read W_d
8.      S_{q,d} ← S_{q,d} / W_d
9.  Identify the r greatest S_{q,d} values and return the corresponding documents
```

Σχήμα 2.4: Αλγόριθμος κατάταξης των κειμένων της συλλογής βάσει του μέτρου συνημιτόνου, με χρήση συχνοτήτων.

Το μειονέκτημα του απλοϊκού αυτού αλγορίθμου είναι πως υπολογίζει εξαντλητικά το μέτρο ομοιότητας για κάθε κείμενο της συλλογής, ενώ σε μία τυπική περίπτωση ισχύει $r \ll N$, δηλαδή μόνο ένα μικρό ποσοστό των κειμένων επιστρέφονται στα αποτελέσματα. Για την πλειοψηφία των κειμένων η ομοιότητα τους με το ερώτημα είναι μηδαμινή, γεγονός το οποίο εκμεταλλεύονται διάφοροι αποδοτικότεροι αλγόριθμοι που θα δούμε στη συνέχεια.

2.2.6 Αλγόριθμος κατάταξης βάσει ανεστραμμένων λιστών

Ο αλγόριθμος κατάταξης των αποτελεσμάτων ενός ερωτήματος αναζήτησης χρησιμοποιώντας ένα ανεστραμμένο αρχείο παρουσιάζεται στο Σχήμα 2.5. Στον αλγόριθμο αυτό

επεξεργαζόμαστε σε κάθε βήμα έναν όρο του ερωτήματος. Αρχικά, κάθε κείμενο έχει ομοιότητα μηδέν με το ερώτημα. Αυτό αναπαρίσταται δημιουργώντας έναν πίνακα A που περιέχει N τιμές ομοιότητας (μία για κάθε κείμενο), οι οποίες αναφέρονται ως *συσσωρευτές* (accumulators), και τις οποίες αρχικοποιούμε στην τιμή μηδέν. Έπειτα, για κάθε όρο t διασχίζουμε την ανεστραμμένη λίστα του, και για κάθε όρο d που εμφανίζεται στην λίστα του αυξάνουμε τον αντίστοιχο συσσωρευτή A_d κατά την συνεισφορά του όρου t στην ομοιότητα του d με το ερώτημα. Όταν ολοκληρωθεί η επεξεργασία όλων των όρων του ερωτήματος, υπολογίζονται τα τελικά αποτελέσματα ομοιότητας S_d για κάθε κείμενο d, διαιρώντας κάθε συσσωρευτή A_d με την αντίστοιχη τιμή του W_d (μήκος κειμένου). Τέλος, εντοπίζονται τα r μεγαλύτερα αποτελέσματα ομοιότητας και επιστρέφονται τα αντίστοιχα κείμενα στο χρήστη.

```
1.  Allocate an accumulator A_d for each document d and set A_d ← 0
2.  For each query term t in q
3.      Calculate w_{q,t} and fetch the inverted list I_t of t
4.      For each pair ⟨d, f_{d,t}⟩ in I_t
5.          Calculate w_{d,t}
6.          A_d ← A_d + w_{q,t} × w_{d,t}
7.  Read the array of W_d values
8.  For each A_d > 0
9.      S_d ← A_d / W_d
10. Identify the r greatest S_d values and return the corresponding documents
```

Σχήμα 2.5: Αλγόριθμος κατάταξης των κειμένων της συλλογής βάσει του μέτρου συνημιτόνου, με χρήση των ανεστραμμένων λιστών.

Το κόστος κατάταξης των αποτελεσμάτων χρησιμοποιώντας ένα ανεστραμμένο αρχείο είναι πολύ μικρότερο από τον εξαντλητικό αλγόριθμο του Σχήματος 2.4. Δεδομένου ενός ερωτήματος που περιέχει τρεις όρους, η αποτίμηση του ερωτήματος περιλαμβάνει την εύρεση των τριών όρων στο λεξικό, την ανάκτηση και επεξεργασία των ανεστραμμένων λιστών τους (συνήθως μεγέθους κάποιων εκατοντάδων Kb μέχρι κάποια Mb), και τέλος δύο γραμμικά περάσματα από έναν πίνακα μεγέθους N (όπου N το πλήθος των κειμένων, με τις συλλογές κειμένων να περιέχουν συνήθως κάποια εκατομμύρια κείμενα). Η όλη διαδικασία απαιτείται λιγότερο από ένα δευτερόλεπτο σε ένα τυπικό υπολογιστή.

Παρόλα αυτά, υπάρχουν ακόμα σημαντικά κόστη για τον συγκεκριμένο αλγόριθμο: μεγάλα μεγέθη μνήμης απαιτούνται για τους N συσσωρευτές (το N είναι συνήθως της τάξης των εκατομμυρίων) καθώς και για την ανάκτηση των ανεστραμμένων λιστών. Επίσης, απαιτείται αρκετός χρόνος επεξεργαστή για την επεξεργασία των ανεστραμμένων λιστών

και αρκετός φόρτος στο δίσκο για την ανάκτηση των λιστών, οι οποίες μπορεί να είναι αρκετά μεγάλες αν έχουμε κάποιον πολύ συχνό όρο σε μεγάλες συλλογές κειμένων.

Όλα τα προηγούμενα κόστη όμως μπορούν να μειωθούν, χρησιμοποιώντας έναν πιο αποδοτικό αλγόριθμο από αυτόν που παρουσιάζεται στο Σχήμα 2.5.

Συγκεκριμένα, μπορούμε να εκμεταλλευτούμε το γεγονός πως για τα περισσότερα κείμενα οι συσσωρευτές θα έχουν τιμή μηδέν, εφόσον δεν θα περιέχουν κανέναν όρο του ερωτήματος, και έτσι διατηρούμε μόνο τους μη-μηδενικούς συσσωρευτές στη μνήμη. Ο αλγόριθμος που παρουσιάζεται στο Σχήμα 2.6 επιστρέφει τα r κείμενα που έχουν την μεγαλύτερη ομοιότητα με το ερώτημα q, υπολογίζοντας σε κάθε συσσωρευτή A_d μία τιμή ανάλογη του μέτρου ομοιότητας $\cos(q, d)$. Για την πιο ολοκληρωμένη διατύπωση του αλγορίθμου, υποθέτουμε την ύπαρξη ενός λεξικού για την αντιστοίχηση των όρων στις ανεστραμμένες λίστες του και τις συχνότητες τους, ενώ θεωρούμε πως χρησιμοποιείται λεξιλογική αποκοπή των λέξεων (βλ. Ενότητα 2.2.2).

```
 1.  A ← ∅
 2.  For each query term t
 3.      Stem t
 4.      Search the lexicon for t
 5.      Read f_t and the address of I_t
 6.      w_t ← 1 + log(N/f_t)
 7.      Read inverted list I_t
 8.      For each ⟨d, f_{d,t}⟩ pair in I_t
 9.          If A_d ∉ A
10.              A_d ← 0
11.              A ← A + {A_d}
12.          A_d ← A_d + log(1 + f_{d,t}) × w_t
13.  For each A_d ∈ A
14.      Ad ← A_d/W_d
15.  For 1 ≤ i ≤ r
16.      Selected d such that A_d = max{A}
17.      Present d to user
18.      A ← A − {A_d}
```

Σχήμα 2.6: Αποδοτικός αλγόριθμος κατάταξης κειμένων με χρήση ανεστραμμένων λιστών.

Υπάρχουν διάφορες παραλλαγές του αλγορίθμου αυτού που ελαχιστοποιούν τόσο τον χρόνο αποτίμησης ενός ερωτήματος όσο και την μνήμη που χρησιμοποιείται. Για παράδειγμα, συσσωρευτές που έχουν πολύ χαμηλή τιμή είναι λογικό να αφαιρούνται από την μνήμη καθώς δεν πρόκειται να καταταγούν στα r έγγραφα με την υψηλότερη ομοιότητα (ή παρόμοια, είναι λογικό να δημιουργούνται συσσωρευτές μόνο για κείμενα που είναι πιθανό να βρεθούν στα τελικά αποτελέσματα), ενώ χρησιμοποιούνται δομές όπως ο σωρός ελαχίστων για την

εύρεση των r εγγράφων με την υψηλότερη ομοιότητα, ελαχιστοποιώντας τις απαιτούμενες συγκρίσεις. Άλλες βελτιστοποιήσεις περιλαμβάνουν την ανάκτηση των ανεστραμμένων λιστών βάσει της συχνότητας των όρων τους (ή βάσει της διάταξης τους στο δίσκο), την διατήρηση στη μνήμη ανά πάσα στιγμή μόνο των r συσσωρευτών με τις υψηλότερες τιμές, τον τερματισμό της διαδικασίας αποτίμησης όταν το υπόλοιπο της διαδικασίας δεν πρόκειται να αλλάξει την κατάταξη των r κειμένων με τις υψηλότερες τιμές ομοιότητας κ.α..

2.2.7 Ευρετήριο επιπέδου λέξεων

Σε όλες τις προηγούμενες μεθόδους θεωρήσαμε πως η ανεστραμμένη λίστα ενός όρου t είναι ένα σύνολο από ζεύγη $\langle d,\ f_{d,t}\rangle$, δηλαδή υποθέσαμε πως διατηρούμε ένα ευρετήριο επιπέδου κειμένων (για κάθε όρο διατηρούμε την πληροφορία του σε ποια κείμενα εμφανίζεται και πόσες φορές εμφανίζεται σε κάθε κείμενο). Στην περίπτωση ενός ευρετηρίου επιπέδου λέξεων, κάθε ανεστραμμένη λίστα περιέχει ένα σύνολο από πλειάδες

$$\langle d;\ f_{d,t};\ pos_1, pos_2, ..., pos_{f_{d,t}}\rangle$$

δηλαδή για κάθε κείμενο d στο οποίο εμφανίζεται ο t διατηρούμε επίσης την πληροφορία σε ποιες θέσεις εμφανίζεται ο t στο κείμενο.

Η επιπλέον αυτή πληροφορία είναι πολύ χρήσιμη κατά την αποτίμηση ερωτημάτων φράσεων, καθώς για παράδειγμα για την εύρεση της φράσης "information retrieval systems" μπορούμε αρχικά να βρούμε όλα τα κείμενα που περιέχουν και τις τρεις αυτές λέξεις και στη συνέχεια να περιορίσουμε τα αποτελέσματα που θα επιστρέψουμε σε εκείνα μόνο τα κείμενα στα οποία οι λέξεις αυτές είναι γειτονικές. Η πληροφορία των θέσεων εμφάνισης μπορεί επίσης να βοηθήσει και στην περίπτωση των απλών ερωτημάτων αναζήτησης, καθώς μπορούμε να κατατάσσουμε υψηλότερα τα κείμενα στα οποία οι λέξεις της αναζήτησης εμφανίζονται σχετικά κοντά μεταξύ τους. Για τους παραπάνω λόγους, η πλειοψηφία των σύγχρονων μηχανών αναζήτησης χρησιμοποιούν ευρετήρια επιπέδου λέξεων.

2.3 Συμπίεση ευρετηρίου

Ένα πολύ σημαντικό θέμα που αφορά την αποθήκευση του ευρετηρίου είναι ο τρόπος με τον οποίο θα αναπαραστήσουμε κάθε αποθηκευμένη τιμή, όπως τα αναγνωριστικά κειμένων, οι συχνότητες εμφάνισης των όρων σε κάθε αρχείο και οι θέσεις εμφάνισης των όρων μέσα σε κάθε αρχείο.

Η επιλογή ενός συγκεκριμένου αριθμού από bits ή bytes για την αναπαράσταση των αριθμών είναι αρκετά αυθαίρετη και έχει δύο σημαντικά μειονεκτήματα: από τη μία υπάρχει πάντα ο κίνδυνος της υπερχείλισης ενός αριθμού, δημιουργώντας προβλήματα κλιμάκωσης (για παράδειγμα, υπάρχει ένας μέγιστος αριθμός που μπορούμε να υποστηρίξουμε με K bits), ενώ από την άλλη δαπανάται πολύς χώρος άσκοπα αν οι περισσότερες τιμές είναι πολύ μικρές.

Παρόλα αυτά, η αποθήκευση των τιμών σε μία συμπιεσμένη μορφή επιφέρει σημαντικά οφέλη, αυξάνοντας την αποδοτικότητα του συστήματος. Επιλέγοντας μία κατάλληλη τεχνική κωδικοποίησης των διαφόρων πεδίων που περιέχει μία ανεστραμμένη λίστα, είναι δυνατόν να μειώσουμε δραστικά το μέγεθος του ανεστραμμένου αρχείου. Επίσης, μειώνοντας το μέγεθος των ανεστραμμένων λιστών μειώνουμε παράλληλα το κόστος μεταφοράς δεδομένων από και προς τον δίσκο, με αποτέλεσμα τόσο η εγγραφή όσο και η ανάκτηση των λιστών να απαιτεί λιγότερο χρόνο και έτσι να έχουμε βελτιωμένους χρόνους αποτίμησης των ερωτημάτων αναζήτησης. Τέλος, η συμπίεση των λιστών αυξάνει την πιθανότητα κάποιο τμήμα μιας λίστας που απαιτείται για την αποτίμηση ενός ερωτήματος να βρίσκεται στην κρυφή μνήμη (cache) του συστήματος.

Συμπερασματικά, η συμπίεση του ανεστραμμένου αρχείου μειώνει τον απαιτούμενο αποθηκευτικό χώρο στο δίσκο, ενώ παράλληλα βελτιώνει τόσο τους χρόνους δεικτοδότησης όσο και τους χρόνους αναζήτησης λόγω της αποδοτικότερης χρήσης του δίσκου.

2.3.1 Αναπαράσταση διαφορών

Το κλειδί στη συμπίεση των ανεστραμμένων λιστών ενός ευρετηρίου επιπέδου κειμένων είναι η παρατήρηση πως, χωρίς βλάβη της γενικότητας, κάθε ανεστραμμένη λίστα μπορεί να αποθηκευτεί σαν μία *μονοτονικά αυξανόμενη ακολουθία ακεραίων*. Για παράδειγμα, ας υποθέσουμε πως ένα όρος εμφανίζεται σε οχτώ κείμενα μιας συλλογή κειμένων, και συγκεκριμένα στα κείμενα με αναγνωριστικά 3, 5, 20, 21, 23, 76, 77, 78. Ο όρος αυτός περιγράφεται στο ανεστραμμένο αρχείο από μία λίστα:

$$\langle 8;\, 3, 5, 20, 21, 23, 76, 77, 78 \rangle$$

της οποίας η τοποθεσίας στο δίσκο περιέχεται στο λεξικό του ευρετηρίου. Πιο γενικά, μία ανεστραμμένη λίστα ενός όρου t περιέχει τον αριθμό των κειμένων f_t στα οποία εμφανίζεται ο όρος, καθώς και τα αναγνωριστικά d_i των f_t αυτών κειμένων:

$$\langle f_t;\, d_1, d_2, ..., d_{f_t} \rangle$$

όπου $d_k < d_{k+1}$, καθώς τα αναγνωριστικά που ανατίθενται στα κείμενα είναι μονοτονικά αυξανόμενα. Επειδή τα αναγνωριστικά κειμένων μέσα σε κάθε λίστα είναι σε αύξουσα διάταξη, και η επεξεργασία της λίστας γίνεται σειριακά ξεκινώντας από την αρχή της λίστας, η λίστα τελικά μπορεί να αποθηκευτεί σαν μία αρχική τιμή ακολουθούμενη από μία *λίστα διαφορών* (d-gap list), των διαφορών $d_{k+1} - d_k$. Για παράδειγμα, η παραπάνω λίστα μπορεί να αποθηκευτεί ως:

$$\langle 8;\, 3, 2, 15, 1, 2, 53, 1, 1 \rangle$$

Όλη η πληροφορία έχει διατηρηθεί, καθώς η αρχική λίστα μπορεί να αναπαραχθεί από την λίστα διαφορών υπολογίζοντας τα προθεματικά αθροίσματα των διαφορών της.

Οι δύο μορφές αναπαράστασης των λιστών είναι ισοδύναμες, αλλά δεν είναι προφανές πως υπάρχει κάποιο κέρδος από τη νέα αναπαράσταση. Αν για παράδειγμα χρησιμοποιούμε 32-bit ακεραίους για την αποθήκευση των διαφορών αυτών δεν έχουμε κάποιο κέρδος στον αποθηκευτικό χώρο. Επιπλέον, η μεγαλύτερη διαφορά στη νέα αναπαράσταση είναι πιθανόν ίδια με το μεγαλύτερο αναγνωριστικό στην αρχική αναπαράσταση, και έτσι αν υπάρχουν N κείμενα στη συλλογή τότε και οι δύο αναπαραστάσεις απαιτούν $\lceil \log N \rceil$ bits ανά δείκτη. Παρ' όλα αυτά, αν θεωρήσουμε κάθε ανεστραμμένη λίστα σαν μία λίστα διαφορών, το άθροισμα των οποίων φράσσεται από το N, τότε μπορούμε να έχουμε μία βελτιωμένη αναπαράσταση των λιστών, κωδικοποιώντας τις λίστες αυτές με αρκετά λιγότερα από $\lceil \log N \rceil$ bits ανά δείκτη.

Έχουν προταθεί διάφορες μέθοδοι συμπίεσης για την αποδοτική αποθήκευση των διαφορών. Οι μέθοδοι αυτοί μπορούμε να πούμε πως γενικά χωρίζονται σε δύο κατηγορίες: τις *καθολικές μεθόδους* (global methods), όπου όλες οι λίστες κωδικοποιούνται χρησιμοποιώντας τις ίδιες παραμέτρους, και τις *τοπικές μεθόδους* (local methods), όπου το μοντέλο κωδικοποίησης κάθε λίστας προσαρμόζεται βάσει κάποιων χαρακτηριστικών της λίστας (συνήθως χρησιμοποιείται η συχνότητα του εκάστοτε όρου). Οι καθολικές μέθοδοι με την σειρά τους χωρίζονται σε δύο υποκατηγορίες, τις *παραμετροποιήσιμες* και τις *μη-παραμετροποιήσιμες* μεθόδους, όπου οι τελευταίες χρησιμοποιούν προεπιλεγμένες παραμέτρους ενώ οι πρώτες χρησιμοποιούν κάποιες παραμέτρους οι τιμές των οποίων έχουν υπολογιστεί βάσει της πραγματικής κατανομής του μεγέθους των διαφορών. Οι τοπικές μέθοδοι είναι πάντα παραμετροποιήσιμες.

Μια απλή μη-παραμετροποιήσιμη μέθοδος, η οποία χρησιμοποιείται συχνά στην κωδικοποίηση των ανεστραμμένων λιστών λόγω των πολύ καλών χαρακτηριστικών της –τόσο από άποψης μείωσης αποθηκευτικού χώρου όσο και από άποψης χρόνου αποκωδικοποίησης–

είναι οι κώδικες *μεταβλητού πλήθους bytes*.

2.3.2 Μη-παραμετροποιήσιμη κωδικοποίηση - Κώδικες μεταβλητού μεγέθους

Μία αποτελεσματική αναπαράσταση των διαφορών από τις οποίες αποτελούνται οι ανεστραμμένες λίστες πρέπει να μπορεί να αναπαριστά τους ακεραίους με έναν μεταβλητό αριθμό από bits. Η χρήση ενός προκαθορισμένου αριθμού bits, είτε 32 είτε 20, δημιουργεί διάφορα προβλήματα όπως είπαμε και νωρίτερα (κίνδυνος υπερχείλισης, άσκοπη σπατάλη χώρου εφόσον οι περισσότεροι ακέραιοι περιέχουν μικρές τιμές). Οι *κώδικες μεταβλητού μεγέθους* αποφεύγουν τα προηγούμενα προβλήματα, καθώς από τη μία μπορούν να αναπαραστήσουν έναν οσοδήποτε μεγάλο ακέραιο, ενώ από την άλλη μπορούν να σχεδιαστούν έτσι ώστε να ευνοούν τις μικρές τιμές. Η αποδοτική αποθήκευση των μικρών τιμών έχει σαν κόστος την λιγότερη αποδοτική αποθήκευση των μεγάλων τιμών, το οποίο φαίνεται πολύ λογικό όσον αφορά την περίπτωση των διαφορών των ανεστραμμένων λιστών.

Οι μέθοδοι κωδικοποίησης μεταβλητού μεγέθους χωρίζονται σε δύο κατηγορίες: τις μεθόδους κωδικοποίησης *μεταβλητού πλήθους bytes* (variable-byte ή bytewise coding) και τις μεθόδους κωδικοποίησης *μεταβλητού πλήθους bits* (variable-bit ή bitwise coding), ανάλογα με το αν ένας αριθμός μπορεί να κωδικοποιηθεί χρησιμοποιώντας έναν μεταβλητό αριθμό από bytes ή bits. Οι κώδικες μεταβλητού πλήθους bits [19, 20, 27, 29] προσφέρουν γενικά καλύτερη απόδοση συμπίεσης, αλλά η αποκωδικοποίηση τους έχει σχετικά αυξημένο κόστος στους υπολογιστές οι οποίοι έχουν σαν βασική μονάδα επεξεργασίας πολλαπλάσια των οχτώ bits. Για παράδειγμα, κατάλληλοι κώδικες μεταβλητού πλήθους bytes πετυχαίνουν ρυθμούς αποκωδικοποίησης τουλάχιστον διπλάσιους από τους αντίστοιχους των κωδίκων μεταβλητού πλήθους bits [24], ενώ παράλληλα προσφέρουν αρκετά καλό λόγο συμπίεσης.

Χρησιμοποιώντας κωδικοποίηση μεταβλητού πλήθους bytes, η διαδικασία της κωδικοποίησης ενός ακεραίου αριθμού x είναι πολύ απλή: αν $x \leq 128$ τότε χρησιμοποιούμε ένα μόνο byte για την αναπαράσταση του x, κωδικοποιώντας το $x - 1$ στο δυαδικό σύστημα και θέτοντας '0' στο σημαντικότερο bit του byte. Αλλιώς, τα εφτά λιγότερο σημαντικά bits του $x - 1$ τοποθετούνται σε ένα byte θέτοντας '1' το σημαντικότερο bit, και η τιμή (x div 128) κωδικοποιείται αναδρομικά με τον ίδιο τρόπο. Ο αλγόριθμος κωδικοποίησης και αποκωδικοποίησης ενός αριθμού $x > 1$ χρησιμοποιώντας την μέθοδο αυτή παρουσιάζεται στο Σχήμα 2.7.

Για παράδειγμα, το byte 2 με δυαδική αναπαράσταση 00000010 αναπαριστά τον ακέραιο 3. Το byte 9 με δυαδική αναπαράσταση 00001001 αναπαριστά τον (δεκαδικό) ακέραιο 10. Και

```
1.  x ← x − 1
2.  While x ≥ 128
3.      write_byte(128 + (x mod 128))
4.      x ← (x div 128) − 1
5.  write_byte(x)
```

(α)

```
1.  b ←read_byte(), x ← 0,   p ← 1
2.  While b ≥ 128
3.      x ← x + (b − 127) × p
4.      p ← p × 128
5.      b ← read_byte()
6.  x ← x + (b + 1) × p
```

(β)

Σχήμα 2.7: (α) Κωδικοποίηση και (β) αποκωδικοποίηση ενός αριθμού x με χρήση κωδικών μεταβλητού πλήθους bytes.

το διπλό byte 147:7 με δυαδική αναπαράσταση 11100111 00000111 αναπαριστά τον αριθμό 1044, εφόσον $1044 = 128 \times (7 + 1) + (147 - 127)$.

Το αποτέλεσμα είναι πως, χρησιμοποιώντας κωδικοποίηση μεταβλητού πλήθους bytes, όλοι οι ακέραιοι μεταξύ 1 και 127 κωδικοποιούνται με ένα μόνο byte, οι ακέραιοι μεταξύ 128 και 16,383 με δύο bytes, οι ακέραιοι μεταξύ 16,384 και 2,097,151 με τρία bytes, κτλ.

Ένα άλλο πλεονέκτημα της κωδικοποίησης αυτής, εκτός από το ότι προσφέρει πολύ καλή συμπίεση και γρήγορη αποκωδικοποίηση, είναι το γεγονός πως προσφέρει την δυνατότητα να βρούμε γρήγορα μέσα σε μία κωδικοποιημένη ροή από bytes τον k-οστό κωδικοποιημένο αριθμό: εφόσον κάθε κωδικοποιημένος αριθμός τερματίζει με ένα byte του οποίου το σημαντικότερο bit είναι '0', είναι πολύ εύκολο να προσπεράσουμε ακριβώς $k-1$ κωδικοποιημένους ακεραίους χωρίς να χρειαστεί να τους αποκωδικοποιήσουμε – απλά ελέγχουμε την τιμή του σημαντικότερου bit από κάθε byte.

Οι πιο γνωστοί μη-παραμετροποιήσιμοι κώδικες μεταβλητού πλήθους bits είναι οι κώδικες *Elias Gamma* και *Delta* [28, 29].

2.3.3 Παραμετροποιήσιμη κωδικοποίηση - Κώδικες Golomb και Rice

Το μειονέκτημα των μη-παραμετροποιήσιμων μεθόδων κωδικοποίησης είναι πως δε λαμβάνουν υπ' όψιν τους την κατανομή του μήκους των τιμών που θέλουμε να κωδικοποιήσουμε. Οι *κώδικες Golomb* και *Rice* [28, 29] έχουν αυτή ακριβώς την ιδιότητα .

Στην κωδικοποίηση Golomb, η οποία παρουσιάζεται στο Σχήμα 2.8, αρχικά παραγοντο-

```
1.  Factor x ≥ 1 into q × b + r + 1, where 0 ≤ r ≤ b
2.  Code q + 1 in unary
3.  e ← ⌈log2(b)⌉
4.  g ← 2^e − b
5.  If 0 ≤ r ≤ g then
6.      Code r in binary using e − 1 bits
7.  Else
8.      Code r + g in binary using e bits
```

Σχήμα 2.8: Κωδικοποίηση του ακεραίου x με χρήση κωδίκων Golomb.

ποιούμε έναν αριθμό $x \geq 1$ στη μορφή $q \times b + r + 1$, όπου το b είναι παράμετρος της κωδικοποίησης και η οποία επιλέγεται βάσει της κατανομής των τιμών. Έπειτα, για την αποθήκευση του q χρησιμοποιούμε την *μοναδιαία αναπαράσταση*, στην οποία ο αριθμός x αναπαρίσταται από $x-1$ συνεχόμενα '1' ακολουθούμενα από ένα '0', ενώ το e αποθηκεύεται βάσει της δυαδικής του αναπαράστασης.

Όταν η παράμετρος b είναι δύναμη του δύο, δηλαδή της μορφής $b = 2^k$, τότε στο βήμα 5 του Σχήματος 2.8 κωδικοποιούμε πάντα το r με k ακριβώς bits. Οι κώδικες αυτοί ονομάζονται κώδικες Rice και προσφέρουν απλούστερη και άρα γρηγορότερη κωδικοποίηση και αποκωδικοποίηση.

2.3.4 Συμπίεση ανεστραμμένων λιστών

Για την αποδοτική αποθήκευση του ευρετηρίου θα θεωρήσουμε πως χρησιμοποιούμε κώδικες μεταβλητού πλήθους bytes για την αναπαράσταση των ανεστραμμένων λιστών. Συγκεκριμένα, έστω πως έχουμε ένα ευρετήριο *επιπέδου κειμένων*. Στην περίπτωση αυτή, η ανεστραμμένη λίστα l ενός όρου t θα είναι της μορφής:

$$\langle f_t;\, d_1, d_2, ..., d_{f_t} \rangle$$

Αρχικά αναπαριστούμε την λίστα με την μορφή διαφορών, όπως περιγράφηκε και στην Ενότητα 2.3.1. Έτσι η ανεστραμμένη λίστα θα είναι πλέον της μορφής:

$$\langle f_t;\, d_1, d_2 - d_1, d_3 - d_2, ..., d_{f_t} - d_{f_t - 1} \rangle$$

Έπειτα αποθηκεύουμε τους αριθμούς αυτούς, συμπεριλαμβανομένου και του f_t, χρησιμοποιώντας κώδικες μεταβλητού πλήθους bytes, όπως περιγράφηκε στην Ενότητα 2.3.2. Εφόσον οι διαφορές $d_k - d_{k-1}$ είναι συνήθως πολύ μικρότερες από τα αρχικά αναγνωριστικά d_i των κειμένων, ένα ή δύο bytes αρκούν για την αποθήκευση κάθε διαφοράς, ενώ για το f_t αρκεί ένα byte, αφού συνήθως αυτό είναι ένας πολύ μικρός αριθμός.

Η παραπάνω τεχνική μπορεί επίσης να χρησιμοποιηθεί για την αποδοτική αποθήκευση ενός ευρετηρίου *επιπέδου λέξεων*, όπου η ανεστραμμένη λίστα πλέον αποτελείται από έναν πλήθος από *εμφανίσεις* (postings) της μορφής $\langle d_i;\ f_{d_i,t};\ pos_1, pos_2, ..., pos_{f_{d_i,t}} \rangle$ αντί για ένα πλήθος από ζεύγη $\langle d_i,\ f_{d_i,t} \rangle$. Σε αυτή την περίπτωση, οι θέσεις pos_i των όρων μέσα σε κάθε κείμενο μπορούν επίσης να αναπαρασταθούν σαν διαφορές, εφόσον αποτελούν μία αύξουσα διάταξη ακεραίων, και έπειτα να κωδικοποιηθούν χρησιμοποιώντας κώδικες μεταβλητού πλήθους bytes. Έτσι, τα αναγνωριστικά κειμένων d_i μεταξύ διαδοχικών εμφανίσεων αναπαρίστανται σαν διαφορές, ενώ επίσης οι θέσεις pos_i μέσα σε κάθε εμφάνιση αναπαρίστανται επίσης σαν διαφορές.

2.3.5 Αποδοτική επέκταση των λιστών

Ένα επίσης πολύ χρήσιμο και επιθυμητό χαρακτηριστικό της παραπάνω τεχνικής (αναπαράσταση διαφορών και κωδικοποίηση με μεταβλητό πλήθος bytes) είναι πως, όταν μετά την λεκτική ανάλυση ενός κειμένου d_i μία νέα εμφάνιση $\langle d_i;\ f_{d_i,t};\ pos_1, pos_2, ..., pos_{f_{d_i,t}} \rangle$ πρέπει να προστεθεί στην ανεστραμμένη λίστα ενός όρου t, τότε δεν χρειάζεται να αποκωδικοποιήσουμε την λίστα, να προσθέσουμε την νέα εμφάνιση και έπειτα να ξανακωδικοποιήσουμε την λίστα. Αρκεί να ξέρουμε το τελευταίο αναγνωριστικό κειμένου d_{last} το οποίο κωδικοποιήθηκε στην λίστα. Βάσει του d_{last} αναπαριστούμε το αναγνωριστικό d_i της νέας εμφάνισης σαν διαφορά $d_i - d_{last}$, ενώ επίσης αναπαριστούμε σαν διαφορές τις θέσεις pos_i μέσα στη νέα εμφάνιση. Έπειτα κωδικοποιούμε τη νέα εμφάνιση, η οποία πλέον θα έχει την μορφή:

$$\langle d_i - d_{last};\ f_{d_i,t};\ pos_1, pos_2 - pos_1, ..., pos_{f_{d_i,t}} - pos_{f_{d_i,t}-1} \rangle$$

χρησιμοποιώντας κώδικες μεταβλητού πλήθους bytes και προσθέτουμε την νέα εμφάνιση ακριβώς μετά το τέλος της ανεστραμμένης λίστας. Τέλος, θέτουμε $d_{last} \leftarrow d_i$. Με τον τρόπο αυτό μπορούμε να κωδικοποιούμε ανεξάρτητα κάθε νέα εμφάνιση που θέλουμε να προσθέσουμε σε μία λίστα, *χωρίς να χρειάζεται να διαβάσουμε και να αποκωδικοποιήσουμε την μέχρι στιγμής ανεστραμμένη λίστα*, και έπειτα απλά να την προσθέτουμε στο τέλος της λίστας.

Η πληροφορία d_{last} του τελευταίου αναγνωριστικού που έχουμε κωδικοποιήσει σε μία λίστα συνήθως διατηρείται στο λεξικό. Έτσι πλέον για κάθε όρο διατηρούμε στο λεξικό, εκτός από την θέση της ανεστραμμένης λίστας του στο δίσκο και το πλήθος f_t των κειμένων στα οποία εμφανίζεται, και το τελευταίο αναγνωριστικό κειμένου που έχουμε κωδικοποιήσει, έτσι ώστε να μη χρειάζεται να διαβάσουμε την λίστα του από τον δίσκο για να προσθέσουμε μία νέα εμφάνιση στην λίστα του.

Κεφαλαιο 3

Μεθοδοι δημιουργιας ευρετηριων

3.1 Ορισμός του προβλήματος

Με τον όρο *δεικτοδότηση ή αντιστροφή* μιας συλλογής κειμένων εννοούμε την διαδικασία της δημιουργία ενός ανεστραμμένου ευρετηρίου για την συλλογή αυτή, το οποίο να επιτρέπει την αποτίμηση ερωτημάτων αναζήτησης. Η διαδικασία αυτή περιλαμβάνει συνήθως την δημιουργία ενός ανεστραμμένου αρχείου το οποίο περιέχει τις ανεστραμμένες λίστες όλων των όρων καθώς και την δημιουργία ενός ανάλογου λεξικού για την αντιστοίχηση των όρων στις λίστες τους.

Το κύριο πρόβλημα της δημιουργίας ενός ανεστραμμένου ευρετηρίου είναι το γεγονός πως ο όγκος των δεδομένων που πρέπει να διαχειριστούμε δεν μπορεί να διατηρηθεί στην κύρια μνήμη, καθώς το μέγεθος των συλλογών είναι συχνά τάξης μεγέθους μεγαλύτερο από την κύρια μνήμη. Για τον λόγο αυτό, οι περισσότερες σύγχρονες τεχνικές δεικτοδοτούν τα κείμενα μιας συλλογής ανά δέσμες μέχρι να εξαντληθεί η διαθέσιμη μνήμη, και κάθε φορά που εξαντλείται η μνήμη αποθηκεύουν τα νέα δεδομένα στο δίσκο, συνενώνοντας τις πληροφορίες από τα νέα κείμενα που δεικτοδοτήθηκαν με το υπάρχον ευρετήριο στο δίσκο.

Κάποιες βασικές αρχές που ακολουθούν συνήθως οι μέθοδοι δεικτοδότησης, κυρίως για λόγους απόδοσης, είναι οι ακόλουθες:

- σε κάθε κείμενο της συλλογής ανατίθεται ένα μονοτονικά αυξανόμενο αριθμητικό αναγνωριστικό, ξεκινώντας από το 1
- οι ανεστραμμένες λίστες αποθηκεύονται στο δίσκο *συνεχόμενες*, με την έννοια πως τα bytes κάθε λίστας βρίσκονται συνεχόμενα αποθηκευμένα στο δίσκο. Συνήθως οι λίστες αποθηκεύονται στο δίσκο σε λεξικογραφική διάταξη
- το λεξικό περιλαμβάνει όλους τους όρους που εμφανίζονται στα κείμενα της συλλογής και αποθηκεύεται σε μία δυναμικά επεκτάσιμη δομή, όπως το B-tree
- μία ανεστραμμένη λίστα ενός όρου t αποτελείται από ένα σύνολο ζευγών $\langle d,\ f_{d,t}\rangle$ (όπου d ένα κείμενο στο οποίο εμφανίζεται ο t και $f_{d,t}$ το πλήθος των εμφανίσεων του t στο d), πιθανόν συμπεριλαμβανομένων και των θέσεων εμφάνισης του t στο d – οπότε και η λίστα αποτελείται από πλειάδες της μορφής $\langle d;\ f_{d,t};\ pos_1, pos_2, ..., pos_{f_{d,t}}\rangle$
- το λεξικό μπορεί να προεπεξεργάζεται, χρησιμοποιώντας λεξικογραφική αποκοπή ή λίστα απαγορευμένων λέξεων

Στη συνέχεια παρουσιάζουμε συνοπτικά την εξέλιξη των μεθόδων δεικτοδότησης, από τις πρώτες και πιο απλές μεθόδους (οι οποίες έχουν αρκετά μειονεκτήματα, όπως μεγάλες απαιτήσεις σε μνήμη και προσωρινό χώρο στο δίσκο, καθώς και προβλήματα κλιμάκωσης) έως την μέθοδο που θεωρείται πλέον η πιο αποδοτική για την δεικτοδότηση στατικών συλλογών.

3.2 Μέθοδοι δημιουργίας ευρετηρίων για στατικές συλλογές κειμένων

3.2.1 Απλή αντιστροφή στη μνήμη

Θεωρούμε ένα ανεστραμμένο αρχείο επιπέδου κειμένων, το οποίο σημαίνει πως για κάθε όρο διατηρούμε μόνο την πληροφορία του πόσες φορές εμφανίζεται σε κάθε κείμενο. Η βασική ιδέα της *απλής αντιστροφής στη μνήμη* είναι πως διατηρούμε το ανεστραμμένο ευρετήριο όχι σαν μία συλλογή λιστών, αλλά σαν έναν *πίνακα αντιστροφής*. Κάθε στήλη του πίνακα αναπαριστά ένα κείμενο και κάθε γραμμή μία ανεστραμμένη λίστα. Οι εγγραφές του πίνακα

αναπαριστούν τις εμφανίσεις των όρων: η τιμή στο κελί (m, n) του πίνακα είναι η συχνότητα $f_{m,n}$ του m-οστού όρου στο n-οστό κείμενο.

Ο πίνακας αυτός δημιουργείται με δύο περάσματα της συλλογής. Στο πρώτο πέρασμα υπολογίζεται το πλήθος M των διακριτών όρων της συλλογής και το πλήθος N των κειμένων. Έπειτα, δεσμεύεται ο κατάλληλος χώρος στη μνήμη για τον $M \times N$ πίνακα αντιστροφής. Στο δεύτερο πέρασμα ενημερώνουμε τον –μέχρι πρότινος άδειο– πίνακα αντιστροφής: για κάθε όρο t ο οποίος εξάγεται από το κείμενο d εντοπίζεται στον πίνακα η αντίστοιχη γραμμή r του όρου και αυξάνεται η τιμή του κελιού (r, d) κατά ένα. Όταν τελειώσει και το δεύτερο πέρασμα, ο πίνακας διασχίζεται κατά γραμμές και δημιουργούνται οι ανεστραμμένες λίστες: για κάθε γραμμή συλλέγουμε τις μη-μηδενικές εγγραφές, κωδικοποιούμε την λίστα και έπειτα την αποθηκεύουμε στο δίσκο.

Τα μειονεκτήματα της μεθόδου αυτής είναι προφανή: απαιτεί δύο διασχίσεις της συλλογής και έχει πολύ μεγάλες απαιτήσεις σε μνήμη. Για παράδειγμα, για μία συλλογή 5Mb και χρησιμοποιώντας μόνο δύο bytes για κάθε συχνότητα, απαιτεί περισσότερα από 800Mb μνήμης.

Η απαιτούμενη μνήμη μπορεί να μειωθεί αποθηκεύοντας κάθε γραμμή του πίνακα σαν μία συνδεδεμένη λίστα, καθώς ο πίνακας είναι αρκετά αραιός. Κάθε κόμβος της λίστας αναπαριστά μία $\langle d, f_{d,t} \rangle$ εμφάνιση και χρησιμοποιεί 12 bytes (8 bytes για την εμφάνιση και επιπλέον 4 bytes για τον δείκτη). Παρόλα αυτά, για μία συλλογή 5Gb απαιτούνται 1.6Gb μνήμης, και μάλιστα χωρίς να διατηρούμε την πληροφορία των θέσεων εμφανίσεων των όρων μέσα στα κείμενα (ευρετήριο επιπέδου λέξεων), δημιουργώντας έτσι προβλήματα κλιμάκωσης.

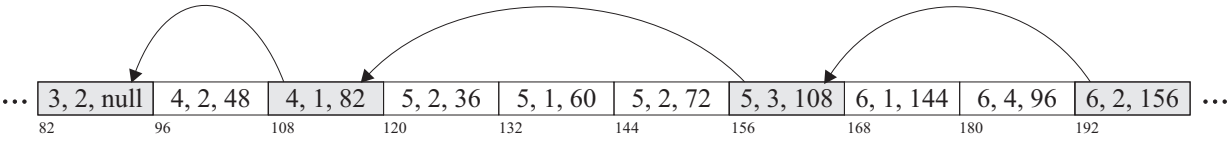
Για το υπόλοιπο της ενότητας θεωρούμε πως η διαδικασία δημιουργίας του ανεστραμμένου αρχείου λαμβάνει σαν είσοδο μία ροή από εμφανίσεις της μορφής $\langle t, d, f_{d,t} \rangle$ (ή της μορφής $\langle d;\ f_{d,t};\ pos_1, pos_2, ..., pos_{f_{d,t}} \rangle$ αν συμπεριλαμβάνονται και οι θέσεις εμφανίσεων των όρων στα κείμενα), οι οποίες εξάγονται από την διαδικασία λεκτικής ανάλυσης των κειμένων και παρέχονται στη διαδικασία δημιουργίας του ανεστραμμένου αρχείου.

3.2.2 Αντιστροφή στο δίσκο

Για να μειωθούν οι απαιτήσεις σε μνήμη, η μέθοδος *αντιστροφής στο δίσκο* συσσωρεύει τις εμφανίσεις των όρων στο δίσκο και όχι στην κύρια μνήμη, ενώ επεξεργάζεται κάθε κείμενο μόνο μία φορά. Η μέθοδος αυτή δημιουργεί στο δίσκο ένα *αρχείο εμφανίσεων*, το οποίο περιέχει πολλαπλές συνδεδεμένες λίστες, μία για κάθε όρο. Κατά την διαδικασία

της δεικτοδότησης, οι εμφανίσεις αποθηκεύονται στο δίσκο και συνδέονται μέσω δεικτών με τις προηγούμενες αποθηκευμένες εμφανίσεις του ίδιου όρου. Όταν η διαδικασία αυτή ολοκληρωθεί για όλα τα κείμενα, δημιουργείται ένα νέο αρχείο στο δίσκο, το οποίο τελικά θα περιέχει τις ανεστραμμένες λίστες. Για να δημιουργηθούν οι τελικές ανεστραμμένες λίστες χρησιμοποιούμε το αρχείο εμφανίσεων: οι συνδεδεμένες λίστες των όρων διασχίζονται κατά λεξικογραφική διάταξη, και για κάθε όρο οι εμφανίσεις του συλλέγονται, κωδικοποιούνται και τέλος αποθηκεύονται στο νέο αρχείο.

Για να μπορούμε να συνδέουμε στο δίσκο αποδοτικά τις εμφανίσεις του ίδιου όρου, απαιτείται να διατηρούμε για κάθε όρο την θέση μέσα στο αρχείο εμφανίσεων στην οποία αποθηκεύτηκε η τελευταία εμφάνιση του. Η πληροφορία αυτή διατηρείται στο λεξικό. Έχοντας πλέον αυτή την πληροφορία, κάθε φορά που μία νέα εμφάνιση $\langle t,\, d,\, f_{d,t}\rangle$ εξάγεται από την διαδικασία λεκτικής ανάλυσης, κάνουμε μία αναζήτηση στο λεξικό για τον όρο t. Με την βοήθεια του λεξικού βρίσκουμε την θέση p μέσα στο αρχείο της τελευταίας εμφάνισης του t, και η νέα εμφάνιση $\langle d,\, f_{d,t},\, p\rangle$ προστίθεται στο τέλος του αρχείου εμφανίσεων.



Σχήμα 3.1: Δομή των εμφανίσεων στο δίσκο.

Όταν ολοκληρωθεί η παραπάνω διαδικασία για όλα τα αρχεία της συλλογής, δημιουργούμε ένα νέο αρχείο, το οποίο θα αποτελέσει το τελικό ανεστραμμένο αρχείο. Όμως οι λίστες εμφανίσεων δεν είναι αποθηκευμένες βάσει λεξικογραφικής διάταξης στο αρχείο εμφανίσεων, ούτε οι διάφορες εμφανίσεις ενός όρου είναι αποθηκευμένες συνεχόμενα μέσα στο αρχείο. Για παράδειγμα, στο Σχήμα 3.1 παρουσιάζεται η δομή των εμφανίσεων ενός όρου t στο δίσκο, οι οποίες αντιστοιχούν στην ανεστραμμένη λίστα $\langle 3,2\rangle\langle 4,1\rangle\langle 5,3\rangle\langle 6,2\rangle$. Έτσι, απαιτείται ένα βήμα προεπεξεργασίας, όπου αρχικά διασχίζουμε το λεξικό με αποτέλεσμα να εξάγουμε όλα τα ζεύγη $\langle$*term, position of last posting*$\rangle$ σε λεξικογραφική διάταξη. Έπειτα, έχοντας πλέον για κάθε όρο την διεύθυνση της τελευταίας αποθηκευμένης εμφάνισης του στο αρχείο εμφανίσεων, και επιπλέον, έχοντας κάθε εμφάνιση συνδεδεμένη με την αμέσως προηγούμενη της για τον ίδιο όρο, η λίστα εμφανίσεων κάθε όρου μπορεί να διασχιστεί από το τέλος προς την αρχή και να συλλεχθούν έτσι όλες οι εμφανίσεις του. Στη συνέχεια δημιουργείται η ανεστραμμένη λίστα του όρου, κωδικοποιείται και τέλος αποθηκεύεται στο νέο αρχείο.

Η μέθοδος αυτή έχει τα μειονέκτηματα πως (i) απαιτεί ολόκληρο το λεξικό να διατηρείται στη μνήμη σε όλη την διάρκεια της λεκτικής ανάλυσης, (ii) χρησιμοποιεί αρκετά μεγάλο προσωρινό χώρο στο δίσκο (καθώς για κάθε εμφάνιση απαιτεί επιπλέον 4 bytes για τον δείκτη στην προηγούμενη εμφάνιση), και (iii) ο μεγάλος αριθμός τυχαίων προσπελάσεων του δίσκου κατά το βήμα της προεπεξεργασίας αυξάνει κατά πολύ τους χρόνους δεικτοδότησης, με αποτέλεσμα η μέθοδος να μην έχει πρακτική χρήση.

3.2.3 Αντιστροφή στη μνήμη μέσω δύο περασμάτων

Η μέθοδος *αντιστροφής στη μνήμη μέσω δύο περασμάτων* προτάθηκε λόγω των αυξημένων απαιτήσεων σε προσωρινό αποθηκευτικό χώρο της μεθόδου αντιστροφής στο δίσκο. Στη μέθοδο αυτή η αντιστροφή της συλλογής λαμβάνει χώρα στη μνήμη απαιτώντας πολύ λιγότερο επιπλέον χώρο στο δίσκο, κάνοντας όμως δύο περάσματα της συλλογής.

Στο πρώτο πέρασμα, δημιουργείται το λεξικό και μαζεύονται διάφορα στατιστικά από την συλλογή κειμένων. Συγκεκριμένα, υπολογίζεται το πλήθος των κειμένων N, το πλήθος των διακριτών όρων n και η συχνότητα εμφάνισης f_t κάθε όρου t. Χρησιμοποιώντας τις προηγούμενες πληροφορίες, δεσμεύεται στη μνήμη ένα τμήμα στο οποίο θα αποθηκευτούν όλες οι εμφανίσεις, οι οποίες θα εξαχθούν κατά το δεύτερο πέρασμα. Το τμήμα αυτό χωρίζεται σε n *διαμερίσεις*, μία για κάθε όρο, όπου σε κάθε διαμέριση θα αποθηκευτούν οι εμφανίσεις του αντίστοιχου όρου. Εφόσον το πλήθος f_t των εμφανίσεων κάθε όρου t καθορίστηκε στο πρώτο πέρασμα, το συνολικό μέγεθος του τμήματος και τα όρια μεταξύ των διαμερίσεων μπορούν εύκολα να υπολογιστούν. Κατά το δεύτερο πέρασμα εξάγονται οι εμφανίσεις από τα κείμενα και αποθηκεύονται στις κατάλληλες διαμερίσεις. Στο τέλος οι διαμερίσεις αποθηκεύονται στο δίσκο, με κάθε διαμέριση να αντιστοιχεί στην ανεστραμμένη λίστα ενός όρου.

Σε σχέση με την μέθοδο αντιστροφής στο δίσκο, ο προσωρινός χώρος στο δίσκο μειώνεται πολύ καθώς εμφανίσεις που ανήκουν στον ίδιο όρο δεν χρειάζεται να συνδεθούν μεταξύ τους απαιτώντας έναν επιπλέον δείκτη. Παρ' όλα αυτά, αν δεν χρησιμοποιηθεί κατάλληλη συμπίεση, απαιτούνται τεράστια μεγέθη μνήμης.

Μία λύση για την περίπτωση που η απαιτούμενη μνήμη είναι μεγαλύτερη από την πραγματική μνήμη του συστήματος είναι να επιτρέψουμε στον αλγόριθμο να χρησιμοποιήσει περισσότερη μνήμη από την πραγματική, αφήνοντας στο λειτουργικό σύστημα την διαχείριση της εικονικής μνήμης. Όμως, οι συχνές τυχαίες προσπελάσεις στο δίσκο που απαιτούνται για την εναλλαγή τμημάτων της μνήμης μεταξύ δίσκου και κύριας μνήμης αυξάνουν δραματικά

τους χρόνους δεικτοδότησης.

Συμπίεση

Για να μειώσουμε το μέγεθος της απαιτούμενης μνήμης, οι εμφανίσεις αποθηκεύονται *κατευθείαν κωδικοποιημένες* στη μνήμη. Αυτό συνεπάγεται πως κατά το πρώτο πέρασμα πρέπει να υπολογιστεί το ακριβές πλήθος από bits που απαιτούνται για την αποθήκευση κάθε κωδικοποιημένης λίστας εμφανίσεων. Όμως το ακριβές πλήθος από bits εξαρτάται από τα πραγματικά νούμερα κάθε λίστας που αντιστοιχούν στα αναγνωριστικά των κειμένων, πληροφορία η οποία δεν μπορεί να συλλεχθεί κατά το πρώτο πέρασμα. Για τον σκοπό αυτό υπολογίζεται ένα άνω όριο βάσει της εκάστοτε μεθόδου κωδικοποίησης που χρησιμοποιείται για να κωδικοποιήσουμε τις εμφανίσεις. Ο αλγόριθμος παρουσιάζεται στο Σχήμα 3.2.

1. Make an initial pass over the collection:
   For each term t count its document freqeuncy f_t and determine an upper bound u_t on the length of the inverted list for t
2. Allocate memory:
   Allocate an in-memory array of $\sum_t u_t$ bytes, and, for each term t create a pointer c_t to the start of a corresponding partition of u_t bytes
3. Process the collection a second time:
   For each document d, and for each term t in d, append a code representing d and $f_{d,t}$ at c_t, and update c_t.
4. Make a sequential pass over the in-memory index that has been constructed and create the inverted file:
   For each term t, copy the f_t representations of the $\langle d, f_{d,t} \rangle$ pointers from the allocated u_t bytes to the inverted file, compressing it if desired.

Σχήμα 3.2: Αλγόριθμος αντιστροφής στη μνήμη μέσω δύο περασμάτων.

Αποθηκεύοντας τις εμφανίσεις απευθείας κωδικοποιημένες στη μνήμη μειώνεται σημαντικά το μέγεθος της απαιτούμενης μνήμης, αλλά παρ' όλα αυτά η δεικτοδότηση μεγάλων συλλογών κειμένων εξακολουθεί να μην είναι πρακτική.

Δέσμευση διαμερίσεων στο δίσκο

Ένας τρόπος να βελτιώσουμε την ικανότητα κλιμάκωσης της μεθόδου αντιστροφής στη μνήμη βάσει δύο περασμάτων είναι να δεσμεύσουμε το διάνυσμα με τις διαμερίσεις για τις εμφανίσεις όχι στη μνήμη αλλά στο δίσκο, και να χωρίσουμε την συλλογή σε ένα σύνολο από *δέσμες* αρχείων. Το μέγεθος κάθε δέσμης επιλέγεται να είναι αρκετά μικρό

ώστε να μπορεί να αντιστραφεί στη μνήμη χρησιμοποιώντας την μέθοδο αντιστροφής που περιγράφηκε παραπάνω.

Κάθε δέσμη αρχείων αντιστρέφεται στη μνήμη χρησιμοποιώντας την μέθοδο που περιγράφεται στο Σχήμα 3.2. Όταν τελειώσει η αντιστροφή της εκάστοτε δέσμης αρχείων, συγχωνεύουμε τις εμφανίσεις που συλλέχθηκαν στη μνήμη με το διάνυσμα διαμερίσεων στο δίσκο. Το διάνυσμα διαβάζεται στη μνήμη σειριακά, μπλοκ ανά μπλοκ, και έπειτα τα μπλοκς ξαναγράφονται πίσω στο δίσκο αφού προσθέσουμε τις νέες εμφανίσεις από την μνήμη. Οι διαμερίσεις από την μνήμη για την τρέχουσα δέσμη αρχείων απελευθερώνονται, και προχωράμε στην επόμενη δέσμη. Στο τελευταίο βήμα, όλος ο αχρησιμοποίητος χώρος[1] μεταξύ των διαμερίσεων αφαιρείται και οι ανεστραμμένες λίστες αποκωδικοποιούνται και κωδικοποιούνται χρησιμοποιώντας την τελική μέθοδο κωδικοποίησης.

Η παραλλαγή αυτή της μεθόδου απαιτεί την συλλογή περισσότερων πληροφοριών κατά το πρώτο πέρασμα της συλλογής. Για κάθε δέσμη κειμένων, το πλήθος των κειμένων της και οι συχνότητες εμφάνισης f_t των όρων μέσα στα κείμενα της συγκεκριμένης δέσμης πρέπει να καθοριστούν, έτσι ώστε να μπορούν να υπολογιστούν τα όρια μεταξύ των διαμερίσεων των διαφόρων όρων στη μνήμη για την αντιστροφή της εκάστοτε δέσμης.

3.2.4 Αντιστροφή βάσει ταξινόμησης

Η μέθοδος της *αντιστροφής βάσει ταξινόμησης* (sort-based inversion) σχεδιάστηκε ώστε να μπορεί να λειτουργεί με περιορισμένα μεγέθη μνήμης ενώ ταυτόχρονα χρησιμοποιεί ελάχιστο προσωρινό χώρο στο δίσκο.

Αρχικά συλλέγουμε στη διαθέσιμη μνήμη τις εμφανίσεις που παράγονται από την διαδικασία λεκτικής ανάλυσης. Όταν το συνολικό μέγεθος των εμφανίσεων στη μνήμη ξεπεράσει ένα κατώφλι, οι εμφανίσεις ταξινομούνται και συμπιέζονται, και έπειτα γράφονται σε ένα προσωρινό αρχείο σαν ένα *τμήμα* (segment). Τα προηγούμενα βήματα (συλλογή εμφανίσεων μέχρι να εξαντληθεί η μνήμη, ταξινόμηση, συμπίεση και αποθήκευση σαν τμήμα) επαναλαμβάνονται μέχρι η διαδικασία λεκτικής ανάλυσης να επεξεργαστεί όλα τα αρχεία. Στο τελευταίο βήμα, τα τμήματα από το προσωρινό αρχείο συνδυάζονται μέσω μίας διαδικασίας πολλαπλής συγχώνευσης και παράγονται οι τελικές ανεστραμμένες λίστες.

Όσον αφορά τη διαχείριση της μνήμης, μπορεί να επιτευχθεί πολύ καλή συμπίεση των εμφανίσεων στη μνήμη με την αντικατάσταση της αλφαριθμητικής αναπαράστασης κάθε

[1] ο αχρησιμοποιήτος χώρος μεταξύ των διαμερίσεων προκύπτει λόγω του ότι για κάθε όρο δεσμεύουμε περισσότερα bytes από όσα χρειάζονται για την ανεστραμμένη λίστα του, καθώς υπολογίζουμε ένα άνω όριο του μεγέθους της διαμέρισης.

όρου t με ένα μοναδικό αριθμητικό αναγνωριστικό t'. Για το σκοπό αυτό, το λεξικό περιέχει για κάθε όρο t και το αριθμητικό αναγνωριστικό του t'. Η απόδοση των αριθμών αυτών στους όρους δεν γίνεται βάσει της λεξικογραφικής διάταξης των όρων αλλά βάσει της σειράς με την οποία εμφανίζονται κατά την επεξεργασία των κειμένων. Έτσι, με την χρήση των αριθμητικών αναγνωριστικών η μορφή των εμφανίσεων είναι πλέον της μορφής $\langle t', d, f_{d,t} \rangle$.

Όταν η μνήμη εξαντληθεί, πριν να επεξεργαστούμε τα επόμενα κείμενα, οι εμφανίσεις ταξινομούνται σε αύξουσα διάταξη με πρωτεύον κλειδί το αριθμητικό αναγνωριστικό του όρου t' και δευτερεύον κλειδί το αναγνωριστικό κειμένου d. Έπειτα οι εμφανίσεις συμπιέζονται, χρησιμοποιώντας την τεχνική των διαφορών μεταξύ των αναγνωριστικών των όρων και των αναγνωριστικών των κειμένων. Τέλος, το νέο τμήμα γράφεται στο δίσκο, στο προσωρινό αρχείο.

Όταν η επεξεργασία όλων των κειμένων ολοκληρωθεί, τα K τμήματα τα οποία έχουν συγκεντρωθεί στο δίσκο συγχωνεύονται για να λάβουμε το τελικό ανεστραμμένο αρχείο. Μία K-δική συγχώνευση απαιτεί την προσπέλαση κάθε τμήματος μόνο μία φορά, χρησιμοποιώντας ένα μικρό τμήμα προσωρινής μνήμης –*buffer*– μεγέθους B για κάθε τμήμα, και βρίσκοντας σε κάθε βήμα την μικρότερη (βάσει αναγνωριστικού όρου και αναγνωριστικού κειμένου) εμφάνιση από τις κεφαλές των K buffers. Όταν μία εμφάνιση από έναν buffer χρησιμοποιηθεί απομακρύνεται από την κεφαλή του, και όταν ένας buffer αδειάσει τότε τον γεμίζουμε με ένα νέο μπλοκ εμφανίσεων (μεγέθους B) από το αντίστοιχο τμήμα του.

Μία προσεκτική *επιτόπου* (in-situ) μέθοδος συγχώνευσης [18] επιτρέπει την αποθήκευση των τελικών ανεστραμμένων λιστών που παράγονται κατά την συγχώνευση πίσω στο προσωρινό αρχείο, χωρίς έτσι να απαιτείται επιπλέον χώρος στο δίσκο: οι ανεστραμμένες λίστες που προκύπτουν από την συγχώνευση μπορούν να γραφούν στην θέση των μπλοκ εμφανίσεων που έχουν ήδη χρησιμοποιηθεί στη συγχώνευση. Έτσι δεν απαιτείται καθόλου επιπλέον χώρος στο δίσκο. Σε αυτή την περίπτωση απαιτείται ένα επιπλέον βήμα στο τέλος της διαδικασίας, στο οποίο αντιμεταθέτουμε τα μπλοκς του ανεστραμμένου αρχείου, ώστε να εξασφαλίσουμε πως η φυσική διάταξη των ανεστραμμένων λιστών στο δίσκο αντιστοιχεί στην λογική διάταξη των αναγνωριστικών των όρων τους.

Η μέθοδος αυτή παράγει ένα ανεστραμμένο αρχείο στο οποίο οι λίστες δεν είναι αποθηκευμένες στο δίσκο βάσει της λεξικογραφικής τους διάταξης, αλλά βρίσκονται αποθηκευμένες σε αύξουσα διάταξη ως προς τα αναγνωριστικά των όρων τους, τα οποία ανατίθενται στους όρους με τυχαία σειρά (ανάλογα με την σειρά πρώτης εμφάνισης των όρων).

3.2.5 Αντιστροφή βάσει συγχώνευσης

Η μέθοδος αυτή περιέχει μία σειρά από τεχνικές που χρησιμοποιούνται στις περισσότερες σημερινές μεθόδους αντιστροφής. Η βασική ιδέα της μεθόδου είναι η απευθείας συμπίεση των εμφανίσεων στη μνήμη, καθώς αυτές παρέχονται από την διαδικασία λεκτικής ανάλυσης, και η διατήρηση τους στη μνήμη σε συμπιεσμένη μορφή. Για το σκοπό αυτό, σε κάθε όρο στο λεξικό ανατίθεται ένα *δυναμικό διάνυσμα* από bits, στο οποίο συλλέγονται οι εμφανίσεις του όρου σε συμπιεσμένη μορφή. Για να είμαστε σε θέση να χρησιμοποιούμε αποδοτικά την μέθοδο της κωδικοποίησης διαφορών, πρέπει να γνωρίζουμε για κάθε όρο το τελευταίο αναγνωριστικό κειμένου που εισήχθη στο διάνυσμα του, ώστε να μπορούμε να υπολογίσουμε την διαφορά του νέου αναγνωριστικού από το ακριβώς προηγούμενο που εισήχθη. Έτσι, για κάθε όρο t κρατάμε επίσης και την πληροφορία του τελευταίου αναγνωριστικού d_{last} που εισάγαμε στο διάνυσμα του.

Χρησιμοποιώντας τα διανύσματα που αναφέρθηκαν παραπάνω, τα βασικά βήματα της μεθόδου είναι τα εξής: για κάθε εμφάνιση $\langle t,\, d,\, f_{d,t}\rangle$ που παράγεται από την διαδικασία λεκτικής ανάλυσης γίνεται μία αναζήτηση για τον όρο t στο λεξικό. Αν ο όρος δεν υπάρχει στο λεξικό, τον εισάγουμε και αρχικοποιούμε το διάνυσμα του. Έπειτα συμπιέζουμε τη νέα εμφάνιση του (κωδικοποιώντας το $\langle d - d_{last},\, f_{d,t}\rangle$) και την εισάγουμε στο διάνυσμα. Τέλος, ενημερώνουμε το πεδίο d_{last} του όρου στο λεξικό ($d_{last} \leftarrow d$). Η διαδικασία αυτή επαναλαμβάνεται μέχρι να εξαντληθεί η μνήμη.

Όταν η μνήμη εξαντληθεί, δημιουργείται ένα νέο τμήμα στο δίσκο το οποίο περιέχει όλες τις εμφανίσεις που έχουν συλλεχθεί μέχρι στιγμής στη μνήμη: αρχικά, οι όροι του λεξικού προσπελαύνονται βάσει *λεξικογραφικής διάταξης*, και για κάθε όρο ανακτούμε το διάνυσμα των bits του. Έπειτα ο όρος γράφεται στο δίσκο, ακολουθούμενος από το διάνυσμα με τις κωδικοποιημένες εμφανίσεις του. Τέλος, απελευθερώνεται η μνήμη που δεσμεύει το λεξικό. Η διαδικασία επαναλαμβάνεται μέχρι να ολοκληρωθεί η επεξεργασία όλων των κειμένων.

Όταν τελικά ολοκληρωθεί η επεξεργασία όλων των κειμένων της συλλογής, όλα τα τμήματα που έχουν δημιουργηθεί στο δίσκο συγχωνεύονται για να παραχθεί το τελικό ανεστραμμένο αρχείο. Για την συγχώνευση των τμημάτων χρησιμοποιείται ο αλγόριθμος με τους K buffers που περιγράφηκε στην μέθοδο της αντιστροφής βάσει ταξινόμησης. Επίσης, χρησιμοποιείται και η μέθοδος για την επιτόπου αποθήκευση του ανεστραμμένου αρχείου πίσω στο προσωρινό αρχείο, και άρα στο τέλος της διαδικασίας απαιτείται ένα επιπλέον βήμα για την αντιμετάθεση των μπλοκς του ανεστραμμένου αρχείου, ώστε οι λίστες να βρίσκονται αποθηκευμένες βάσει της λεξικογραφικής τους διάταξης.

```
1.  Until all documents have been processed:
2.      Initialize an in-memory index, using dynamic structures for the
        vocabularies and a static coding scheme for inverted lists;
        store lists either in dynamically resized arrays or in linked
        blocks.
3.      Read documents and insert ⟨d, f_{d,t}⟩ pointers —compressed— into
        the in-memory index, continuing until all allocated memory is
        consumed
4.      When memory is full, flush this temporary index to disk
        -sorting postings by terms-, including its vocabulary
5.  Merge the set of partial indexes to form a single index, compressing
    the inverted lists if required.
```

Σχήμα 3.3: Αλγόριθμος αντιστροφής βάσει ταξινόμησης.

Το σημαντικότερο πλεονέκτημα της μεθόδου αυτής είναι πως δεν χρειάζεται να διατηρεί όλο το λεξικό στη μνήμη, καθώς κάθε φορά που εξαντλείται η μνήμη οι εμφανίσεις που έχουν συλλεχθεί γράφονται σαν ένα τμήμα στο προσωρινό αρχείο και η μνήμη του λεξικού απελευθερώνεται. Επίσης, σε αντίθεση με την αντιστροφή βάσει ταξινόμησης, το ανεστραμμένο αρχείο είναι λεξικογραφικά ταξινομημένο. Ένα μειονέκτημα της μεθόδου είναι πως σε κάθε τμήμα μαζί με τις εμφανίσεις ενός όρου πρέπει να αποθηκεύουμε και τον ίδιο τον όρο (ενώ πριν αποθηκεύαμε μόνο έναν αριθμό αντί για ένα αλφαριθμητικό). Λόγω του γεγονότος όμως πως διαδοχικοί όροι έχουν με μεγάλη πιθανότητα κοινό πρόθεμα, μπορούμε να χρησιμοποιήσουμε την μέθοδο της *ευθείας κωδικοποίησης* (front-coding) για την αποθήκευση των όρων: για κάθε όρο που θέλουμε να αποθηκεύσουμε, αποθηκεύουμε αντί αυτού μία τριάδα $\langle n_p, n_s, s\rangle$, όπου n_p το μήκος του κοινού προθέματος του όρου με τον προηγούμενο του όρο, n_s το μήκος του όρου χωρίς το κοινό πρόθεμα, και s οι χαρακτήρες του όρου χωρίς το κοινό πρόθεμα (για κάθε όρο δηλαδή αποθηκεύουμε δύο ακεραίους και το τμήμα του όρου που δεν είναι κοινό με τον προηγούμενο όρο). Χρησιμοποιώντας την τελευταία τεχνική κωδικοποίησης μπορούμε να μειώσουμε αρκετά τον επιπλέον αποθηκευτικό χώρο που απαιτείται.

Σύμφωνα με μελέτες [11] η μέθοδος αυτή είναι η πιο αποδοτική μέθοδος δεικτοδότησης για στατικές συλλογές κειμένων.

3.3 Μέθοδοι δημιουργίας ευρετηρίων για δυναμικές συλλογές κειμένων

Ένα ανεστραμμένο ευρετήριο μπορεί να κατασκευαστεί αρκετά γρήγορα χρησιμοποιώντας μία μέθοδο δημιουργίας ευρετηρίων για στατικές συλλογές: επεξεργαζόμαστε τα κείμενα

της στατικής συλλογής, κάνοντας βάσει της εκάστοτε μεθόδου ένα ή περισσότερα περάσματα από τα κείμενα της, και στο τέλος δημιουργούμε ένα ευρετήριο το οποίο αποτελείται από ένα ανεστραμμένο αρχείο και ένα λεξικό, με την βοήθεια των οποίων μπορούμε να αποτιμάμε ερωτήματα αναζήτησης.

Η πιο αποδοτική μέθοδος δημιουργίας ευρετηρίων για στατικές συλλογές φαίνεται να είναι η αντιστροφή στη μνήμη βάσει συγχώνευσης (Ενότητα 3.2.5), στην οποία μέθοδο οι εμφανίσεις συλλέγονται και συμπιέζονται απευθείας στη μνήμη, μέχρι να εξαντληθεί η μνήμη, οπότε και όλες οι εμφανίσεις της μνήμης γράφονται σαν ένα νέο τμήμα στο δίσκο. Η διαδικασία αυτή (συλλογή και συμπίεση εμφανίσεων μέχρι να εξαντληθεί η μνήμη, αποθήκευση τους στο δίσκο σαν ένα νέο τμήμα) επαναλαμβάνεται μέχρι να ολοκληρωθεί η επεξεργασία όλης της συλλογής. Στο τέλος, όλα τα τμήματα συνενώνονται για να παραχθεί το τελικό ανεστραμμένο αρχείο, το οποίο μπορεί να αποθηκευτεί πίσω στον προσωρινό χώρο που δεσμεύτηκε για τα διάφορα τμήματα, απαιτώντας έτσι μηδενικό επιπλέον χώρο.

Οι μέθοδοι δημιουργίας ευρετηρίων για στατικές συλλογές είναι πάρα πολύ αποδοτικές, αν είναι ανεκτή μία καθυστέρηση μεταξύ της άφιξης ενός νέου κειμένου και της στιγμής που αυτό γίνεται διαθέσιμο στα ερωτήματα αναζήτησης. Για παράδειγμα, στην περίπτωση του διαδικτύου, η συλλογή και δεικτοδότηση όλων των κειμένων σε καθημερινή βάση είναι ανεκτή, καθώς εξασφαλίζει πως κείμενα τα οποία έχουν δημιουργηθεί τουλάχιστον μία μέρα πριν (αλλά πιθανώς όχι τις τελευταίες ώρες) είναι διαθέσιμα στους χρήστες μέσω των αναζητήσεων, κάτι που τελικά είναι ανεκτό από τον χρήστη. Στις περιπτώσεις αυτές συνήθως η διαδικασία δημιουργίας του νέου ευρετηρίου εκτελείται στο παρασκήνιο, ενώ τα ερωτήματα εξακολουθούν να απαντώνται με το παλιό ευρετήριο. Όταν η το νέο ευρετήριο δημιουργηθεί τα ερωτήματα πλέον ανακατευθύνονται σε αυτό, ενώ το παλιό ευρετήριο διαγράφεται.

Υπάρχουν όμως πολλές περιπτώσεις στις οποίες τα νέα κείμενα πρέπει να είναι άμεσα προσπελάσιμα, το οποίο σημαίνει πως το ευρετήριο πρέπει να υποστηρίζει ερωτήματα και κατά την διάρκεια της δημιουργίας του (καθώς αυτή μπορεί να διαρκέσει αρκετή ώρα), ενώ πρέπει να είναι συνεχώς ενημερωμένο ώστε να συμπεριλαμβάνει στα αποτελέσματα και όσα κείμενα έχουν προστεθεί πρόσφατα.

Στις ενότητες που ακολουθούν θα μελετήσουμε τις σημαντικότερες μεθόδους δημιουργίας ευρετηρίων για δυναμικές συλλογές. Η βασική λογική όλων των μεθόδων είναι πως καθώς νέα κείμενα προστίθενται στη συλλογή, αναλύονται σε ένα σύνολο εμφανίσεων οι οποίες συλλέγονται συμπιεσμένες στη μνήμη, δημιουργώντας έτσι στη μνήμη ένα μικρό ευρετήριο για τα νέα κείμενα. Όποτε υποβάλλονται ερωτήματα αναζήτησης, συνδυάζουμε

τις λίστες από τον δίσκο με τις νέες εμφανίσεις από τη μνήμη για να απαντήσουμε στα ερωτήματα, συμπεριλαμβάνοντας πληροφορία τόσο για τα παλιά κείμενα όσο και για αυτά που μόλις προστέθηκαν. Όταν η μνήμη εξαντληθεί συγχωνεύουμε τις εμφανίσεις από τη μνήμη με το ευρετήριο στο δίσκο.

Θεωρούμε κυρίως την περίπτωση δυναμικών συλλογών στις οποίες έχουμε εισαγωγή νέων κειμένων. Η περίπτωση της διαγραφής κειμένων έχει μελετηθεί στα [8, 5, 9], ενώ η περίπτωση της τροποποίησης ενός κειμένου μπορεί να αντιμετωπιστεί σαν μία διαγραφή του κειμένου και μία εισαγωγή του τροποποιημένου κειμένου.

3.3.1 Ανακατασκευή ευρετηρίου

Η πιο απλοϊκή προσέγγιση για την υποστήριξη δυναμικών συλλογών είναι η κατασκευή ενός νέου ευρετηρίου κάθε φορά που νέα κείμενα προστίθενται στη συλλογή. Τα προβλήματα βέβαια της μεθόδου αυτής είναι προφανή: το κόστος είναι τεράστιο αν ανακατασκευάζουμε το ευρετήριο για κάθε νέο κείμενο που προστίθεται στη συλλογή, ενώ παράλληλα το ευρετήριο δεν μπορεί να χρησιμοποιηθεί κατά την διάρκεια της δημιουργίας του. Η λύση στο τελευταίο πρόβλημα είναι η διατήρηση του παλιού ευρετηρίου και η χρήση του για την αποτίμηση ερωτήσεων αναζήτησης κατά την διάρκεια της δημιουργίας του νέου ευρετηρίου, υπό την προϋπόθεση φυσικά πως ο επιπλέον χώρος για την διατήρηση ενός δεύτερου ευρετηρίου είναι ανεκτός. Για να μειώσουμε το κόστος ενημέρωσης του ευρετηρίου ανά αρχείο που προστίθεται, δημιουργούμε το νέο ευρετήριο περιοδικά και όχι κάθε φορά που προστίθεται ένα νέο αρχείο. Συγκεκριμένα, όσο προστίθενται νέα κείμενα οι εμφανίσεις τους εξάγονται και συλλέγονται στη μνήμη. Τόσο το ευρετήριο του δίσκου όσο και οι εμφανίσεις από τη μνήμη συνδυάζονται για την απάντηση ερωτημάτων. Όταν η μνήμη εξαντληθεί, δημιουργούμε ένα νέο ευρετήριο χρησιμοποιώντας πλέον όλα τα κείμενα της συλλογής (και τα καινούργια), και διαγράφουμε το παλιό ευρετήριο και τις εμφανίσεις από τη μνήμη. Ο αλγόριθμος παρουσιάζεται στο Σχήμα 3.4.

```
1. Postings are accumulated in main memory as documents are added to
   the collection
2. Once main memory is exhausted, build a new index from the current
   entire collection
3. The old index and in-memory postings are discarded, replaced by the
   the new index
```

Σχήμα 3.4: Αλγόριθμος ανακατασκευής ευρετηρίου.

Προφανώς το κόστος της μεθόδου αυτής είναι απαγορευτικό για μεγάλες συλλογές. Κάτι

τέτοιο όμως δεν συνεπάγεται πως η μέθοδος δεν έχει πρακτική αξία, καθώς σε περιπτώσεις που οι συλλογές είναι μικρές και οι οποίες δεικτοδοτούνται περιοδικά (π.χ. κάθε μέρα) ή μέθοδος αυτή έχει πολύ καλή απόδοση (για παράδειγμα, οι σύγχρονες μηχανές αναζήτησης του διαδικτύου επιλέγουν να ξαναχτίζουν το ευρετήριο ανά κάποιες ώρες, αντί να ενημερώνουν το υπάρχον ευρετήριο για τις νέες αλλαγές, καθώς αυτό είναι πιο αποδοτικό).

3.3.2 Επιτόπου ενημέρωση ευρετηρίου

Κάθε φορά που προστίθενται κάποια νέα κείμενα στη συλλογή, εξάγονται από αυτά ένα σύνολο εμφανίσεων. Προφανώς οι εμφανίσεις αυτές αναφέρονται σε ένα υποσύνολο των όρων του ευρετηρίου, και άρα μόνο ένα υποσύνολο των ανεστραμμένων λιστών του ευρετηρίου πρέπει να ενημερωθούν. Βάσει αυτής της παρατήρησης, η μέθοδος της *επιτόπου*[2] *ενημέρωσης* δεν δημιουργεί ένα νέο ευρετήριο κάθε φορά που προστίθενται νέα κείμενα, αλλά συγχωνεύει τις νέες εμφανίσεις που συλλέχθηκαν στη μνήμη με το υπάρχον ευρετήριο, προσθέτοντας τις νέες εμφανίσεις στο τέλος των ανεστραμμένων λιστών του ευρετηρίου στο δίσκο. Για να είναι αυτό εφικτό, κάθε φορά που μία λίστα ενός όρου t αποθηκεύεται στο δίσκο, δεσμεύουμε γι' αυτήν περισσότερο χώρο από όσο χρειάζεται, έτσι ώστε μελλοντικά να μπορούμε να προσθέσουμε νέες εμφανίσεις του t στο τέλος της λίστας του. Φυσικά, κάποιες φορές ο ελεύθερος χώρος στο τέλος μιας λίστας δεν θα επαρκεί για την αποθήκευση των νέων εμφανίσεων, και έτσι περιοδικά η λίστα θα πρέπει να μεταφέρεται σε μία νέα τοποθεσία στο δίσκο για να δημιουργηθεί περισσότερος χώρος για τις νέες εμφανίσεις.

Γενικά, υπάρχουν δύο είδη μεθόδων για την επιτόπου ενημέρωση: αυτές που διατηρούν ανά πάσα στιγμή κάθε λίστα συνεχόμενα αποθηκευμένη στο δίσκο [14] και αυτές που επιτρέπουν σε μία λίστα να βρίσκεται κατακερματισμένη σε διάφορα τμήματα του δίσκου, τα οποία είναι συνδεδεμένα μεταξύ τους [26]. Διατηρώντας τις λίστες πάντα συνεχόμενες στο δίσκο ελαχιστοποιούμε τον πλήθος των προσπελάσεων δίσκου που χρειάζονται για να ανακτήσουμε μία λίστα. Από την άλλη, επιτρέποντας σε μία λίστα να βρίσκεται κατακερματισμένη σε πολλά σημεία στο δίσκο μειώνουμε το κόστος της ενημέρωσης του ευρετηρίου, αποφεύγοντας τις μετακινήσεις των λιστών κάθε φορά που δεν υπάρχει αρκετός χώρος για τις νέες εμφανίσεις στο τέλος τους. Με λίγα λόγια, αναγκάζοντας τις λίστες να διατηρούνται συνεχόμενες βελτιώνουμε την απόδοση των ερωτημάτων αναζήτησης, αλλά αυξάνουμε το κόστος διατήρησης του ευρετηρίου καθώς συχνά ολόκληρες λίστες πρέπει να μετακινούνται σε μία νέα θέση στο δίσκο.

[2]δεν έχει κάποια σχέση με την *επιτόπου αποθήκευση* των τελικών ανεστραμμένων λιστών πίσω στο προσωρινό αρχείο, όπως αυτή περιγράφηκε στην Ενότητα 3.2.4

Η μέθοδος αυτή απαιτεί την ύπαρξη επιπλέον δομών για την διαχείριση του ελεύθερου χώρου στο ευρετήριο. Για το σκοπό αυτό χρησιμοποιούνται λίστες στις οποίες καταγράφονται τα ελεύθερα τμήματα στο δίσκο. Τα τμήματα αυτά ταξινομούνται στις λίστες βάσει της θέσης τους στο δίσκο, ώστε να μπορούμε να βρούμε γρήγορα με μία δυαδική αναζήτηση αν μία ανεστραμμένη λίστα στο δίσκο μπορεί να επεκταθεί χρησιμοποιώντας τον ελεύθερο χώρο που βρίσκεται μετά από αυτήν. Στην περίπτωση της μετακίνησης μίας λίστας σε μία νέα θέση, ή στην περίπτωση της δημιουργίας μιας νέας λίστας, χρησιμοποιείται ένας αλγόριθμος πρώτης τοποθέτησης για την εύρεση ενός κατάλληλου ελεύθερου τμήματος.

Ο αλγόριθμος της επιτόπου ενημέρωσης περιγράφεται στο Σχήμα 3.5. Ο παραπάνω αλγόριθμος προϋποθέτει πως οι νέες εμφανίσεις μπορούν να προστεθούν σε μία υπάρχουσα ανεστραμμένη λίστα χωρίς να χρειαστεί να αποκωδικοποιήσουμε τη λίστα. Για τον σκοπό αυτό, για κάθε λίστα διατηρούμε κάποιες επιπλέον πληροφορίες, όπως τον τελευταίο αριθμό που κωδικοποιήσαμε, το πλήθος των ελεύθερων bits στο τελευταίο byte κ.α..

```
1.  Postings are accumulated in main memory as documents are added to
    the collection
2.  Once main memory is exhausted, for each in-memory postings list:
3.     Determine how much free space follows the corresponding on-disk
       postings list
4.     If there is sufficient free space
5.        Append the in-memory postings list
6.     Else
7.        Determine a new disk location with sufficient space to hold
          the on-disk and in-memory postings lists, using a first-fit
          algorithm
8.        Read the on-disk postings list from its previous location
          and write it to the new location
9.        Append the in-memory postings to the new location
10.    Discard the in-memory postings and advance to the next in-memory
       postings list
```

Σχήμα 3.5: Αλγόριθμος επιτόπου ενημέρωσης.

Η μέθοδος φαίνεται αρκετά ελκυστική, καθώς κάθε φορά ενημερώνουμε μόνο εκείνες τις λίστες για τις οποίες συλλέξαμε κάποιες νέες εμφανίσεις. Ωστόσο, λόγω των συχνών μετακινήσεων των λιστών (και κυρίως, λόγω των μετακινήσεων των μεγάλων λιστών), αλλά και λόγω των πολύπλοκων δομών και αλγορίθμων που απαιτούνται τελικά για τη διαχείριση του ελεύθερου χώρου, η μέθοδος αυτή δεν φαίνεται να είναι και τόσο αποδοτική έναντι άλλων μεθόδων [14]: μέθοδοι οι οποίες βασίζονται στην *συγχώνευση* (κάθε φορά που εξαντλείται η μνήμη, συγχωνεύουν όλο το παλιό ευρετήριο με τις νέες εμφανίσεις από τη μνήμη δημιουργώντας ένα καινούργιο ευρετήριο) και τις οποίες μελετάμε στη συνέχεια

υπερτερούν σημαντικά της μεθόδου αυτής, παρόλο που διαχειρίζονται πολύ μεγαλύτερα μεγέθη δεδομένων (κάθε φορά που εξαντλείται η μνήμη ανακτούμε όλο ευρετήριο, και όχι μόνο τις λίστες στις οποίες θα προστεθούν νέες εμφανίσεις), λόγω ακριβώς της αποδοτικότερης χρήσης του δίσκου.

3.3.3 Άμεση συγχώνευση ευρετηρίου

Σύμφωνα με τις πιο αποδοτικές μεθόδους δεικτοδότησης στατικών συλλογών, τα νέα κείμενα αναλύονται σε ένα σύνολο εμφανίσεων οι οποίες συμπιέζονται και συλλέγονται στη μνήμη. Όταν η μνήμη εξαντληθεί, οι εμφανίσεις από την μνήμη συγκεντρώνονται και γράφονται στο δίσκο σαν ένα νέο τμήμα. Όταν ολοκληρωθεί η επεξεργασία όλων των κειμένων της συλλογής, συγχωνεύουμε όλα τα τμήματα από τον δίσκο για να δημιουργήσουμε το τελικό ανεστραμμένο αρχείο.

Μία απλή παραλλαγή της μεθόδου αυτής μπορεί να χρησιμοποιηθεί και για την περίπτωση δυναμικών συλλογών: καθώς νέα κείμενα προστίθενται στη συλλογή, αναλύονται σε ένα σύνολο εμφανίσεων οι οποίες συλλέγονται στη μνήμη. Όταν εξαντληθεί η μνήμη, συγχωνεύουμε τις νέες εμφανίσεις με το υπάρχον ανεστραμμένο αρχείο στο δίσκο, διασχίζοντας μία φορά το ευρετήριο και συγχωνεύοντας για κάθε όρο την λίστα του από τον δίσκο με τις νέες εμφανίσεις του από τη μνήμη. Οι λίστες που παράγονται μετά την συγχώνευση (ακόμα και αυτές στις οποίες δεν προστέθηκαν νέες εμφανίσεις) αποθηκεύονται σε ένα νέο αρχείο στο δίσκο, δημιουργώντας έτσι το τελικό ανεστραμμένο αρχείο. Τόσο οι λίστες από τον δίσκο όσο και οι λίστες από τη μνήμη διασχίζονται βάσει λεξικογραφικής διάταξης. Ο αλγόριθμος παρουσιάζεται στο Σχήμα 3.6.
Κάθε φορά που η μνήμη εξαντλείται, διασχίζουμε μία φορά το παλιό ευρετήριο, ανακτούμε κάθε λίστα και την γράφουμε στο καινούργιο ευρετήριο, προσθέτοντας πιθανώς τις νέες εμφανίσεις που έχουν συλλεχθεί στη μνήμη για την συγκεκριμένη λίστα. Παρόλο που η διαδικασία αυτή ανακτά και γράφει στο καινούργιο ευρετήριο κάθε λίστα του παλιού ευρετηρίου, ακόμα και αυτές για τις οποίες δεν συλλέχθηκαν νέες εμφανίσεις στη μνήμη, εν τούτοις η διαδικασία εκτελείται πολύ αποδοτικά καθώς το ευρετήριο προσπελάζεται σειριακά. Το πρόβλημα προφανώς όμως έγκειται στο γεγονός πως για κάθε ενημέρωση απαιτείται η διάσχιση ολόκληρου του ευρετηρίου.

Αν κατά την διάρκεια της ενημέρωσης του ευρετηρίου πρέπει να υποστηρίζεται η δυνατότητα απάντησης ερωτημάτων, τότε πρέπει να διατηρείται και το παλιό ευρετήριο μέχρι να ολοκληρωθεί η διαδικασία συγχώνευσης. Το μειονέκτημα προφανώς είναι πως το νέο

```
1.  Postings are accumulated in main memory as documents are added to
    the collection
2.  Once main memory is exhausted, for each in-memory postings list and
    on-disk postings list in lexicographical order:
3.     If the term of the in-memory postings list is smaller than the
       term of the on-disk postings list then
4.        Write the in-memory postings list to the new index
5.        Advance to the next in-memory postings list
6.     Else if the in-memory postings term is the same as the on-disk
       postings term, then
7.        Write the on-disk postings list followed by the in-memory
          postings list to the new index
8.        Advance to the next on-disk and in-memory postings list
9.     Else
10.       Write the on-disk postings list to the new index
11.       Advance to the next on-disk postings list
```

Σχήμα 3.6: Αλγόριθμος άμεσης συγχώνευσης.

ευρετήριο πρέπει να γραφεί σε μία νέα θέση στο δίσκο, και άρα απαιτείται διπλάσιος χώρος. Για να μειώσουμε τον επιπλέον χώρο, το ευρετήριο μπορεί να χωριστεί σε τμήμα και η συγχώνευση να γίνεται ανά τμήματα, αντικαθιστώντας κάθε φορά ένα παλιό τμήμα με το νέο.

3.3.4 Μη-συγχώνευση ευρετηρίου

Το σημαντικότερο μειονέκτημα της μεθόδου άμεσης συγχώνευσης είναι το γεγονός πως σε κάθε συγχώνευση πρέπει να ανακτήσουμε όλο το ευρετήριο για να δημιουργήσουμε το καινούργιο ευρετήριο. Η μέθοδος αυτή είναι αποδοτικότερη από την μέθοδο της επιτόπου ενημέρωσης λόγω της σειριακής ανάκτησης του ευρετηρίου, αλλά παρουσιάζει εντονότερα προβλήματα κλιμάκωσης (προφανώς καθώς μεγαλώνει η συλλογή πρέπει να διαβάζουμε περισσότερα δεδομένα ανά συγχώνευση).

Ένας απλός τρόπος για να μειώσουμε το κόστος συγχώνευσης είναι κάθε φορά που εξαντλείται η μνήμη να μην συγχωνεύουμε καθόλου τις νέες εμφανίσεις τις μνήμης με το ευρετήριο στο δίσκο, αλλά να τις αποθηκεύουμε στο δίσκο σαν ένα νέο ξεχωριστό τμήμα (*run*), το οποίο αποτελεί ένα μικρό ευρετήριο. Για να απαντήσουμε στα ερωτήματα αναζήτησης συνδυάζουμε όλα τα τμήματα, καθώς ένας όρος μπορεί να έχει εμφανίσεις σε πολλά τμήματα στο δίσκο. Η μέθοδος παρουσιάζεται στο Σχήμα 3.7.

Η μέθοδος αυτή προσφέρει το βέλτιστο χρόνο ενημέρωσης του ευρετηρίου, καθώς κάθε φορά που εξαντλείται η μνήμη απλά αποθηκεύουμε τα περιεχόμενα της μνήμης σε ένα τμήμα

```
1. Postings are accumulated in main memory as documents are added to
   the collection
2. Once main memory is exhausted, flush in-memory postings on disk,
   creating a new segment.
```

Σχήμα 3.7: Αλγόριθμος μη-συγχώνευσης ευρετηρίου.

στο δίσκο, χωρίς να κάνουμε καμία συγχώνευση. Προφανώς όμως το κόστος αναζήτησης αυξάνεται πάρα πολύ: για την ανάκτηση μιας λίστας απαιτούνται αρκετές προσπελάσεις στο δίσκο, καθώς μία λίστα μπορεί να είναι κατακερματισμένη σε πολλά τμήματα. Μάλιστα το κόστος αυτό αυξάνεται καθώς μεγαλώνει το μέγεθος της συλλογής, καθώς αυξάνεται και ο αριθμός των τμημάτων.

3.3.5 Γεωμετρική διαμέριση / Λογαριθμική συγχώνευση

Συγκρίνοντας τις δύο παραπάνω μεθόδους, την μέθοδο της άμεσης συγχώνευσης και την μέθοδο της μη-συγχώνευσης, θα μπορούσαμε να πούμε πως η πρώτη ουσιαστικά παρέχει την βέλτιστη απόδοση κατά την ανάκτηση των λιστών, διατηρώντας όλες τις λίστες συνεχόμενα αποθηκευμένες στο δίσκο (και μάλιστα σε λεξικογραφική διάταξη), ενώ η δεύτερη παρέχει την βέλτιστη απόδοση κατά την ενημέρωση του ευρετηρίου, αφού κάθε φορά που εξαντλείται η μνήμη απλά αποθηκεύει τις νέες εμφανίσεις της μνήμης σαν ένα νέο τμήμα στο δίσκο χωρίς να εκτελέσει κανενός είδους συγχώνευση. Ανάμεσα σε αυτές τις δύο περιπτώσεις υπάρχει μία νέα κατηγορία μεθόδων, οι οποίες προσπαθούν να διατηρήσουν μία καλή αναλογία μεταξύ του χρόνου ενημέρωσης του ευρετηρίου και του χρόνου αναζήτησης κειμένων.

```
1. Postings are accumulated in main memory as documents are added to
   the collection
2. Once main memory is exhausted, create a new segment on disk,
   flushing all in-memory postings lists
3. If a merge-condition is true (for example, the number of segments
   on disk is greater than P):
4.     Select a number of segments
5.     Merge them, creating a new bigger segment
6.     Discard the merged segments from disk
```

Σχήμα 3.8: Γενική μεθοδολογία των αλγορίθμων γεωμετρικής διαμέρισης και λογαριθμικής συγχώνευσης.

Στο Σχήμα 3.8 σκιαγραφείται η γενική μεθοδολογία των αλγορίθμων *γεωμετρικής διαμέρισης* [13] και *λογαριθμικής συγχώνευσης* [5]: παρόμοια με την μέθοδο της μη-συγχώνευσης,

οι εμφανίσεις συλλέγονται στη μνήμη καθώς νέα κείμενα προστίθενται στη συλλογή, ενώ κάθε φορά που εξαντλείται η μνήμη ένα νέο τμήμα δημιουργείται στο δίσκο το οποίο περιέχει όλες τις εμφανίσεις της μνήμης. Κάθε τμήμα αποτελεί ένα μικρό ευρετήριο και μπορεί να χρησιμοποιηθεί για την απάντηση των ερωτημάτων. Περιοδικά, επιλέγονται κάποια τμήματα τα οποία και συγχωνεύονται σε ένα μεγαλύτερο ενιαίο τμήμα, μειώνοντας έτσι τον αριθμό των τμημάτων που υπάρχουν στο σύστημα. Ο αλγόριθμος βάσει του οποίου επιλέγονται τα τμήματα που θα συγχωνευτούν διαφέρει στις δύο μεθόδους.

Το αποτέλεσμα των παραπάνω μεθόδων είναι πως σε κάθε χρονική στιγμή υπάρχει στο σύστημα ένας (μικρός) αριθμός από τμήματα, καθένα από τα οποία μπορεί να έχει διαφορετικό μέγεθος, ανάλογα με το πλήθος των τμημάτων από τη συγχώνευση των οποίων έχει προκύψει. Για την απάντηση ενός ερωτήματος συνδυάζονται όλα τα τμήματα του δίσκου, καθώς οι εμφανίσεις ενός όρου μπορεί να βρίσκονται σε οποιαδήποτε τμήματα – αν στο σύστημα υπάρχουν K τμήματα, τότε η ανάκτηση μιας λίστας απαιτεί το πολύ K μετακινήσεις της κεφαλής του δίσκου, καθώς οι εμφανίσεις της λίστας μπορεί να βρίσκονται και στα K τμήματα. Και οι δύο μέθοδοι διατηρούν το πλήθος K των τμημάτων που μπορεί υπάρχουν ταυτόχρονα στο σύστημα αρκετά χαμηλό, μειώνοντας έτσι τους χρόνους αναζήτησης. Για παράδειγμα, στην γεωμετρική διαμέριση ο αριθμός των τμημάτων που βρίσκονται στο σύστημα μπορεί να είναι σταθερός –έστω P– σε κάθε χρονική στιγμή, ενώ στην λογαριθμική συγχώνευση αν N το μέγεθος της συλλογής και M το μέγεθος της μνήμης, τότε το πλήθος των τμημάτων είναι $\log_k(\lfloor N/M \rfloor)$, όπου το k είναι παράμετρος του αλγορίθμου.

Έτσι, και οι δύο μέθοδοι έχουν σαν αποτέλεσμα τη μείωση του χρόνου συγχώνευσης, αποφεύγοντας σε κάθε συγχώνευση την ανάκτηση όλου του ευρετηρίου, ενώ παράλληλα συγχωνεύοντας περιοδικά τα τμήματα του δίσκου φράσσουν τον μέγιστο αριθμό από τμήματα που μπορεί να υπάρχουν στο σύστημα, διατηρώντας έτσι χαμηλούς τους χρόνους αναζήτησης.

3.3.6 Υβριδική ενημέρωση

Η συγκεκριμένη μέθοδος βασίζεται στην παρατήρηση πως για τις μικρές λίστες είναι πιο αποδοτικό να τις ανακτήσουμε, σαν μέρος μιας μεγαλύτερης σειριακής ανάγνωσης πολλών λιστών, από το να κάνουμε μία μετακίνηση της κεφαλής του δίσκου για κάθε μικρή λίστα, ενώ για τις μεγάλες λίστες φαίνεται να είναι πιο αποδοτικό να κάνουμε μία μετακίνηση της κεφαλής του δίσκου (για να μετακινηθούμε για παράδειγμα στο τέλος της λίστας και να

προσθέσουμε κάποιες νέες εμφανίσεις) από το να ανακτήσουμε ολόκληρη τη λίστα.

Βάσει της παραπάνω παρατήρησης, στη μέθοδο *υβριδικής ενημέρωσης* οι ανεστραμμένες λίστες χωρίζονται βάσει του μήκους τους σε *μικρές* και *μεγάλες* λίστες. Όλες οι μικρές λίστες αποθηκεύονται μαζί σε τμήματα και ενημερώνονται χρησιμοποιώντας μία μέθοδο βάσει συγχώνευσης (π.χ. άμεση συγχώνευση, λογαριθμική συγχώνευση, γεωμετρική διαμέριση, κτλ), ενώ οι μεγάλες λίστες ενημερώνονται βάσει της επιτόπου ενημέρωσης, προσθέτοντας κάθε φορά τις νέες εμφανίσεις στο τέλος της υπάρχουσας λίστας. Υποθέτουμε (κυρίως για λόγους απλότητας) πως για κάθε μεγάλη λίστα δημιουργούμε ένα ξεχωριστό αρχείο στο σύστημα αρχείων. Κάθε φορά που θέλουμε να προσθέσουμε κάποιες νέες εμφανίσεις σε μία μεγάλη λίστα, προσθέτουμε τις εμφανίσεις αυτές στο τέλος του αντίστοιχου αρχείου (βασιζόμαστε στο σύστημα αρχείων για τον έλεγχο του κατακερματισμού των αρχείων, το οποίο όπως φαίνεται και στην Ενότητα 6.7 από τα πειράματα που εκτελέσαμε, δεν είναι και τόσο καλή τεχνική).

Ας υποθέσουμε πως η υβριδική ενημέρωση χρησιμοποιεί τη μέθοδο της άμεσης συγχώνευσης για τις μικρές λίστες. Τότε ο αλγόριθμος είναι αρκετά απλός: κάθε φορά που εξαντλείται η μνήμη και πρόκειται να συγχωνευθούν οι εμφανίσεις της μνήμης με το ευρετήριο, κάθε λίστα χαρακτηρίζεται ως μικρή ή μεγάλη ανάλογα με το αν το μέγεθος της ξεπερνά ένα κατώφλι T. Η διαδικασία συγχώνευσης εκτελείται κανονικά όπως και στην περίπτωση της άμεσης συγχώνευσης, με τη μόνη διαφορά πως όποτε συναντάται μία μεγάλη λίστα τότε οι εμφανίσεις τις προστίθενται στο τέλος του αρχείου που αντιστοιχεί στη συγκεκριμένη μεγάλη λίστα (αν το αρχείο δεν υπάρχει, δημιουργείται). Για τις μικρές λίστες δεν υπάρχουν αρχεία, οι εμφανίσεις τους προστίθενται στο ευρετήριο βάσει του αλγορίθμου άμεσης συγχώνευσης.

Ο παραπάνω αλγόριθμος μπορε εύκολα να τροποποιηθεί ώστε να χρησιμοποιείται ο αλγόριθμος της λογαριθμικής συγχώνευσης για τις μικρές λίστες, αντί για την άμεση συγχώνευση. Το μόνο πρόβλημα είναι πως στην περίπτωση αυτή υπάρχουν κάποιες δυσκολίες στο χαρακτηρισμό μιας λίστας ως μικρή η μεγάλη, καθώς σε κάθε διαδικασία συγχώνευσης συμμετέχει ένα υποσύνολο των τμημάτων του ευρετηρίου, χωρίς έτσι να είμαστε σε θέση να γνωρίζουμε τον συνολικό μέγεθος μιας λίστας και άρα να την χαρακτηρίσουμε ως μικρή ή μεγάλη (η λίστα μπορεί να έχει και εμφανίσεις σε τμήματα τα οποία δε συμμετέχουν στην τρέχουσα συγχώνευση). Μια απλή, αλλά σίγουρα όχι βέλτιστη προσέγγιση, είναι να διαχωρίσουμε τις λίστες σε μικρές και μεγάλες κατά την διάρκεια της πρώτης συγχώνευσης: την πρώτη φορά που εξαντλείται η μνήμη χωρίζουμε τις λίστες στις δύο κατηγορίες, ανάλογα με το αν το μέχρι στιγμής μέγεθος τους ξεπερνά κάποιο προκαθορισμένο κατώφλι T. Από

την στιγμή εκείνη και μετά καμία λίστα δεν μπορεί να αλλάξει κατηγορία.

Χρησιμοποιώντας τη μέθοδο της επιτόπου ενημέρωσης για τις μεγάλες λίστες και τη μέθοδο της λογαριθμικής συγχώνευσης για τις μικρές λίστες, η υβριδική μέθοδος ενημέρωσης φαίνεται να είναι η πιο αποδοτική μέθοδος δεικτοδότησης δυναμικών συλλογών [7, 3] .

Κεφαλαιο 4

Περιγραφη συστηματος

4.1 Βασική ιδέα

Μία από τις βασικές σχεδιαστικές αρχές των αλγορίθμων δημιουργίας ανεστραμμένων αρχείων, όπως αναφέρθηκε και στην Ενότητα 3.1, είναι η αποθήκευση των ανεστραμμένων λιστών *συνεχόμενα* στο δίσκο και η διατήρηση τους σε *λεξικογραφική διάταξη*.

Η αποθήκευση των λιστών συνεχόμενα στο δίσκο επιτρέπει την αποδοτικότερη ανάκτηση τους, καθώς αρκεί μόνο μία μετακίνηση της κεφαλής του δίσκου και έπειτα η ανάγνωση ενός αριθμού από bytes. Έτσι έχουμε μείωση των χρόνων αναζήτησης, κάθως για την αποτίμηση ενός ερωτήματος απαιτείται συνήθως η ανάκτηση των ανεστραμμένων λιστών όλων των όρων του. Από την άλλη, η λεξικογραφική διάταξη των όρων επιτρέπει την γρήγορη αποτίμηση *ερωτημάτων εύρους* (range queries), όπου πρέπει να ανακτήσουμε τις ανεστραμμένες λίστες ενός αριθμού από λεξικογραφικά διαδοχικούς όρους, ενώ επίσης επιτρέπει την ύπαρξη *μερικών λεξικών* (partial vocabularies) στα οποία δεν είναι ανάγκη να διατηρούμε όλους τους όρους αλλά μόνο ορισμένους. Για παράδειγμα, γνωρίζοντας τις

θέσεις στο δίσκο στις οποίες ξεκινάνε οι ανεστραμμένες λίστες των όρων όρου "cash" και "cat", και διατηρώντας τις λίστες αποθηκευμένες βάσει λεξικογραφικής διάταξης, μπορούμε εύκολα να βρούμε την λίστα του όρου "castle" ανακτώντας όλες τις λίστες που βρίσκονται στο δίσκο μεταξύ των λιστών των "cat" και "cash". Φυσικά, ο χώρος στο δίσκο μεταξύ των λιστών δύο διαδοχικών όρων στο λεξικό πρέπει να είναι αρκετά μικρός ώστε η αναζήτηση αυτή να γίνεται αποδοτικά.

Αρκετά νωρίς προτάθηκαν διάφορες μέθοδοι οι οποίες δεν τηρούσαν τον περιορισμό της διατήρησης των λιστών στο δίσκο σε λεξικογραφική διάταξη, με σκοπό την μείωση του χρόνου δεικτοδότησης των κειμένων. Για παράδειγμα, η μέθοδος αντιστροφής βάσει ταξινόμησης αντικαθιστά την αλφαριθμητική αναπαράσταση ενός όρου με ένα αριθμητικό αναγνωριστικό, το οποίο ανατίθεται τυχαία σε κάθε όρο (βάσει της σειράς της πρώτης εμφάνισης των όρων), και οι λίστες αποθηκεύονται στο δίσκο σε αύξουσα διάταξη ως προς το αριθμητικό αναγνωριστικό τους. Ουσιαστικά, εφόσον το ευρετήριο *δεν* υποστηρίζει ερωτήματα εύρους και θεωρήσουμε πως το κόστος διατήρησης *όλων* των όρων στο λεξικό είναι αποδεκτό, και από τη στιγμή που η μέθοδος δεικτοδότησης δεν απαιτεί την ανάκτηση των λιστών βάσει λεξικογραφικής διάταξης, δεν υπάρχει κάποιος σημαντικός λόγος για την διατήρηση των λιστών στο δίσκο σε λεξικογραφική διάταξη.

Επίσης, σχετικά νωρίς συναντάμε στην βιβλιογραφία μεθόδους οι οποίες διατηρούν κάθε λίστα αποθηκευμένη σε περισσότερα από ένα τμήματα στο δίσκο. Για παράδειγμα, διάφορες μέθοδοι [26, 2] επιτρέπουν στις λίστες (κυρίως στις μεγάλες λίστες) να αποθηκεύονται σε πολλαπλά τμήματα στο δίσκο, συνδεδεμένα μεταξύ τους, προκειμένου να μειωθεί το κόστος ενημέρωσης του ευρετηρίου. Νεότερες μέθοδοι [13, 5] επιτρέπουν την διαμέριση του ανεστραμμένου αρχείου σε έναν –σχετικά μικρό– αριθμό από μικρότερα ανεστραμμένα αρχεία, επιτρέποντας σε μία λίστα να διατηρεί τις εμφανίσεις της σε πολλές θέσεις στο δίσκο.

Παρατηρούμε δηλαδή πως, με απώτερο σκοπό την μείωση του κόστους ενημέρωσης του ευρετηρίου, οι αρχικές απαιτήσεις της συνέχειας των λιστών στο δίσκο και της λεξικογραφικής διάταξης τους σταδιακά έχουν πάψει να τηρούνται, επιτρέποντας σε μία λίστα να βρίσκεται σε πολλά τμήματα στο δίσκο.

Την ίδια στιγμή, αρκετές μελέτες έχουν προτείνει τον διαχωρισμό των λιστών βάσει του μεγέθους τους: ανάλογα με το αν το μέγεθος μιας λίστας ξεπερνάει ένα προκαθορισμένο κατώφλι, η λίστα θεωρείται μικρή ή μεγάλη αντίστοιχα. Οι μεγάλες λίστες ενημερώνονται βάσει της επιτόπου ενημέρωσης, ενώ οι μικρές λίστες βάσει μιας μεθόδου συγχώνευσης. Το μοντέλο αυτό φαίνεται να έχει πάρα πολύ καλά χαρακτηριστικά, αφού η υβριδική μέθοδος

η οποία υλοποιεί το παραπάνω μοντέλο φαίνεται να είναι η πιο αποδοτική μέχρι στιγμής μέθοδος [3].

Αδιαμφισβήτητα, τα μεγέθη των συλλογών κειμένων έχουν αυξηθεί πάρα πολύ τα τελευταία χρόνια, κάτι το οποίο έχει ως αποτέλεσμα οι μεγάλες λίστες να καταλαμβάνουν το μεγαλύτερο τμήμα ενός ανεστραμμένου αρχείου. Ταυτόχρονα, καθώς τα χαρακτηριστικά των σκληρών δίσκων έχουν βελτιωθεί αρκετά, φαίνεται να είναι αρκετά λογική η οργάνωση του ανεστραμμένου αρχείου σε μπλοκς προεπιλεγμένου μεγέθους, με σκοπό την απλοποίηση των μεθόδων ενημέρωσης και τη βελτίωση των χρόνων δεικτοδότησης και αναζήτησης κειμένων. Προσεγγίσεις οι οποίες οργανώνουν το ευρετήριο σε μπλοκς και αποθηκεύουν τις ανεστραμμένες λίστες σε μπλοκς προεπιλεγμένων μεγεθών δεν έχουν μελετηθεί αρκετά, εφόσον οι έρευνες που έχουν γίνει περιορίζουν το μέγεθος του μπλοκ σε μερικές δεκάδες kilobytes [2, 26].

Βάσει λοιπόν μιας τέτοιας οργάνωσης του ευρετηρίου σε μπλοκς, η αποθήκευση των μεγάλων λιστών μεταφράζεται στην αποθήκευση των εμφανίσεων τους σε έναν αριθμό από ανεξάρτητα μπλοκς, όπου ένα νέο μπλοκ δεσμεύεται κάθε φορά που ο χώρος στο υπάρχον μπλοκ δεν είναι αρκετός για την αποθήκευση των νέων εμφανίσεων. Παρόμοια, η ενημέρωση των μικρών λιστών απλοποιείται, αποθηκεύοντας πολλές μικρές λίστες σε ένα μπλοκ και ενημερώνοντας τις βάσει μιας μεθόδου συγχώνευσης. Με τον τρόπο αυτό, η ενημέρωση του ευρετηρίου περιορίζεται στην ενημέρωση των μπλοκς του ευρετηρίου που περιέχουν όρους για τους οποίους συλλέχθηκαν νέες εμφανίσεις, ενώ η διαχείριση του ελεύθερου χώρου στο ανεστραμμένο ευρετήριο –η οποία μέχρι πρότινος αποτελούσε ένα σημαντικό πρόβλημα λόγω των πολύπλοκων δομών που απαιτούσε– γίνεται αρκετά πιο απλή.

Το σύστημα δεικτοδότησης που προτείνουμε βασίζεται στο διαχωρισμό των λιστών ανάλογα με το μέγεθος τους σε μικρές και μεγάλες και στη χρήση διαφορετικών αλγορίθμων ενημέρωσης για κάθε κατηγορία. Τόσο οι μικρές όσο και οι μεγάλες λίστες αποθηκεύονται σε μπλοκς προεπιλεγμένου μεγέθους, με τη μόνη διαφορά πως αποθηκεύουμε πολλές μικρές λίστες σε ένα μπλοκ ενώ κάθε μεγάλη λίστα αποθηκεύεται σε έναν αριθμό από αποκλειστικά δικά της μπλοκς. Ανά πάσα στιγμή διατηρούμε την πληροφορία του ποιες λίστες βρίσκονται αποθηκευμένες σε κάθε μπλοκ του συστήματος και πόσος ελεύθερος χώρος υπάρχει σε αυτό. Χρησιμοποιώντας τις πληροφορίες αυτές και τον προτεινόμενο αλγόριθμο συγχώνευσης, κάθε φορά που εξαντλείται η μνήμη επιλέγουμε να συγχωνεύσουμε με το δίσκο ένα υποσύνολο των λιστών της μνήμης, και συγκεκριμένα επιλέγουμε εκείνες τις λίστες οι οποίες προσφέρουν το μέγιστο κέρδος με κριτήριο την ελαχιστοποίηση των μετακινήσεων του

δίσκου και την απελευθέρωση του μεγαλύτερου ποσοστού μνήμης.

4.2 Αρχιτεκτονική συστήματος

Θεωρούμε πως το ευρετήριο αποτελείται από δύο δομές, το λεξικό και το ανεστραμμένο αρχείο. Για κάθε όρο που εμφανίζεται στη συλλογή κειμένων, το ανεστραμμένο αρχείο περιέχει ένα σύνολο δεικτών σε όλα τα κείμενα στα οποία εμφανίζεται ο όρος, όπως επίσης και τις θέσεις στις οποίες εμφανίζεται ο όρος μέσα σε κάθε κείμενο. Το σύνολο αυτών των εμφανίσεων του όρου αποτελεί την ανεστραμμένη λίστα του (inverted list) ή λίστα εμφανίσεων του (postings list). Το λεξικό παρέχει την δυνατότητα αντιστοίχισης ενός όρου στην ανεστραμμένη λίστα του στο δίσκο, ενώ για κάθε όρο περιέχει και διάφορες άλλες βοηθητικές πληροφορίες οι οποίες χρησιμοποιούνται κυρίως κατά την αποτίμηση ερωτημάτων, όπως το συνολικό πλήθος των εμφανίσεων του, το πλήθος των κειμένων στα οποία εμφανίζεται κτλ. Κάθε *εμφάνιση* (posting) ενός όρου t είναι της μορφής:

$$\langle d;\ f_{d,t};\ pos_1, pos_2, ..., pos_{f_{d,t}} \rangle$$

και αντιστοιχεί στο σύνολο των εμφανίσεων του όρου t στο κείμενο d, ενώ η ανεστραμμένη λίστα του όρου είναι της μορφής:

$$\langle d_1;\ f_{d_1,t};\ pos_1, ..., pos_{f_{d_1,t}} \rangle, \langle d_2;\ f_{d_2,t};\ pos_1, ..., pos_{f_{d_2,t}} \rangle, ..., \langle d_k;\ f_{d_k,t};\ pos_1, ..., pos_{f_{d_k,t}} \rangle$$

Οι εμφανίσεις ενός όρου αποθηκεύονται στην ανεστραμμένη λίστα του ταξινομημένες ως προς τα αναγνωριστικό των κειμένων d_i, ενώ οι θέσεις εμφάνισης pos_i του όρου μέσα στο κείμενο αποθηκεύονται σε αύξουσα διάταξη, έτσι ώστε τόσο τα αναγνωριστικά όσο και οι θέσεις εμφάνισης να μπορούν να αποθηκευτούν αποδοτικά στο δίσκο (βλέπε Ενότητα 2.3.4). Στον Πίνακα 4.1 παρουσιάζονται οι κυριότερες παράμετροι του συστήματος, οι οποίες εξηγούνται στη συνέχεια. Θεωρούμε πως το λεξικό αποθηκεύεται στο δίσκο σαν ένα B-tree. Χρησιμοποιούμε την παράμετρο N_b για να ορίσουμε το μέγεθος ενός κόμβου του B-tree.

Κατά την δεικτοδότηση μιας συλλογής κειμένων, νέα κείμενα ανατίθενται συνεχώς στη διαδικασία λεκτικής ανάλυσης. Το αποτέλεσμα της λεκτικής ανάλυσης ενός κειμένου είναι η ανάλυση του σε ένα σύνολο από εμφανίσεις οι οποίες συλλέγονται σε ένα τμήμα προσωρινής μνήμης (buffer), μεγέθους P_t. Για την αποδοτική συλλογή των εμφανίσεων χρησιμοποιείται ένας κατάλληλος *πίνακας κατακερματισμού*[1] [30]. Όταν η προσωρινή

[1] διάφορες άλλες δομές για την συλλογή των εμφανίσεων (όπως τα burst tries [12]) τα οποία επιτρέπουν προσπέλαση βάσει λεξικογραφικής διάταξης, περιγράφονται και συγκρίνονται στο [10]

Πίνακας 4.1: Παράμετροι του συστήματος.

Σύμβολο	Περιγραφή
B_s	Μέγεθος μπλοκ συστήματος αρχείων
B_p	Μέγεθος μπλοκ εμφανίσεων (πολλαπλάσιο του B_s)
N_b	Μέγεθος κόμβου του B-tree (πολλαπλάσιο του B_s)
P_t	Διαθέσιμη Μνήμη για την συγκέντρωση των εμφανίσεων
P_f	Μνήμη Συγχώνευσης
K_t	Αναλογία Κόστους Συγχώνευσης
T_t	Κατώφλι Μεγάλων Λιστών
t_{short}	Χρόνος συγχώνευσης μικρού εύρους ($t_{short} = K_t \times t_{long}$)
t_{long}	Χρόνος συγχώνευσης μεγάλου εύρους

μνήμη για την συλλογή των εμφανίσεων γεμίσει, αποθηκεύουμε στο δίσκο τις εμφανίσεις κάποιων όρων. Αυτό γίνεται συγχωνεύοντας για τους επιλεγμένους όρους τις εμφανίσεις τους από την μνήμη με τις εμφανίσεις τους που περιέχονται στις ανεστραμμένες λίστες τους στο δίσκο.

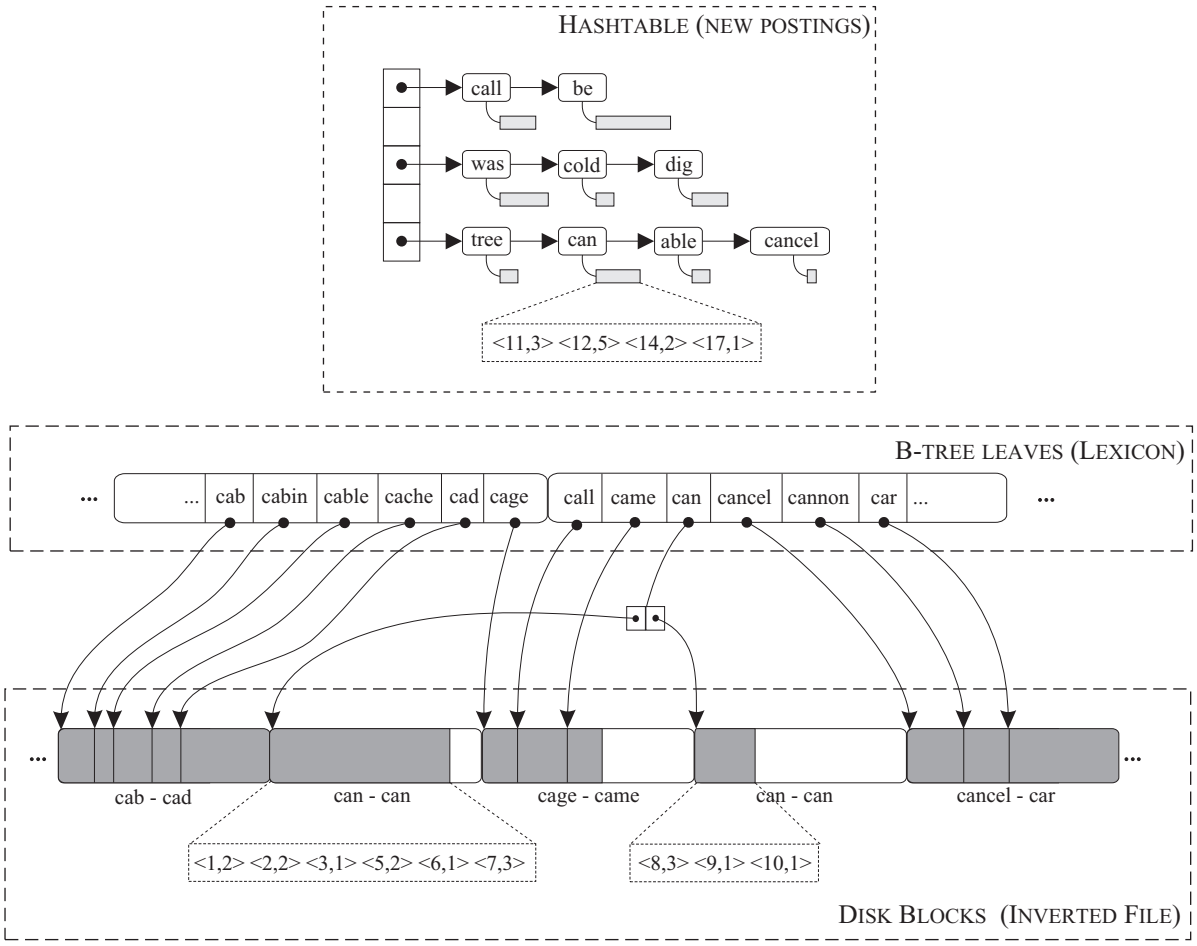
Βάσει του συνολικού μεγέθους που καταλαμβάνουν οι εμφανίσεις μιας λίστας (τόσο αυτές που είναι αποθηκευμένες στο δίσκο όσο και αυτές που έχουν συλλεχθεί για αυτήν στη μνήμη), η λίστα κατηγοριοποιείται ως *μικρή ή μεγάλη*, ενώ ο όρος στον οποίο αναφέρεται η λίστα ως *σπάνιος ή συχνός*. Ο διαχωρισμός αυτός γίνεται βάσει της παραμέτρου T_t, την οποία γι αυτό το λόγο αποκαλούμε *κατώφλι μεγάλων λιστών*.

Όλες οι εμφανίσεις αποθηκεύονται στο δίσκο σε μπλοκς προεπιλεγμένου μεγέθους τα οποία ονομάζουμε *μπλοκ εμφανίσεων*, για να τα διακρίνουμε από τα μπλοκ του συστήματος αρχείων. Το μέγεθος B_p ενός μπλοκ εμφανίσεων είναι μία από τις παραμέτρους του συστήματος και ορίζεται σαν ακέραιο πολλαπλάσιο του μεγέθους μπλοκ B_s που ορίζεται από το σύστημα αρχείων που χρησιμοποιούμε. Στο εξής θα χρησιμοποιούμε απλά τον όρο "μπλοκ" για να αναφερόμαστε στα *μπλοκ εμφανίσεων* και όχι στα φυσικά μπλοκ του συστήματος αρχείων, τον όρο "εμφανίσεις" για να αναφερόμαστε στις εμφανίσεις ενός όρου που έχουν συλλεχθεί *στη μνήμη*, και τον όρο "λίστα" για να αναφερόμαστε στις εμφανίσεις ενός όρου που είναι αποθηκευμένες *στο δίσκο* στην ανεστραμμένη λίστα του.

Αποθηκεύουμε πολλές μικρές λίστες σε ένα μπλοκ, ενώ οι μεγάλες λίστες αποθηκεύονται σε ένα ή περισσότερα (ανάλογα με το μέγεθος της λίστας) αποκλειστικά δικά τους μπλοκς. Καθώς νέα έγγραφα αναλύονται λεκτικά και νέες εμφανίσεις προστίθενται στις λίστες των όρων στο δίσκο, μία λίστα μπορεί να αλλάξει κατηγορία από μικρή σε μεγάλη.

Καλούμε *εύρος* το σύνολο των όρων των οποίων οι λίστες τους είναι αποθηκευμένες

στο ίδιο μπλοκ. Χάριν απλότητας, καλούμε επίσης *μικρά ή μεγάλα* τα εύρη που περιέχουν μικρές ή μεγάλες λίστες αντίστοιχα. Υπάρχει έτσι μία N προς 1 αντιστοιχία μεταξύ όρων και μπλοκς για τα μικρά εύρη, και μία 1 προς N αντιστοιχία όρων και μπλοκς για τα μεγάλα εύρη: ένα μικρό εύρος περιέχει *πολλούς* όρους των οποίων οι λίστες αποθηκεύονται σε *ένα* μπλοκ, ενώ ένα μεγάλο εύρος περιέχει *έναν* όρο του οποίου η λίστα αποθηκεύεται σε *πολλά* μπλοκς. Ένα στιγμιότυπο της διαδικασίας δεικτοδότησης μιας συλλογής περιγράφεται στο Σχήμα 4.1 (για λόγους απλότητας δεν απεικονίζουμε τις θέσεις εμφάνισης pos_i μέσα στις εμφανίσεις).



Σχήμα 4.1: Στιγμιότυπο της διαδικασίας δεικτοδότησης.

4.3 Αλγόριθμος Πρωτέας

Αν θεωρήσουμε την λεξικογραφική διάταξη $t_1 < t_2 < ... < t_N$, όπου t_i κάθε όρος που έχει εμφανιστεί τουλάχιστον μία φορά στα κείμενα που έχουμε δεικτοδοτήσει μέχρι στιγμής,

τότε κάθε εύρος R_i θα περιέχει πάντα όρους που ανήκουν σε ένα υποδιάστημα $[t_m, t_n]$ του $[t_1, t_N]$. Ο συμβολισμός $R_i = [t_m, t_n]$ σημαίνει πως το εύρος R_i περιέχει όλους τους όρους που είναι λεξικογραφικά μεγαλύτεροι ή ίσοι του όρου t_m και μικρότεροι ή ίσοι του όρου t_n. Ισχύει επίσης πως $R_1 \cup R_2 \cup ... \cup R_M = [t_1, t_N]$ όπου M το πλήθος των ευρών, και $R_i \cap R_j = \emptyset$ για $i \neq j$. Συμβολίζουμε με R.low και R.high τον μικρότερο και μεγαλύτερο αντίστοιχα όρο που ανήκει σε ένα εύρος R (στο παράδειγμα, R_i.low $= t_m$, R_i.high $= t_n$). Για τα μεγάλα εύρη, τα εύρη δηλαδή που περιέχουν μία μεγάλη λίστα, το εύρος προφανώς είναι τετριμμένο, δηλαδή ισχύει R.low $= R$.high. Με R.mem_postings συμβολίζουμε το συνολικό μέγεθος που καταλαμβάνουν οι εμφανίσεις όλων των όρων ενός εύρους R στη μνήμη.

Υποθέτουμε πως το σύστημα δεσμεύει προσωρινή μνήμη μεγέθους P_t για την συγκέντρωση των εμφανίσεων που εξάγονται από τα αρχεία, όπως αναφέραμε και νωρίτερα. Κάθε φορά που η προσωρινή αυτή μνήμη γεμίζει, αποθηκεύουμε στο δίσκο ένα τμήμα της προσωρινής μνήμης μεγέθους P_f. Ονομάζουμε το μέγεθος P_f μέγεθος *μνήμης συγχώνευσης*, καθώς οι εμφανίσεις που θα αποθηκευτούν στο δίσκο ουσιαστικά θα συγχωνευτούν με τις αντίστοιχες λίστες τους στο δίσκο.

Το P_f είναι συνήθως μία τάξη μεγέθους μικρότερο από το P_t. Έτσι, περιοδικά αποθηκεύουμε στο δίσκο μόνο ένα μικρό τμήμα των εμφανίσεων που έχουν συγκεντρωθεί στη μνήμη, και πιο συγκεκριμένα, αποθηκεύουμε στο δίσκο τις εμφανίσεις που ανήκουν σε ένα σύνολο από εύρη, τα οποία επιλέγουμε εμείς: μόνο εκείνα τα εύρη των οποίων οι όροι έχουν συγκεντρώσει συνολικά "αρκετές" εμφανίσεις στη μνήμη θα αποθηκευθούν στο δίσκο. Στο Σχήμα 4.2 παρουσιάζεται ο γενικός αλγόριθμος δημιουργίας του ευρετηρίου.

```
1.  Postings are accumulated in main memory as documents are added to
    the collection
2.  Once main memory is exhausted (in-memory postings size > Pt)
3.      Based on Proteus algorithm, select a number of ranges and merge
        their in-memory postings with their lists on disk
4.      Discard the flushed postings from memory
```

Σχήμα 4.2: Γενικός αλγόριθμος δημιουργίας ευρετηρίου.

Καθώς τα εύρη είναι κατηγοριοποιημένα ως μικρά ή μεγάλα, ανάλογα με το αν περιέχουν πολλές μικρές λίστες ή μία μεγάλη, χρησιμοποιούμε διαφορετικό αλγόριθμο για την συγχώνευση των εμφανίσεων ενός εύρους με τις αντίστοιχες λίστες του στο δίσκο, ανάλογα με την κατηγορία στην οποία ανήκει το εύρος. Για τα μικρά εύρη χρησιμοποιούμε έναν αλγόριθμο συγχώνευσης: διαβάζουμε το μπλοκ του εύρους από τον δίσκο, προσθέτουμε

τις νέες εμφανίσεις στις αντίστοιχες λίστες και τέλος γράφουμε το μπλοκ πίσω στο δίσκο. Για τα μεγάλα εύρη χρησιμοποιούμε τον αλγόριθμο της επιτόπου ενημέρωσης: προσθέτουμε απλά τα τις νέες εμφανίσεις στο τέλος της λίστας στο δίσκο. Ο αλγόριθμος συγχώνευσης ο οποίος εκτελείται όταν η προσωρινή μνήμη γεμίσει σκιαγραφείται στο Σχήμα 4.3. Στη συνέχεια περιγράφονται τα διάφορα βήματα του.

```
    /* flush P_f bytes from postings buffer to disk */
 1. proc Proteus_algorithm
 2.    Sort long ranges by occupied buffer space
 3.    Sort short ranges by occupied buffer space
 4.    While (flushed buffer space < P_f) do
 5.       R_long ← max long range currently in buffer
 6.       R_short ← max short range currently in buffer
 7.       If (R_short.mem_postings/R_long.mem_postings < K_t) then
 8.          Append in-memory postings of R_long to last postings block
             on disk
 9.          Allocate additional postings blocks if necessary
10.          Remove R_long from postings buffer
11.       Else
12.          Merge in-memory postings of R_short with corresponding
             postings block
13.          If (postings block capacity exceeded) then
14              split contents into multiple postings blocks
15.          Remove R_short from postings buffer
16.       Update lexicon
```

Σχήμα 4.3: Αλγόριθμος συγχώνευσης Πρωτέας.

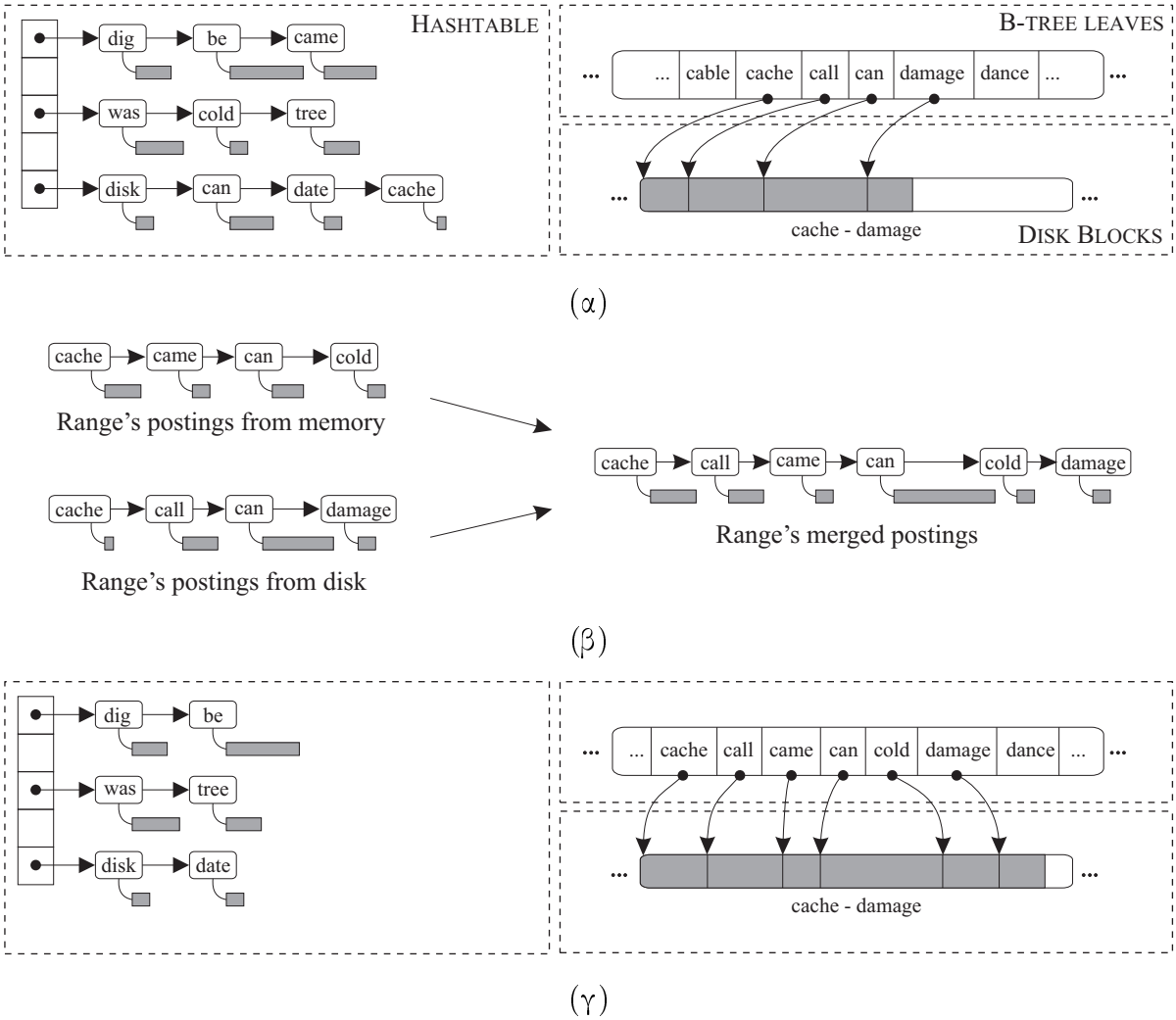
4.3.1 Αποθήκευση μικρού εύρους

Όταν επιλέξουμε ένα μικρό εύρος για να αποθηκεύσουμε τις εμφανίσεις του στο δίσκο, ουσιαστικά συγχωνεύοντάς τις με τις αντίστοιχες λίστες του δίσκου, τότε αρχικά ανακτούμε το μπλοκ του από τον δίσκο. Φυσικά ανακτούμε μόνο το τμήμα που μπλοκ που περιέχει τις λίστες και όχι ολόκληρο το μπλοκ. Έπειτα συγκεντρώνουμε από την μνήμη όλους τους όρους του εύρους μαζί με τις νέες εμφανίσεις τους, και ταξινομούμε τους όρους λεξικογραφικά[2].

Οι λίστες μέσα σε ένα μπλοκ στο δίσκο αποθηκεύονται *πάντα* λεξικογραφικά ταξινομημένες. Έχοντας πλέον και τις εμφανίσεις από τη μνήμη λεξικογραφικά ταξινομημένες, στη συνέχεια διατρέχουμε τις λίστες από το μπλοκ και τις εμφανίσεις των όρων από την

[2] η χρήση του πίνακα κατακερματισμού [30] για την συγκέντρωση των εμφανίσεων στη μνήμη παρέχει το πλεονέκτημα της πολύ γρήγορης προσθήκης των νέων εμφανίσεων στον πίνακα, αλλά έχει το μειονέκτημα πως δεν μπορούμε να προσπελάσουμε τους όρους βάσει λεξικογραφικής διάταξης.

μνήμη, συγχωνεύοντας[3] όπου χρειάζεται τις νέες εμφανίσεις με την αντίστοιχη λίστα, ή δημιουργώντας νέες λίστες στην περίπτωση που κάποιος όρος εμφανίζεται για πρώτη φορά. Στο τέλος θα έχει παραχθεί ένα σύνολο από λίστες, λεξικογραφικά ταξινομημένες, από τις οποίες κάποιες θα είναι λίστες από τον δίσκο με την προσθήκη κάποιων νέων εμφανίσεων, κάποιες θα είναι νέες λίστες, και κάποιες θα είναι λίστες του δίσκου οι οποίες έχουν παραμείνει ανεπηρέαστες. Το σύνολο αυτό των λιστών αποθηκεύεται τελικά πίσω στο μπλοκ στο δίσκο. Τέλος, σβήνουμε τις εμφανίσεις του εύρους από τον πίνακα κατακερματισμού και ενημερώνουμε το λεξικό. Τα παραπάνω βήματα περιγράφονται στο Σχήμα 4.4.



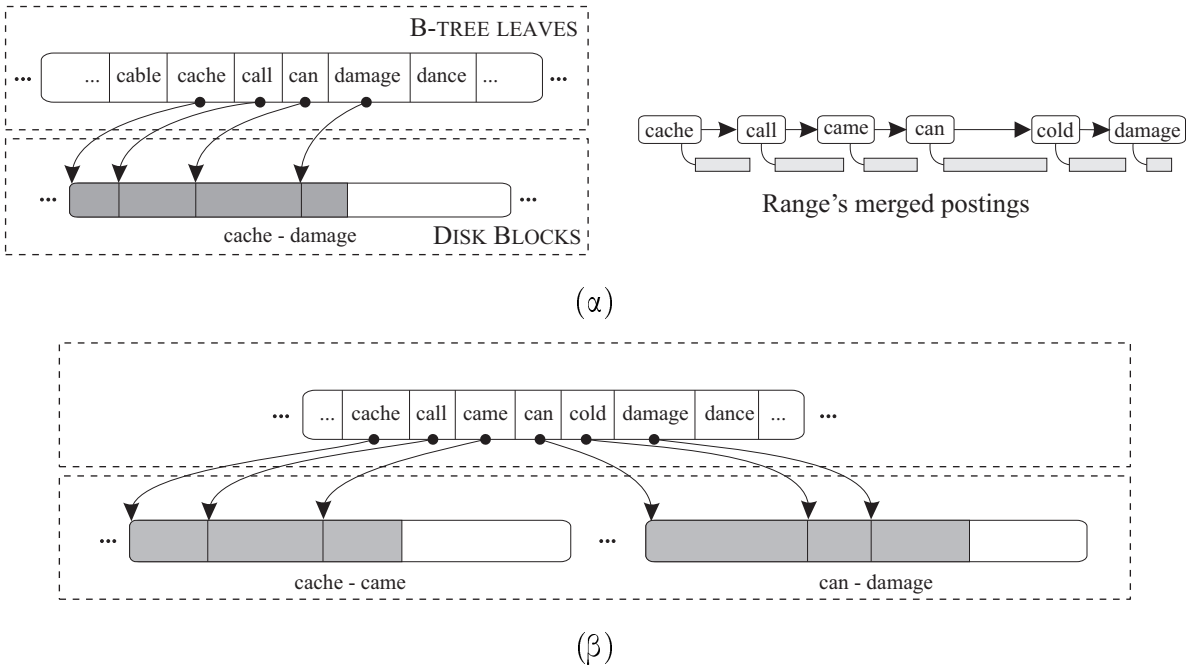
Σχήμα 4.4: Διαδικασία αποθήκευσης των εμφανίσεων ενός μικρού εύρους: (α) κατάσταση πριν την συγχώνευση, (β) συγχώνευση των εμφανίσεων, (γ) κατάσταση μετά την συγχώνευση.

[3] η συγχώνευση νέων εμφανίσεων σε μία λίστα γίνεται απλά κωδικοποιώντας το πρώτο αναγνωριστικό κειμένου από τις εμφανίσεις της μνήμης και έπειτα προσθέτοντας τις εμφανίσεις στο τέλος της λίστας – βλ. Ενότητα 2.3.5

Διάσπαση μικρού εύρους λόγω υπερχείλισης του μπλοκ

Αν μετά την συγχώνευση το συνολικό μέγεθος των λιστών του εύρους είναι μεγαλύτερο από το μέγεθος του μπλοκ, και άρα δεν μπορούν να αποθηκευτούν πίσω στο μπλοκ στο δίσκο (έχουμε *υπερχείλιση* του μπλοκ), τότε αποθηκεύουμε τις λίστες σε δύο μπλοκς όπου κάθε μπλοκ περιέχει περίπου το ίδιο μέγεθος λιστών. Η ανάθεση των λιστών στα δύο μπλοκς γίνεται με βάση την λεξικογραφική τους διάταξη: αρχικά επιλέγεται ένας όρος t από το εύρος, και βάσει του t αποθηκεύουμε στο πρώτο μπλοκ όλες τις λίστες των όρων που είναι μικρότεροι ή ίσοι του t και στο δεύτερο μπλοκ όλες τις υπόλοιπες λίστες. Ο όρος t επιλέγεται με τέτοιον τρόπο ώστε οι λίστες στα δύο μπλοκς να καταλαμβάνουν περίπου το ίδιο μέγεθος.

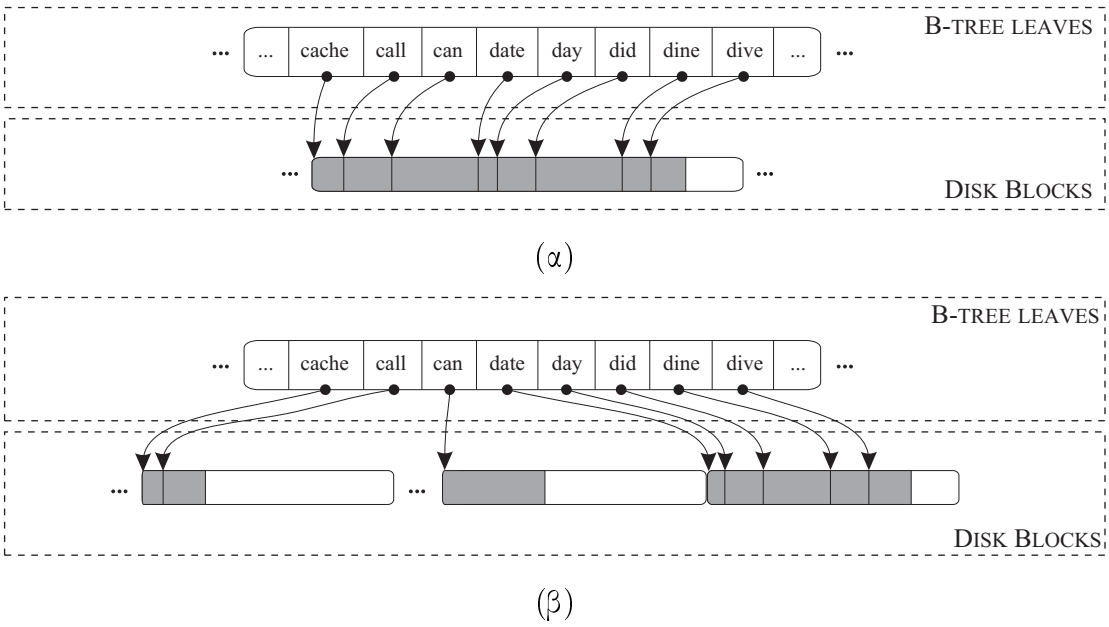
Η διάσπαση αυτή των λιστών σε δύο μπλοκς έχει ως συνέπεια και την διάσπαση του αρχικού εύρους σε δύο μικρότερα εύρη, σε αναλογία με τις λίστες που αποθηκεύτηκαν σε κάθε μπλοκ: το πρώτο εύρος θα περιέχει όλους τους όρους που αποθηκεύτηκαν στο πρώτο μπλοκ, και το δεύτερο εύρος τους όρους που αποθηκεύτηκαν στο δεύτερο μπλοκ. Η παραπάνω διαδικασία περιγράφεται στο Σχήμα 4.5.



Σχήμα 4.5: Διάσπαση ενός μικρού εύρους λόγω υπερχείλισης του μπλοκ: (α) κατάσταση πριν τη συγχώνευση, (β) κατάσταση μετά τη συγχώνευση

Διάσπαση μικρού εύρους λόγω εμφάνισης μεγάλης λίστας

Κατά την συγχώνευση ενός μικρού εύρους R υπάρχει η πιθανότητα το μέγεθος μίας λίστας I_t ενός όρου t μετά την προσθήκη κάποιων νέων εμφανίσεων να ξεπεράσει το κατώφλι T_t. Από την στιγμή εκείνη η συγκεκριμένη λίστα I_t θεωρείται 'μεγάλη' λίστα και ο όρος t 'συχνός' όρος. Επειδή κάθε μεγάλη λίστα αποθηκεύεται σε αποκλειστικά δικά της μπλοκς, για τον σκοπό αυτό δεσμεύουμε ένα νέο μπλοκ για τον όρο t και αποθηκεύουμε στο συγκεκριμένο μπλοκ την I_t. Το γεγονός αυτό έχει ως αποτέλεσμα πως για να είμαστε συνεπείς[4] το αρχικό εύρος R πρέπει να διασπαστεί σε τρία εύρη, έστω R_1, R_2, R_3: το εύρος R_1 θα περιέχει όλους τους όρους R οι οποίοι είναι μικρότεροι από τον t, το R_2 θα περιέχει μόνο τον t, και το R_3 θα περιέχει αντίστοιχα όλους τους όρους οι οποίοι είναι μεγαλύτεροι του t. Αντίστοιχα, οι λίστες των R_1 και R_3 πρέπει να αποθηκευτούν σε διαφορετικά μπλοκς στο δίσκο (Σχήμα 4.6).

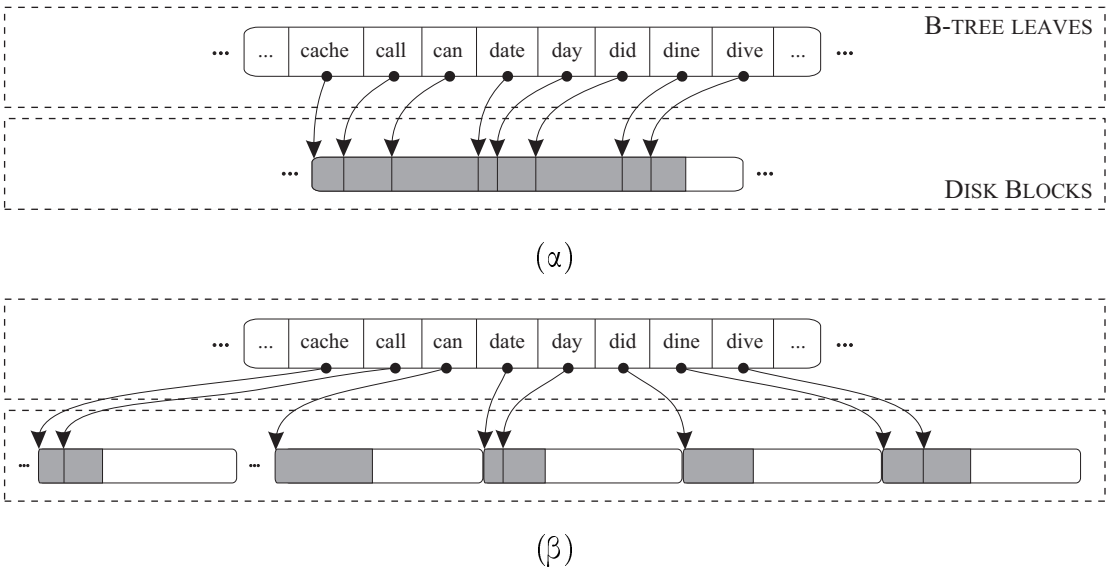


Σχήμα 4.6: Διάσπαση ενός μικρού εύρους λόγω εμφάνισης μεγάλης λίστας ("can"): (α) κατάσταση πριν τη συγχώνευση, (β) κατάσταση μετά την συγχώνευση.

Μετά από την παραπάνω "τριπλή διάσπαση" του αρχικού εύρους R κατά την συγχώνευση του, συνεχίζουμε να συγχωνεύουμε τις εμφανίσεις του εύρους R_3, καθώς είχαμε "διακόψει" προσωρινά την διαδικασία συγχώνευσης των εμφανίσεων στον όρο t. Κατά την συγχώνευση τώρα των όρων του R_3 υπάρχει επίσης η πιθανότητα η λίστα I_x ενός όρου x να αλλάξει κατηγορία από μικρή σε μεγάλη, μετά από τη συγχώνευση της λίστας από τον δίσκο με

[4] η ανάγκη αυτή προκύπτει από το γεγονός πως κάθε εύρος (όπως και κάθε μπλοκ) μπορεί να περιέχει είτε μόνο μικρές λίστες είτε μόνο μεγάλες λίστες

τις εμφανίσεις των όρων από την μνήμη. Στην περίπτωση αυτή το εύρος R_3 θα χωριστεί αναδρομικά σε τρία εύρη βάσει του x, με παρόμοιο τρόπο (Σχήμα 4.7).

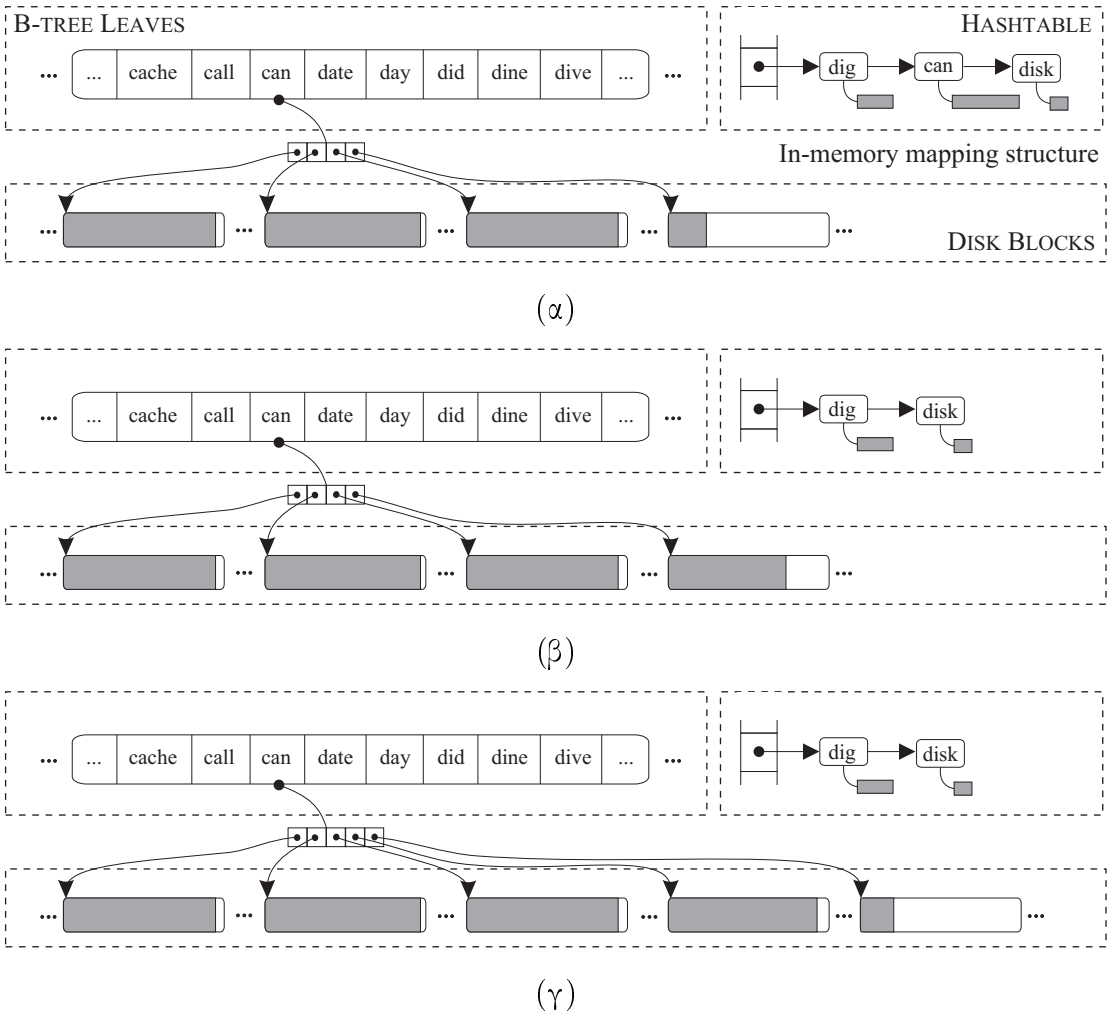


Σχήμα 4.7: Δύο διαδοχικές διασπάσεις ενός μικρού εύρους λόγω εμφάνισης δύο μεγάλων λιστών ("can", "did"): (α) κατάσταση πριν τη συγχώνευση, (β) κατάσταση μετά την συγχώνευση.

4.3.2 Αποθήκευση μεγάλου εύρους

Ένα μεγάλο εύρος συγκεντρώνει στη μνήμη εμφανίσεις που ανήκουν σε έναν μόνο όρο και ενημερώνεται βάσει της επιτόπου ενημέρωσης. Έτσι, αν ένα μεγάλο εύρος επιλεγεί για να αποθηκευτεί στο δίσκο, αρκεί να συλλέξουμε τις εμφανίσεις του από την μνήμη και να τις προσθέσουμε στο τέλος της λίστας του στο δίσκο. Φυσικά αυτό προϋποθέτει, όπως έχουμε αναφέρει, πως όλες οι πληροφορίες για την κωδικοποίηση των νέων εμφανίσεων βρίσκονται στη μνήμη, και έτσι δεν χρειάζεται να διαβάσουμε την λίστα από το δίσκο για να προσθέσουμε τις νέες εμφανίσεις: αρκεί να κωδικοποιήσουμε τις νέες εμφανίσεις και να τις γράψουμε στον ελεύθερο χώρο στο τελευταίο μπλοκ της λίστας στο δίσκο. Σε περίπτωση υπερχείλισης του τελευταίου μπλοκ δεσμεύουμε ένα νέο μπλοκ, αποθηκεύουμε όσες εμφανίσεις χωράνε στο τελευταίο μπλοκ και τις υπόλοιπες στο νέο μπλοκ (Σχήμα 4.8).

Επειδή, σε αντίθεση με την συγχώνευση ενός μικρού εύρους, στην περίπτωση της συγχώνευσης ενός μεγάλου εύρους δεν απαιτείται η ανάκτηση του μπλοκ από τον δίσκο ούτε χρειάζονται διάφορες ολισθήσεις στη μνήμη για την συγχώνευση, για το λόγο αυτό η αποθήκευση ενός μεγάλου εύρους στο δίσκο είναι πολύ αποδοτικότερη από την αποθήκευση ενός μικρού εύρους.



Σχήμα 4.8: Ενημέρωση της λίστας ενός συχνού όρου: (α) κατάσταση πριν τη συγχώνευση (β) οι νέες εμφανίσεις χωράνε στο τελευταίο μπλοκ στο δίσκο (γ) οι νέες εμφανίσεις δεν χωράνε στο τελευταίο μπλοκ.

4.3.3 Επιλογή των ευρών

Όπως έχουμε αναφέρει, η αποθήκευση ενός μικρού εύρους περιλαμβάνει μία ανάγνωση από τον δίσκο, ολισθήσεις στη μνήμη για την συγχώνευση των λιστών, και τέλος μία η περισσότερες[5] εγγραφές στο δίσκο. Από την άλλη, η αποθήκευση ενός μεγάλου εύρους συνεπάγεται μόνο εγγραφές στο δίσκο. Επίσης, η αποθήκευση ενός μικρού εύρους περιλαμβάνει την ανάκτηση και αποθήκευση *όλων* των λιστών του εύρους, ακόμα και αν κάποιες από αυτές δεν τροποποιηθούν κατά τη διάρκεια της συγχώνευσης, ενώ η αποθήκευση ενός μεγάλου εύρους περιλαμβάνει την εγγραφή μόνο των νέων εμφανίσεων που έχουν συγκεντρωθεί στη μνήμη.

[5] ανάλογα με το αν έχουμε διασπάσεις του εύρους λόγω εμφάνισης μεγάλων όρων ή λόγω υπερχείλισης του μπλοκ

Από τον τρόπο με τον οποίον αποθηκεύεται στο δίσκο ένα μικρό και ένα μεγάλο εύρος γίνεται σαφές πως η αποθήκευση ενός μικρού εύρους απαιτεί περισσότερο χρόνο από την αποθήκευση ενός μεγάλου εύρους. Ιδανικά, θα θέλαμε να επιλέξουμε εκείνη την ακολουθία αποθήκευσης ευρών η οποία θα μεγιστοποιούσε την *ρυθμαπόδοση αποθήκευσης* (flushing throughput) των εμφανίσεων στο δίσκο. Πιο συγκεκριμένα, κάθε φορά που γεμίζει η προσωρινή μνήμη και καλούμαστε να αποθηκεύσουμε ένα σύνολο ευρών στο δίσκο, θα θέλαμε να αποθηκεύσουμε τα εύρη εκείνα τα οποία θα απελευθερώσουν το μέγιστο δυνατό μέγεθος μνήμης στον ελάχιστο δυνατό χρόνο. Στον αλγόριθμο Πρωτέας που περιγράφεται στο Σχήμα 4.3 χρησιμοποιούμε μία άπληστη προσέγγιση για την επιλογή του εύρους που θα αποθηκεύσουμε σε κάθε βήμα.

Μοντέλο κόστους

Ας υποθέσουμε πως ο χρόνος για να αποθηκευτεί ένα μικρό ή μεγάλο εύρος είναι t_{short} και t_{long} αντίστοιχα, *ανεξάρτητα από το πλήθος των εμφανίσεων που περιέχουν*[6]. Υποθέτουμε δηλαδή πως ο χρόνος για να αποθηκευτεί ένα εύρος είναι σταθερός, και επιπλέον πως $t_{short} = K_t \times t_{long}$, δηλαδή ο χρόνος για να αποθηκεύσουμε ένα μικρό εύρος είναι K_t φορές μεγαλύτερος από τον αντίστοιχο χρόνο αποθήκευσης ενός μεγάλου εύρους. Η σταθερά K_t είναι μία από τις παραμέτρους του αλγορίθμου και εξαρτάται από τα διάφορα τεχνικά χαρακτηριστικά και την αρχιτεκτονική του συστήματος. Σύμφωνα λοιπόν με τα παραπάνω, αν αποθηκεύσουμε 1 byte κατά την αποθήκευση ενός μικρού ή μεγάλου εύρους, λαμβάνουμε αντίστοιχα ρυθμαπόδοση αποθήκευσης $1/t_{short}$ και $1/t_{long} = K_t/t_{long}$. Με άλλα λόγια, κάθε byte που αποθηκεύουμε κατά την αποθήκευση ενός μεγάλου εύρους πετυχαίνει ρυθμαπόδοση αποθήκευσης (bytes/s) K_t φορές μεγαλύτερη από την περίπτωση που αυτό το byte θα αποθηκεύονταν κατά την αποθήκευση ενός μικρού εύρους. Έτσι, ένα μικρό εύρος πρέπει να περιέχει ακριβώς K_t φορές περισσότερες εμφανίσεις ώστε να επιτύχει ίδια ρυθμαπόδοση αποθήκευσης με ένα μεγάλο εύρος. Αν το μικρό εύρος περιέχει περισσότερες ή λιγότερες εμφανίσεις, τότε η ρυθμαπόδοση αποθήκευσης που επιτυγχάνει θα είναι μεγαλύτερη ή μικρότερη αντίστοιχα.

Σύμφωνα με τα παραπάνω, σε κάθε βήμα του αλγορίθμου συγχώνευσης βρίσκουμε εκείνο το μικρό και μεγάλο εύρος τα οποία έχουν συγκεντρώσει τις περισσότερες εμφανίσεις στη μνήμη, έστω $R_{short,max}$ και $R_{long,max}$, και υπολογίζουμε το λόγο μεταξύ του μεγέθους των εμφανίσεων που έχει συλλέξει το καθένα. Αν ο λόγος αυτός είναι μεγαλύτερος από

[6] η υπόθεση αυτή, όπως φαίνεται και από τα πειραματικά αποτελέσματα (βλ. Ενότητα 6.4), είναι αρκετά βάσιμη

K_t, τότε επιλέγουμε να αποθηκεύσουμε στο δίσκο το $R_{short,max}$, αλλιώς επιλέγουμε το $R_{long,max}$.

Φυσικά, τόσο η άπληστη τεχνική που χρησιμοποιείται για την επιλογή του εύρους που θα αποθηκεύσουμε όσο και η εκτίμηση της ρυθμαπόδοσης αποθήκευσης είναι απλές προσεγγίσεις οι οποίες μπορούν να βελτιωθούν περαιτέρω, χρησιμοποιώντας πιο σύνθετα μοντέλα κόστους.

4.4 Περίληψη

Στο σύστημα υπάρχουν τρεις βασικές δομές: ο πίνακας κατακερματισμού για την συλλογή των νέων εμφανίσεων, τα μπλοκς στο δίσκο τα οποία περιέχουν τις ανεστραμμένες λίστες, και το λεξικό (B-tree) το οποίο αντιστοιχίζει τους όρους στις ανεστραμμένες λίστες τους στον δίσκο (βλ. Σχήμα 4.1). Κάθε μπλοκ του δίσκου είτε περιέχει πολλές μικρές λίστες, είτε περιέχει εμφανίσεις μίας μεγάλης λίστας. Με λίγα λόγια, αποθηκεύουμε πολλές μικρές λίστες σε ένα μπλοκ, ενώ μία μεγάλη λίστα αποθηκεύεται σε πολλά μπλοκς. Κάθε σύνολο όρων των οποίων οι λίστες αποθηκεύονται στο ίδιο μπλοκ αποτελούν ένα *εύρος*. Στην περίπτωση των μεγάλων λιστών, κάθε εύρος περιέχει μόνο έναν όρο.

Καθώς νέα κείμενα ανατίθενται στη διαδικασία λεκτικής ανάλυσης, αυτά αναλύονται σε ένα σύνολο από εμφανίσεις οι οποίες συλλέγονται στην μνήμη, στον πίνακα κατακερματισμού. Για κάθε μπλοκ στο δίσκο διατηρούμε στο αντίστοιχο εύρος του την πληροφορία του ποιες λίστες περιέχει, πόσος ελεύθερος χώρος υπάρχει στο τέλος του, καθώς πόσες εμφανίσεις έχουν συλλεχθεί στη μνήμη για το συγκεκριμένο μπλοκ.

Όταν η μνήμη εξαντληθεί, επιλέγουμε κάποιες εμφανίσεις από τη μνήμη και τις αποθηκεύουμε στα αντίστοιχα μπλοκς τους στο δίσκο. Συγκεκριμένα, επιλέγουμε ένα σύνολο ευρών, για τα οποία συλλέγουμε όλες τις εμφανίσεις τους από τη μνήμη και τις συγχωνεύουμε με τις αντίστοιχες λίστες τους στο δίσκο. Η επιλογή των ευρών τα οποία θα αποθηκεύσουμε στον δίσκο γίνεται βάσει του αλγορίθμου Πρωτέας (Σχήμα 4.3). Γενικά μπορούμε να πούμε πως επιλέγουμε τα εύρη που έχουν συλλέξει τις περισσότερες εμφανίσεις στη μνήμη και που η αποθήκευση τους στο δίσκο μεγιστοποιεί τη ρυθμαπόδοση αποθήκευσης.

Κάθε μικρό εύρος αποθηκεύεται στον δίσκο βάσει της μεθόδου συγχώνευσης (ανακτούμε το μπλοκ από το δίσκο, προσθέτουμε τις νέες εμφανίσεις, γράφουμε το μπλοκ πίσω στο δίσκο), ενώ κάθε μεγάλο εύρος βάσει της επιτόπου ενημέρωσης (προσθέτουμε τις νέες εμφανίσεις στο τελευταίο μπλοκ της λίστας του).

Κεφαλαιο 5

Υλοποιηση

5.1 Σύστημα υλοποίησης

Για την πειραματική μελέτη του συστήματος υλοποιήσαμε τον αλγόριθμο Πρωτέας χρησιμοποιώντας το σύστημα Zettair[1]. Το Zettair είναι μία μηχανή αναζήτησης η οποία υποστηρίζει την δεικτοδότηση και αναζήτηση κειμένων για αρκετά μεγάλες συλλογές. Στις επόμενες ενότητες περιγράφεται επιγραμματικά η αρχιτεκτονική του Zettair και αναφέρονται διάφορες τεχνικές και αλγόριθμοι για την αποδοτική υλοποίηση του αλγορίθμου Πρωτέας.

5.1.1 Το σύστημα Zettair

Το Zettair είναι μία μηχανή αναζήτησης η οποία έχει υλοποιηθεί από την ερευνητική ομάδα Search Engine Group[2] του Πανεπιστημίου RMIT της Μελβούρνης. Γραμμένο σε γλώσσα C, υποστηρίζει ένα μεγάλο εύρος από σύγχρονα λειτουργικά συστήματα, όπως Linux, Free BSD, Max OS X, Solaris, Windows. Είναι ανοιχτού κώδικα και διανείμεται υπό την

[1] http://www.seg.rmit.edu.au/zettair/

[2] http://www.seg.rmit.edu.au/

άδεια BSD. Υποστηρίζει την δημιουργία ενός ευρετηρίου βάσει του οποίου μπορούν να απαντώνται λογικά ερωτήματα (boolean queries), ερωτήματα κατάταξης (rank queries) ή ερωτήματα προτάσεων (phrase queries), ενώ είναι αρκετά γρήγορο και έχει πολύ καλές δυνατότητες κλιμάκωσης. Μπορεί εύκολα να συμπεριληφθεί σε μεγαλύτερα συστήματα μέσω της διεπιφάνειας C που παρέχει, ενώ υποστηρίζει τη δεικτοδότηση ενός αριθμού διαφορετικών τύπων αρχείων, όπως απλά αρχεία κειμένου, κείμενα HTML και κείμενα μορφοποίησης TREC.

5.1.2 Αρχιτεκτονική του Zettair

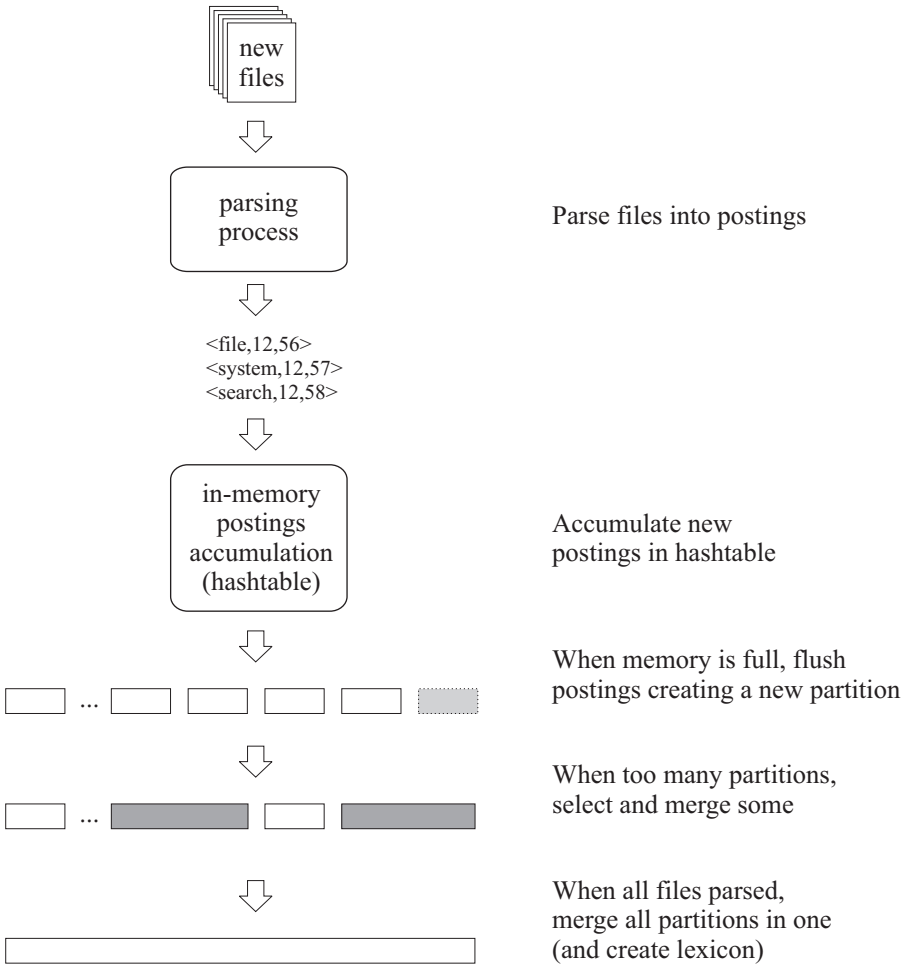
Το Zettair διατηρεί στη μνήμη έναν *πίνακα κατακερματισμού* για να συλλέγει αποδοτικά τους όρους και τις εμφανίσεις τους, καθώς αυτά εξάγονται από την διαδικασία λεκτικής ανάλυσης. Η διαδικασία της λεκτικής ανάλυσης ενός κειμένου έχει σαν αποτέλεσμα την ανάλυση του κειμένου σε έναν αριθμό από εμφανίσεις, οι οποίες αρχικά κωδικοποιούνται και έπειτα προστίθενται στις λίστες εμφανίσεων που διατηρούνται για κάθε όρο στον πίνακα κατακερματισμού. Η διαδικασία αυτή περιγράφεται στην Ενότητα 2.3.5.

Για κάθε όρο για τον οποίο έχουμε συλλέξει στη μνήμη κάποιες εμφανίσεις, υπάρχει στον πίνακα κατακερματισμού μία ειδική δομή, η *δομή εμφανίσεων*, η οποία επιτρέπει την άμεση κωδικοποίηση και προσθήκη των νέων εμφανίσεων του όρου. Μεταξύ των άλλων πληροφοριών που διατηρεί η δομή αυτή για έναν όρο περιέχεται η λίστα με τις εμφανίσεις που έχουμε συλλέξει μέχρι στιγμής στη μνήμη για τον συγκεκριμένο όρο, το τελευταίο αναγνωριστικό κειμένου που έχουμε κωδικοποιήσει στη συγκεκριμένη λίστα (για να είναι εφικτή η άμεση κωδικοποίηση κάθε νέας εμφάνισης του όρου) κ.α..

Όταν το συνολικό μέγεθος του πίνακα κατακερματισμού ξεπεράσει κάποιο συγκεκριμένο όριο, τότε τα περιεχόμενα του αποθηκεύονται στο δίσκο σαν μία ξεχωριστή *διαμέριση* και απελευθερώνεται η μνήμη που δεσμεύουν. Συγκεκριμένα, όταν γεμίσει η διαθέσιμη μνήμη αρχικά συλλέγονται όλες οι δομές εμφανίσεων από την μνήμη και ταξινομούνται βάσει των όρων τους. Έπειτα οι δομές αυτές προσπελαύνονται κατά λεξικογραφική διάταξη και για κάθε όρο αποθηκεύονται στο δίσκο τα εξής: ο ίδιος ο όρος, το πλήθος των κειμένων στα οποία εμφανίστηκε ο όρος από την τελευταία δημιουργία διαμέρισης, το πλήθος των εμφανίσεων που έχει συλλέξει στη μνήμη από την τελευταία δημιουργία διαμέρισης, το τελευταίο αναγνωριστικό κειμένου που κωδικοποιήσαμε, και τέλος, η λίστα με τις εμφανίσεις του όρου που συλλέχθηκαν στη μνήμη. Όλες οι προηγούμενες πληροφορίες αποθηκεύονται στο δίσκο κωδικοποιημένες, έτσι ώστε κάθε διαμέριση να έχει το ελάχιστο δυνατό

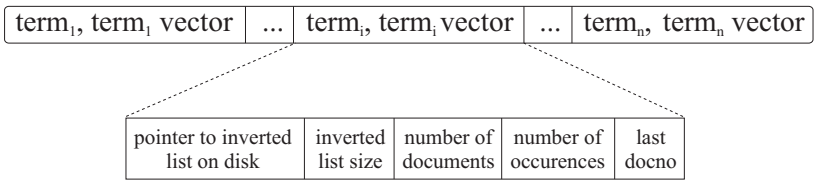
μέγεθος, αλλά κυρίως για να ελαττώνεται το πλήθος των δεδομένων που ανακτώνται και εγγράφονται από και προς τον δίσκο.

Η διαδικασία αυτή (λεκτική ανάλυση νέων κειμένων, συγκέντρωση των εμφανίσεων τους στη μνήμη, δημιουργίας μιας νέας διαμέρισης όταν η μνήμη εξαντληθεί) επαναλαμβάνεται μέχρι να αναλυθούν λεκτικά όλα τα κείμενα της συλλογής. Επειδή οι εμφανίσεις ενός συγκεκριμένου όρου μπορεί να βρίσκονται σε οποιεσδήποτε διαμερίσεις, η ταυτόχρονη ύπαρξη ενός μεγάλου αριθμού διαμερίσεων σημαίνει αυτόματα πως η αναστραμμένη λίστα ενός όρου μπορεί να βρίσκεται κατακερματισμένη σε έναν μεγάλο αριθμό από διαμερίσεις. Κάτι τέτοιο φυσικά επιφέρει μεγάλες επιβαρύνσεις κατά την ανάκτηση της λίστας ενός όρου, με αποτέλεσμα τα ερωτήματα αναζήτησης κατά την διάρκεια της δημιουργίας του ευρετηρίου να απαιτούν πολύ χρόνο. Για το λόγο αυτό το Zettair χρησιμοποιεί τον αλγόριθμο της *γεωμετρικής διαμέρισης* [13] για την μείωση του πλήθους των διαμερίσεων: περιοδικά επιλέγεται ένα πλήθος από διαμερίσεις οι οποίες και συγχωνεύονται σε μια μεγαλύτερη και ενιαία διαμέριση, μειώνοντας έτσι το πλήθος των διαμερίσεων (Σχήμα 5.1).



Σχήμα 5.1: Διάγραμμα ροής για την λειτουργία του Zettair.

Αφού αναλυθούν όλα τα κείμενα της συλλογής, το Zettair συγχωνεύει όλες τις διαμερίσεις σε μια ενιαία διαμέριση, η οποία αποτελεί και το τελικό ανεστραμμένο αρχείο. Η ανεστραμμένη λίστα κάθε όρου αποθηκεύεται συνεχόμενα στο δίσκο μέσα στο ανεστραμμένο αρχείο. Κατά την διάρκεια της τελευταίας αυτής συγχώνευσης δημιουργείται και το λεξικό το οποίο κάνει την αντιστοίχιση μεταξύ των όρων και των ανεστραμμένων λιστών τους στο δίσκο. Το Zettair υλοποιεί το λεξικό με ένα B-tree, οπότε και κατά την διαδικασία της τελευταίας συγχώνευσης των διαμερίσεων δημιουργεί απευθείας στο δίσκο τους κόμβους του B-tree. Για κάθε όρο διατηρείται στο B-tree ένα *διάνυσμα* (term vector) το οποίο περιέχει τις εξής πληροφορίες: έναν δείκτη[3] στην θέση στην οποία ξεκινάει η ανεστραμμένη λίστα του όρου μέσα στο ανεστραμμένο αρχείο, το μέγεθος της λίστας, το πλήθος των κειμένων στα οποία εμφανίζεται ο όρος, το συνολικό πλήθος των εμφανίσεων του όρου σε όλα τα κείμενα, καθώς και το τελευταίο αναγνωριστικό κειμένου που κωδικοποιήσαμε για τον συγκεκριμένο όρο. Το μέγεθος κάθε κόμβου του B-tree είναι ορισμένο από το Zettair στα 8Kb. Στο Σχήμα 5.2 παρουσιάζεται η εσωτερική δομή ενός κόμβου του B-tree.



Σχήμα 5.2: (α) Εσωτερική δομή ενός κόμβου του B-tree.

5.2 Τροποποίηση του Zettair

Από το σύστημα του Zettair διατηρήσαμε την διαδικασία της λεκτικής ανάλυσης των κειμένων σε εμφανίσεις και της συλλογής αυτών στη μνήμη, στον πίνακα κατακερματισμού. Αντικαταστήσαμε τον κώδικα ο οποίος εκτελείται όταν η μνήμη εξαντληθεί (δημιουργία διαμερίσεων στο δίσκο) με τον κώδικα του αλγορίθμου Πρωτέας. Παράλληλα, κάναμε κάποιες τροποποιήσεις στον κώδικα της διαδικασίας συλλογής εμφανίσεων στη μνήμη, όπως περιγράφεται και παρακάτω, ώστε ανά πάσα στιγμή να διατηρούμε διάφορες πληροφορίες

[3]λόγω περιορισμών από το εκάστοτε σύστημα αρχείων, το ανεστραμμένο αρχείο συνήθως αποθηκεύεται σαν ένα σύνολο από αρχεία μεγέθους 2Gb ή 4Gb. Έτσι, ο δείκτης στην θέση μιας λίστας μέσα στο ανεστραμμένο αρχείο συνήθως αποτελείται από ένα ζεύγος ⟨*αριθμός αρχείου, σχετική θέση της λίστας μέσα στο συγκεκριμένο αρχείο*⟩

που αφορούν τα μπλοκς, όπως το πόσες εμφανίσεις συλλέχθηκαν στη μνήμη για ένα συγκεκριμένο μπλοκ.

5.2.1 Δομές

Υποθέτουμε πως η διαδικασία δεικτοδότησης μιας συλλογής κειμένων αποτελείται από δύο ανεξάρτητες διαδικασίες: την διαδικασία λεκτικής ανάλυσης και την διαδικασία δημιουργίας του ευρετηρίου. Η διαδικασία λεκτικής ανάλυσης λαμβάνει σαν είσοδο μία συλλογή κειμένων και αναλύει κάθε κείμενο d σε ένα σύνολο από εμφανίσεις της μορφής $\langle t, d, pos \rangle$, το οποίο σημαίνει πως ο όρος t εμφανίστηκε στο κείμενο d στην θέση pos. Η διαδικασία δημιουργίας του ευρετηρίου λαμβάνει σαν είσοδο μία ροή από εμφανίσεις $\langle t, d, pos \rangle$ οι οποίες παράγονται από την διαδικασία λεκτικής ανάλυσης, και δημιουργεί για κάθε όρο t μία ανεστραμμένη λίστα της μορφής:

$$\langle d_1;\ f_{d_1,t};\ pos_1, ..., pos_{f_{d_1,t}} \rangle, \langle d_2;\ f_{d_2,t};\ pos_1, ..., pos_{f_{d_2,t}} \rangle, ..., \langle d_k;\ f_{d_k,t};\ pos_1, ..., pos_{f_{d_k,t}} \rangle$$

Ο βασικός αλγόριθμος δημιουργίας του ευρετηρίου παρουσιάστηκε στο Σχήμα 4.2: οι εμφανίσεις που παράγονται από την διαδικασία της λεκτικής ανάλυσης συμπιέζονται και συλλέγονται στη μνήμη, με την βοήθεια κατάλληλων βοηθητικών δομών. Όταν το συνολικό μέγεθος των εμφανίσεων στη μνήμη ξεπεράσει το όριο P_t, τότε επιλέγουμε έναν αριθμό από εύρη και αποθηκεύουμε τις εμφανίσεις τους στο δίσκο βάσει του αλγορίθμου Πρωτέας (Σχήμα 4.3). Για κάθε επιλεγμένο εύρος συλλέγουμε όλες τις εμφανίσεις των όρων του από τη μνήμη και τις συγχωνεύουμε με τις αντίστοιχες λίστες που βρίσκονται στα μπλοκ στο δίσκο.

Για να έχουμε την δυνατότητα να γνωρίζουμε ανά πάσα στιγμή διάφορες πληροφορίες σχετικά με τα εύρη που βρίσκονται αποθηκευμένα σε κάθε μπλοκ στο δίσκο, διατηρούμε στη μνήμη μία νέα δομή, τον *πίνακα ευρών* (rangetable). Ο πίνακας ευρών είναι απλά ένας πίνακας που περιέχει για κάθε εύρος στο δίσκο μία ειδική δομή την οποία ονομάζουμε *δομή εύρους* (range structure). Για κάθε εύρος R η αντίστοιχη δομή εύρους περιέχει τις εξής πληροφορίες: (α) τον μικρότερο και τον μεγαλύτερο όρο που περιέχεται στο R, (β) το μέγεθος που καταλαμβάνουν στη μνήμη όλες οι εμφανίσεις όρων που ανήκουν στο R, (γ) μία συνδεδεμένη λίστα με δείκτες σε όλους τους κόμβους του πίνακα κατακερματισμού που ανήκουν στο R (*λίστα δομών εμφανίσεων* του R), (δ) έναν δείκτη στο μπλοκ του δίσκου στο οποίο βρίσκονται αποθηκευμένες οι λίστες όλων των όρων του R. Αν το R είναι ένα μεγάλο εύρος, τότε διατηρούμε μία *λίστα με δείκτες* σε όλα τα μπλοκ του R στο δίσκο, (ε) το μέγεθος του ελεύθερου χώρου που υπάρχει στο τέλος του μπλοκ στο δίσκο (για τα

μεγάλα εύρη: το μέγεθος του ελεύθερου χώρου που υπάρχει στο τελευταίο μπλοκ του R στο δίσκο).

Στο Σχήμα 5.3 παρουσιάζονται οι βασικές δομές του συστήματος, η δομή εύρους και η δομή εμφανίσεων.

```
range structure {
   min_term            /* the lexicographically smallest term contained in range */
   max_term            /* the lexicographically biggest term contained in range */
   mem_postings        /* list with all range's postings structures in hashtable */
   mem_postings_size   /* total amount of memory allocated for range's postings */
   block_pointer       /* pointer (or list of pointers) to range's block(s) on disk */
   block_free_space    /* amount of free space at the end of the block on disk */
}
```

(α)

```
postings structure {
   term                /* the term this structure accumulates postings for */
   postings_list       /* term's accumulated postings */
   num_postings        /* number of accumulated postings */
   last_docno          /* last document number encoded */
   last_pos            /* last position encoded */
}
```

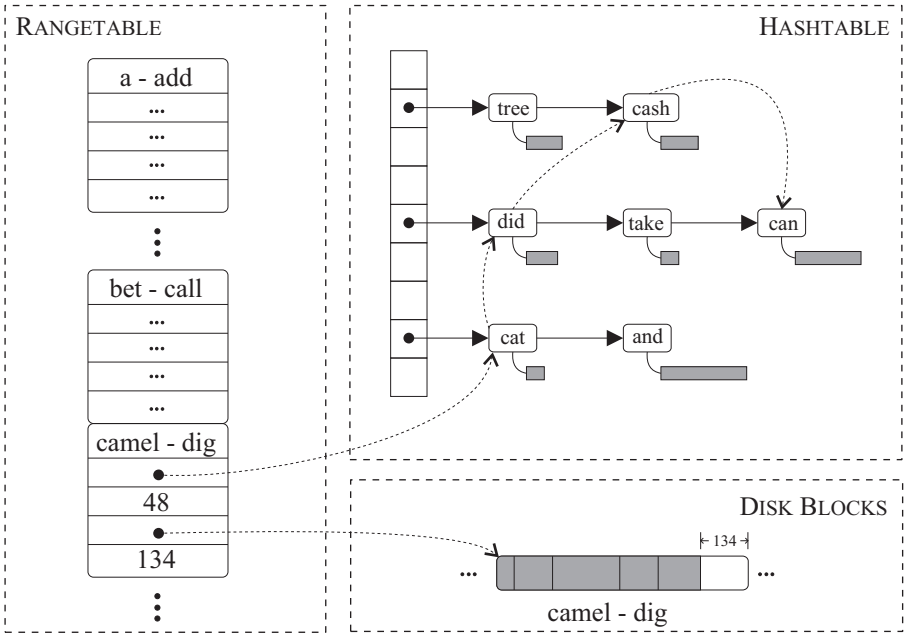
(β)

Σχήμα 5.3: Βασικές πληροφορίες που διατηρούνται (α) για κάθε εύρος στον πίνακα ευρών, (β) για κάθε όρο στον πίνακα κατακερματισμού.

Όπως αναφέραμε παραπάνω, η δομή ευρους διατηρεί όλες τις απαραίτητες πληροφορίες για ένα εύρος, ενώ η δομή εμφανίσεων επιτρέπει την συλλογή και άμεση συμπίεση των εμφανίσεων ενός όρου στη μνήμη. Ο πίνακας ευρών περιέχει μία δομή εύρους για κάθε εύρος, ενώ ο πίνακας κατακερματισμού περιέχει μία δομή εμφάνισης για κάθε όρο για τον οποίο έχουν συλλεχθεί κάποιες εμφανίσεις στη μνήμη (και όχι γενικά για κάθε όρο). Οι δομές εύρους στον πίνακα ευρών διατηρούνται σε λεξικογραφική διάταξη ως προς τον μικρότερο όρο τον οποίο περιέχει κάθε εύρος (R.min_term), έτσι ώστε η εύρεση του εύρους στο οποίο ανήκει ένας όρος t να μπορεί να γίνει αποδοτικά μέσω μιας δυαδικής αναζήτησης του t στον πίνακα ευρών. Στο Σχήμα 5.4 παρουσιάζονται οπτικά οι παραπάνω δομές.

5.2.2 Συλλογή εμφανίσεων

Στο Σχήμα 5.5 παρουσιάζεται η διαδικασία λεκτικής ανάλυσης, η οποία παίρνει σαν είσοδο μία συλλογή κειμένων. Για κάθε νέο όρο t που εξάγεται από κάποιο κείμενο για πρώτη



Σχήμα 5.4: Πίνακας κατακερματισμού και πίνακας ευρών.

φορά δημιουργούμε μία δομή εμφανίσεων P_t στον πίνακα κατακερματισμού (βήμα 7) η οποία θα συλλέγει τις εμφανίσεις του όρου. Έπειτα, διενεργούμε μία δυαδική αναζήτηση στον πίνακα ευρών για να εντοπίσουμε το εύρος R στο οποίο ανήκει ο όρος και τοποθετούμε έναν δείκτη από την δομή εμφανίσεων P_t του t προς το R (βήματα 8–9). Ο δείκτης αυτός παρέχει γρήγορη πρόσβαση στο εύρος του t για μελλοντικές χρήσεις. Τέλος, προσθέτουμε[4] την δομή εμφανίσεων P_t στην λίστα δομών εμφανίσεων του R (βήματα 10–11), η οποία παρέχει γρήγορη[5] πρόσβαση σε όλες τις δομές εμφανίσεων του συγκεκριμένου εύρους στη μνήμη, σε περίπτωση για παράδειγμα που θέλουμε να συλλέξουμε όλες τις εμφανίσεις του R για να τις αποθηκεύσουμε στο δίσκο.

Όταν η λεκτική ανάλυση ενός κειμένου εξάγει μία νέα εμφάνιση $\langle t, d, pos\rangle$ για έναν όρο t, τότε αφού εντοπίσουμε (βήμα 13) ή δημιουργήσουμε (βήματα 7–11) την δομή εμφανίσεων P_t του t, προσθέτουμε την νέα εμφάνιση $\langle d, pos\rangle$ στις υπάρχουσες εμφανίσεις του όρου οι οποίες διατηρούνται στη δομή P_t (βήματα 15–18). Έπειτα, πηγαίνουμε στο εύρος R στο οποίο ανήκει ο όρος και ενημερώνουμε το πεδίο R.mem_postings_size που περιέχει το συνολικό μέγεθος των εμφανίσεων των όρων του R στη μνήμη (βήμα 19).

Καθώς προστίθενται νέες εμφανίσεις στη μνήμη, ενημερώνουμε μία μεταβλητή σχετικά με το συνολικό μέγεθος που καταλαμβάνουν όλες οι εμφανίσεις στη μνήμη (βήμα 20). Κάθε

[4] συγκεκριμένα, προσθέτουμε την νέα δομή εμφάνισης στην αρχή της λίστας του R, σε χρόνο $O(1)$.

[5] με την βοήθεια της λίστας αυτής μπορούμε να προσπελάσουμε όλες τις δομές εμφανίσεων στη μνήμη ενός συγκεκριμένου εύρους σε χρόνο $O(N)$, όπου N το πλήθος των δομών εμφανίσεων του.

```
    function index_collection (input: documents set D)
 1.    total_memory_postings_size ← 0
 2.    While D ≠ ∅
 3.       d ← next document from D
 4.       D ← D − {d}
 5.       For each posting ⟨t, d, pos⟩ extracted from d by the parsing process
 6.           If t does not exist in hashtable
 7.              Create a postings structure P_t for t in hashtable
 8.              Find the range R that t belongs to, using the rangetable
 9.              P_t.range ← R                /* add a pointer from P_t to R */
10.              P_t.next ← R.mem_postings    /* add P_t to R's in-memory
11.              R.mem_postings ← P_t          * postings structures */
12.          Else
13.              Find t's postings structure P_t in hashtable
14.          End if
15.          Compress ⟨d − P_t.last_docno, pos − P_t.last_pos⟩ and
             append it to P_t.postings_list
16.          P_t.num_postings ← P_t.num_postings + 1
17.          P_t.last_docno ← d
18.          P_t.last_pos ← pos
19.          (P_t.range).mem_postings_size ← (P_t.range).mem_postings_size + |⟨d, pos⟩|
20.          total_mem_postings_size ← total_mem_postings_size + |⟨d, pos⟩|
21.       End for
22.       If total_mem_postings_size > Pt
23.          proteus_algorithm()
24.       End if
25.   End while
```

Σχήμα 5.5: Διαδικασία δημιουργίας ευρετηρίου.

φορά που ολοκληρώνεται η λεκτική ανάλυση ενός κειμένου, ελέγχουμε αν το συνολικό μέγεθος των εμφανίσεων ξεπέρασε το όριο P_t, και αν αυτό ισχύει εκτελούμε τον αλγόριθμο Πρωτέας για να επιλέξουμε κάποια εύρη και να αποθηκεύσουμε τις εμφανίσεις τους στο δίσκο (βήματα 22–24).

Για κάθε όρο για τον οποίον έχουμε συλλέξει κάποιες εμφανίσεις στη μνήμη διατηρούμε μία επιπλέον βοηθητική δομή η οποία προφανώς δεσμεύει κάποιον επιπλέον χώρο στη μνήμη. Στα πειράματα μας, το πλήθος των όρων στη μνήμη (και αντίστοιχα, το πλήθος των δομών εμφανίσεων) μεταβάλλεται σημαντικά κατά την διάρκεια της δεικτοδότησης, σε αντίθεση με το πλήθος των εμφανίσεων το οποίο φράσσουμε ορίζοντας ένα μέγιστο μέγεθος προσωρινής μνήμης P_t για την συγκέντρωση τους. Έτσι, στο σύστημά μας η παράμετρος P_t αναφέρεται στο συνολικό μέγεθος των εμφανίσεων στη μνήμη και *δεν συμπεριλαμβάνει το μέγεθος των βοηθητικών δομών*. Παρόλα αυτά, υποθέτουμε πως το P_t είναι αρκετά μικρό ώστε να εξασφαλίζει πως υπάρχει αρκετός χώρος για τις υπόλοιπες βοηθητικές δομές.

5.2.3 Δημιουργία και εισαγωγή νέου εύρους

Νέα εύρη δημιουργούνται κάθε φορά που μία μικρή λίστα αλλάζει κατηγορία και γίνεται μεγάλη (οπότε και το μικρό εύρος στο οποίο ανήκε διασπάται σε τρία εύρη) ή κάθε φορά που το μπλοκ ενός μικρού εύρους υπερχειλίζει μετά από μία συγχώνευση (οπότε και το εύρος διασπάται σε δύο εύρη – βλ. Ενότητα 4.3.1). Στις περιπτώσεις αυτές πρέπει να δημιουργηθούν μία ή περισσότερες νέες δομές εύρους και να εισαχθούν στον πίνακα ευρών. Για να μειώσουμε το κόστος εισαγωγής νέων ευρών στον πίνακα ευρών, ο πίνακας δε περιέχει τις ίδιες τις δομές ευρών αλλά *δείκτες* στις δομές ευρών. Έτσι, η δημιουργία ενός νέου εύρους περιλαμβάνει τη δημιουργία μίας νέας δομής εύρους στη μνήμη και την εισαγωγή στον πίνακα ευρών ενός δείκτη σε αυτή τη δομή. Φυσικά, ο δείκτης πρέπει να εισαχθεί σε ένα συγκεκριμένο σημείο του πίνακα έτσι ώστε τα εύρη να εξακολουθούν να βρίσκονται στον πίνακα σε λεξικογραφική διάταξη. Έτσι, η εισαγωγή του νέου εύρους περιλαμβάνει την εύρεση του σημείου στον πίνακα στο οποίο θα εισαχθεί ο δείκτης, την ολίσθηση ενός πλήθους δεικτών, και τέλος την εισαγωγή του νέου δείκτη. Η μελέτη και χρήση αποδοτικότερων δομών ανήκει επίσης στην κατηγορία "μελλοντική εργασία".

Καθώς νέοι όροι προστίθενται στο λεξικό, οι κόμβοι του B-tree μπορεί να διασπώνται ανάλογα. Οι διασπάσεις των κόμβων του B-tree είναι εντελώς ανεξάρτητες από τις διασπάσεις των ευρών, και εξαρτώνται αποκλειστικά από το πότε υπερχειλίζει ένας κόμβος του B-tree.

5.2.4 Αποθήκευση ευρών στο δίσκο

Όταν η προσωρινή μνήμη (χωρητικότητας P_t) για την συλλογή των εμφανίσεων γεμίσει, χρησιμοποιούμε δύο βοηθητικούς πίνακες για να ταξινομήσουμε το σύνολο των μικρών και αντίστοιχα μεγάλων ευρών ανάλογα με το πλήθος των εμφανίσεων τους (Σχήμα 4.3, βήματα 2–3). Βάσει του αλγορίθμου Πρωτέας, σε κάθε βήμα επιλέγουμε μεταξύ του μέγιστου (βάσει του πλήθους εμφανίσεων) μικρού και μέγιστου μεγάλου εύρους για το ποιο από τα δύο θα αποθηκεύσουμε στο δίσκο, με την βοήθεια των δυο αυτών πινάκων.

Όταν επιλέξουμε ένα εύρος R για να αποθηκεύσουμε τις εμφανίσεις του στο δίσκο, τότε χρησιμοποιούμε την λίστα δομών εμφανίσεων του R.mem_postings για να συλλέξουμε αποδοτικά από την μνήμη όλους τους όρους του μαζί με τις εμφανίσεις τους (Σχήμα 5.4) – προφανώς στην περίπτωση ενός μεγάλου εύρους συλλέγουμε μόνο έναν όρο. Έπειτα, αντιγράφουμε τους όρους μαζί με τις εμφανίσεις τους σε έναν πίνακα και ταξινομούμε λεξικογραφικά τον πίνακα αυτό, ώστε να μπορούμε να διασχίσουμε βάσει λεξικογραφικής

διάταξης όλους τους όρους του R από τη μνήμη. Τέλος, εντοπίζουμε το μπλοκ του εύρους στο δίσκο, χρησιμοποιώντας τον δείκτη R.block_pointer. Ανάλογα με το αν το εύρος είναι μικρό ή μεγάλο, είτε ανακτούμε το μπλοκ στη μνήμη για να κάνουμε τις κατάλληλες συγχωνεύσεις λιστών (Σχήμα 4.4), είτε απλά προσθέτουμε τις νέες εμφανίσεις απευθείας στο δίσκο, στο τέλος του μπλοκ (Σχήμα 4.8). Παρά το γεγονός πως όλες οι εμφανίσεις αποθηκεύονται κωδικοποιημένες στο δίσκο, εξασφαλίζουμε τουλάχιστον πως κάθε μπλοκ μιας μεγάλης λίστας μπορεί να ανακτηθεί και να αποκωδικοποιηθεί αυτόνομα[6], χωρίς να απαιτείται η ανάκτηση των προηγούμενων μπλοκς.

Για να ελέγξουμε αν για κάποιο εύρος θα υπάρξει υπερχείλιση στο μπλοκ στο δίσκο μετά την συγχώνευση του, ελέγχουμε αν ισχύει:

$$R.mem\_postings\_size > R.block\_free\_space$$

Αν ισχύει κάτι τέτοιο, τότε προφανώς οι νέες εμφανίσεις από τη μνήμη δεν χωράνε στον υπάρχοντα ελεύθερο χώρο στο μπλοκ και άρα θα υπάρξει υπερχείλιση του μπλοκ. Στην περίπτωση αυτή δεσμεύουμε ένα ή περισσότερα νέα μπλοκς, και ανάλογα με το είδος του εύρους και το είδος της διάσπασης χωρίζουμε τις εμφανίσεις του εύρους στα μπλοκς.

Ενημέρωση πίνακα κατακερματισμού και λεξικού

Μετά την αποθήκευση ενός εύρους R στο δίσκο διαγράφουμε από τον πίνακα κατακερματισμού όλους τους όρους του, μαζί με τις εμφανίσεις τους και τις δομές εμφανίσεων τους, και ενημερώνουμε τον πίνακα ευρών ώστε το σύστημα μας να βρίσκεται σε συνεπή κατάσταση. Η διαγραφή των εμφανίσεων και των δομών εμφανίσεων ενός εύρους γίνεται σε χρόνο $O(N)$, χρησιμοποιώντας την λίστα R.mem_postings[7]. Τέλος, ενημερώνουμε στο λεξικό τις πληροφορίες για τους όρους τους εύρους, καθώς οι ανεστραμμένες λίστες τους είτε ενημερώθηκαν (προστέθηκαν νέες εμφανίσεις) είτε απλά άλλαξαν θέση στο δίσκο.

[6] ο λόγος για τον οποίο απαιτούμε κάτι τέτοιο είναι για να υπάρχει η δυνατότητα επιλεκτικής ανάκτησης ορισμένων μόνο μπλοκς μιας μεγάλης λίστας (βλ. Ενότητα 7.2 – "Μελλοντική εργασία")

[7] ο πίνακας κατακερματισμού περιλαμβάνει ένα σύνολο από συνδεδεμένες λίστες για την αντιμετώπιση των συγκρούσεων. Οι λίστες αυτές περιέχουν τις δομές εμφανίσεων και είναι διπλά συνδεδεμένες, ώστε να μπορούμε να διαγράψουμε έναν οποιονδήποτε κόμβο σε χρόνο $O(1)$

5.3 Περίληψη

Στο Κεφάλαιο αυτό περιγράφηκαν διάφορες δομές και τεχνικές που χρησιμοποιήθηκαν για την αποδοτική υλοποίηση του αλγορίθμου Πρωτέας.

Η βασική δομή που εισήχθη είναι ο πίνακας ευρών, ένας πίνακας στον οποίο διατηρούμε για κάθε εύρος όλες τις πληροφορίες που απαιτούνται για τον αλγόριθμο συγχώνευσης, όπως το που βρίσκεται το εύρος αποθηκευμένο στον δίσκο, πόσος ελεύθερος χώρος υπάρχει στο μπλοκ του εύρους στο δίσκο, πόσες εμφανίσεις έχουν συλλεχθεί στη μνήμη για το συγκεκριμένο εύρος κτλ. Με τη βοήθεια του πίνακα αυτού, καθώς και των δομών εμφανίσεων που χρησιμοποιούνται για τη συλλογή των εμφανίσεων στη μνήμη, στη συνέχεια περιγράφηκε ο τρόπος με τον οποίο μπορεί να υλοποιηθεί αποδοτικά κάθε βήμα του αλγορίθμου Πρωτέας, από την επιλογή και αποθήκευση των ευρών μέχρι την ενημέρωση των διαφόρων δομών του συστήματος.

Κεφαλαιο 6

Πειραματικα αποτελεσματα

6.1 Περιγραφή πειραμάτων

Για την εκτέλεση των πειραμάτων χρησιμοποιήσαμε HP DL140G3 κόμβους. Το λειτουργικό σύστημα το οποίο είναι εγκατεστημένο στους κόμβους είναι το Debian 4.0 με έκδοση πυρήνα 2.6.18. Κάθε κόμβος περιλαμβάνει έναν τετραπύρηνο επεξεργαστή[1] Intel Xeon E5345 2.33GHz, 2Gb μνήμη RAM και δύο 250Gb 7200 RPM SATA δίσκους μοντέλο WD2500YS (ονομαστικά χαρακτηριστικά: average seek time 8.7ms, buffer size 16Mb,

[1] το σύστημα μας χρησιμοποιεί μόνο έναν επεξεργαστή

transfer rate 70Mb/s). Η συλλογή αρχείων και το τελικό ευρετήριο αποθηκεύονται σε ξεχωριστούς δίσκους.

Εφόσον χρησιμοποιούμε το προεπιλεγμένο σύστημα αρχείων του Linux (ext3), δημιουργούμε προκαταβολικά αρκετά αρχεία των 4 Gb για την αποθήκευση του ανεστραμμένου αρχείου, για να αποφύγουμε επιβαρύνσεις ή άλλες επιπλοκές από πιθανό εξωτερικό κατακερματισμό των αρχείων. Η προσέγγιση αυτή μελετάται στη Ενότητα 6.7.

Για συλλογή αρχείων χρησιμοποιήσαμε τα πρώτα 100Gb της συλλογής GOV2 από το TREC Terabyte track. Το συγκεκριμένο μέγεθος συλλογής έχει χρησιμοποιηθεί σε προηγούμενες έρευνες [13, 15] και επιτρέπει σχετικές συγκρίσεις με αντίστοιχες μελέτες προηγούμενων εργασιών. Επίσης, λόγω της φύσης και του πλήθους των πειραμάτων, το μέγεθος αυτό επιτρέπει την ολοκλήρωση των πειραμάτων σε εύλογο χρονικό διάστημα (κάθε πείραμα με 100Gb κειμένων απαιτεί 2.5 με 3 ώρες, και εκτελέσαμε περισσότερα από 50 πειράματα).

Πίνακας 6.1: Συμβολισμοί ανάλυσης.

Σύμβολο	Περιγράφή
T_{build}	συνολικός χρόνος που απαιτείται για την δημιουργία του ευρετηρίου
T_{parse}	χρόνος για την λεκτική ανάλυση των αρχείων της συλλογής
T_{merge}	χρόνος για την συγχώνευση των ευρών
$T_{merge,prepare}$	χρόνος για την προετοιμασία της συγχώνευσης των ευρών
$T_{merge,short}$	χρόνος για την συγχώνευση των μικρών ευρών
$T_{merge,long}$	χρόνος για την συγχώνευση των μεγάλων ευρών
T_{btree}	χρόνος για την ενημέρωση του B-tree
t_{short}	μέσος χρόνος για την συγχώνευση ενός μικρού εύρους
t_{long}	μέσος χρόνος για την συγχώνευση ενός μεγάλου εύρους
R_{total}	πλήθος ευρών που υπάρχουν στον πίνακα ευρών
R_{short}	πλήθος μικρών ευρών που υπάρχουν στον πίνακα ευρών
R_{long}	πλήθος μεγάλων ευρών που υπάρχουν στον πίνακα ευρών
M_{total}	συνολικός αριθμός ευρών που συγχωνεύτηκαν
M_{short}	συνολικός αριθμός μικρών ευρών που συγχωνεύτηκαν
M_{long}	συνολικός αριθμός μεγάλων ευρών που συγχωνεύτηκαν
F_{total}	ποσοστό εσωτερικού κατακερματισμού στο ανεστραμμένο αρχείο
F_{short}	ποσοστό εσωτερικού κατακερματισμού στα μικρά μπλοκς
F_{long}	ποσοστό εσωτερικού κατακερματισμού στα μεγάλα μπλοκς
N_{short}	ποσοστό εσωτερικού κατακερματισμού στα μικρά μπλοκς
N_{long}	ποσοστό εσωτερικού κατακερματισμού στα μεγάλα μπλοκς

Για την περιγραφή των αποτελεσμάτων χρησιμοποιούμε τους συμβολισμούς που παρουσιάζονται στον Πίνακα 6.1.

Θεωρούμε το εξής απλοποιημένο μοντέλο για την ανάλυση που ακολουθεί:

$$T_{build} = T_{parse} + T_{merge} + T_{btree} \tag{6.1}$$

$$T_{merge} = T_{merge,prepare} + T_{merge,short} + T_{merge,long} = \tag{6.2}$$

$$= T_{merge,prepare} + (M_{short} \times t_{short}) + (M_{long} \times t_{long}) \tag{6.3}$$

Σύμφωνα με το παραπάνω μοντέλο, θεωρούμε πως ο χρόνος T_{build} για την δημιουργία του ευρετηρίου ισούται με το άθροισμα του χρόνου T_{parse} που απαιτείται για την λεκτική ανάλυση των κειμένων σε ένα σύνολο εμφανίσεων, του χρόνου T_{merge} που απαιτείται για την αποθήκευση (συγχώνευση) των εμφανίσεων από τη μνήμη στο δίσκο, και του χρόνου T_{btree} που απαιτείται για την ενημέρωση του B-tree.

Με την σειρά του, ο χρόνος T_{merge} ισούται με το άθροισμα των χρόνων $T_{merge,short}$ και $T_{merge,long}$ που απαιτούνται για την αποθήκευση των μικρών και αντίστοιχα μεγάλων ευρών από τη μνήμη στο δίσκο, καθώς και του χρόνου $T_{merge,prepare}$ που απαιτείται για την προετοιμασία της διαδικασίας συγχώνευσης (π.χ. επιλογή των ευρών που θα συγχωνευτούν), η οποία εκτελείται κάθε φορά που εξαντλείται η μνήμη.

Τέλος, ο χρόνος $T_{merge,short}$ ισούται με το γινόμενο του πλήθους M_{short} των μικρών ευρών που αποθηκεύονται στον δίσκο επί τον μέσο χρόνο αποθήκευσης ενός μικρού εύρους t_{short}, και με παρόμοιο τρόπο αναλύεται και ο χρόνος $T_{merge,long}$.

Θεωρώντας πως ο χρόνος T_{parse} που απαιτείται για την λεκτική ανάλυση των αρχείων είναι σχετικά σταθερός και ανεξάρτητος από τις παραμέτρους των πειραμάτων (γεγονός το οποίο επιβεβαιώνεται και από τα πειραματικά αποτελέσματα), ο συνολικός χρόνος για την δημιουργία του ευρετηρίου εξαρτάται τελικά κυρίως από τις τιμές των M_{short}, M_{long}, t_{short}, t_{long}, καθώς και από τις τιμές των T_{btree} και $T_{merge,prepare}$.

Οι τιμές που χρησιμοποιούνται στα πειράματα για τις διάφορες παραμέτρους του συστήματος είναι οι ακόλουθες, εκτός και αν αναφέρεται ρητά κάποια διαφορετική τιμή:

- Προσωρινή Μνήμη $P_t = 512\text{Mb}$
- Μνήμη Συγχώνευσης $P_f = 10\text{Mb}$
- Μέγεθος Μπλοκ $B_p = 1\text{Mb}$
- Αναλογία Κόστους Συγχώνευσης $K_t = 1.7$

- Κατώφλι Μεγάλων Λιστών $T_t = 30\% \times B_p$

Όλα τα αποτελέσματα των πειραμάτων παρουσιάζονται συγκεντρωτικά με την μορφή πινάκων στο Παράρτημα. Όλοι οι χρόνοι που παρουσιάζονται στα πειράματα αντιστοιχούν σε *δευτερόλεπτα*.

6.2 Μέγεθος μπλοκ

Η σημαντικότερη ίσως παράμετρος του συστήματος, η οποία επηρεάζει τόσο τον συνολικό χρόνο δημιουργίας του ευρετηρίου όσο και τον χρόνο ανάκτησης των ανεστραμμένων λιστών κατά την αναζήτηση κειμένων, καθώς και το μέγεθος του παραγόμενου ανεστραμμένου αρχείου, είναι το μέγεθος του μπλοκ B_p. Παρακάτω αναλύεται ο τρόπος με τον οποίο επιδρά η παράμετρος αυτή σε κάθε μία από τις προαναφερθείσες περιπτώσεις, ενώ επίσης παρουσιάζονται πειραματικά αποτελέσματα που επιβεβαιώνουν την ανάλυση.

6.2.1 Χρόνος συγχώνευσης ευρών

Επίδραση στο πλήθος των ευρών που συγχωνεύονται

Το μέγεθος του μπλοκ καθορίζει έμμεσα την τιμή του ορίου T_t το οποίο διαχωρίζει τις λίστες σε μικρές και μεγάλες. Συγκεκριμένα, το όριο T_t ορίζεται ως το γινόμενο ενός ποσοστού α $(0 < \alpha < 1)$ επί το μέγεθος του μπλοκ:

$$T_t = \alpha \times B_p$$

και ορίζει πως κάθε λίστα που ξεπερνά σε μέγεθος το ποσοστό α του μεγέθους του μπλοκ θεωρείται μεγάλη λίστα. Θέτοντας στην παράμετρο α μία συγκεκριμένη τιμή (π.χ. 30%) και διατηρώντας την τιμή αυτή σταθερή για το σύνολο των πειραμάτων που αφορούν την επίδραση του μεγέθους του μπλοκ, το όριο T_t εξαρτάται αποκλειστικά από το μέγεθος του μπλοκ (η επίδραση της παραμέτρου α μελετάτε στην Ενότητα 6.3).

Βάσει της παραπάνω σχέσης που συνδέει τις παραμέτρους T_t και B_p, όσο μεγαλύτερο είναι το μέγεθος του μπλοκ τόσο μεγαλύτερο είναι και το όριο T_t το οποίο ορίζει το πότε μία λίστα θεωρείται μεγάλη. Η συλλογή κειμένων που χρησιμοποιείται είναι ίδια για όλα τα πειράματα, και άρα η κατανομή του μεγέθους των λιστών παραμένει επίσης ίδια (εφόσον είναι ανεξάρτητη από τις παραμέτρους των πειραμάτων). Το τελευταίο σημαίνει πως *αυξάνοντας το μέγεθος* του μπλοκ αυξάνεται το T_t, και άρα *μειώνεται το πλήθος* των

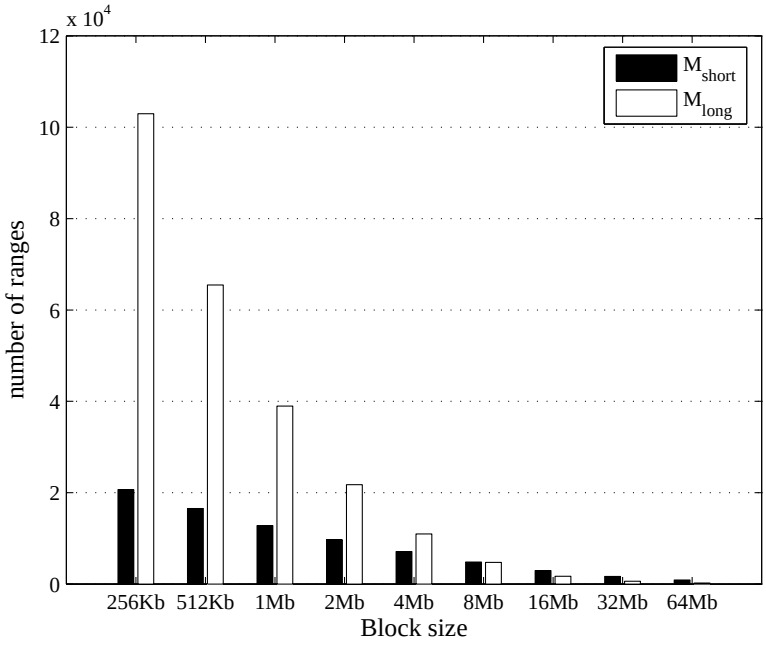
μεγάλων λιστών. Λόγω της ένα προς ένα αντιστοιχίας μεταξύ μεγάλων λιστών και μεγάλων ευρών, αυτό έχει σαν αποτέλεσμα τελικά να μειώνεται το πλήθος των μεγάλων ευρών R_{long}.

Καθώς αυξάνεται το μέγεθος του μπλοκ, εκτός από το πλήθος R_{long} των μεγάλων ευρών μειώνεται επίσης και το πλήθος R_{short} των μικρών ευρών. Η μείωση αυτή είναι αποτέλεσμα δύο συνιστωσών:

- έχοντας μεγαλύτερο μέγεθος μπλοκ, τα μπλοκς που περιέχουν μικρές λίστες υπερχειλίζουν λιγότερο συχνά, με αποτέλεσμα να μειώνεται ο ρυθμός με τον οποίο τα μικρά εύρη διασπώνται λόγω της υπερχείλισης των μπλοκ τους και συνεπακόλουθα να μειώνεται και το πλήθος των μικρών ευρών.

- καθώς μειώνεται ο ρυθμός με τον οποίο δημιουργούνται μεγάλες λίστες (λίστες με μέγεθος μεγαλύτερο από T_t), μειώνεται ταυτόχρονα και ο ρυθμός με τον οποίο τα μικρά εύρη διασπώνται λόγω της εμφάνισης μεγάλων λιστών

Συνοπτικά, το πλήθος των μικρών ευρών εξαρτάται από το πλήθος των διασπάσεων τους. Μία τέτοια διάσπαση μπορεί να συμβεί είτε επειδή υπερχείλισε το μπλοκ ενός μικρού εύρους, είτε επειδή εμφανίστηκε ένας μεγάλος όρος σε ένα μικρό εύρος. Αυξάνοντας το μέγεθος του μπλοκ μειώνεται αυτόματα ο ρυθμός με τον οποίον συμβαίνουν και τα δύο είδη διασπάσεων.

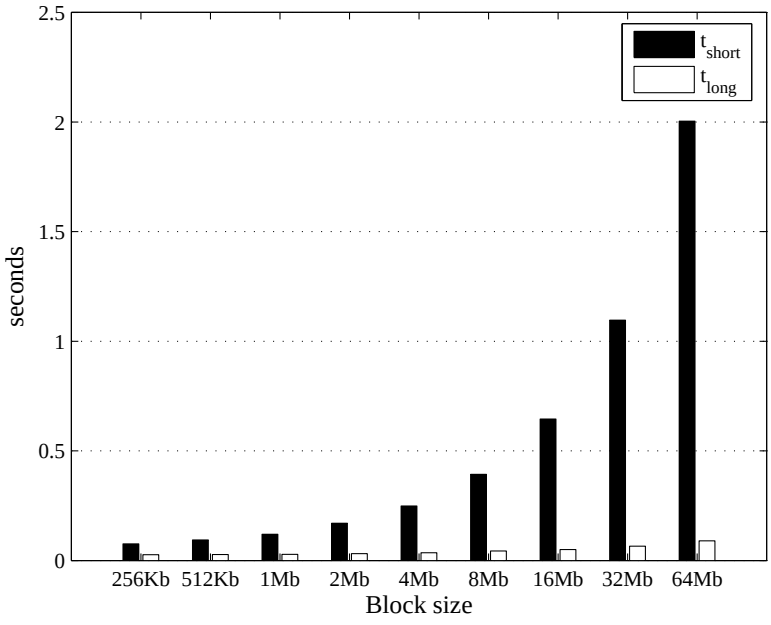


Σχήμα 6.1: Επίδραση του μεγέθους του μπλοκ στο συνολικό αριθμό των ευρών που συγχωνεύουμε.

Η μείωση του πλήθους R_{short} και R_{long} των μικρών και μεγάλων ευρών αντίστοιχα που περιέχονται στον πίνακα ευρών έχει σαν άμεση επίπτωση την *μείωση* του αριθμού των ευρών M_{short} και M_{long} που συγχωνεύονται κατά την διάρκεια της δημιουργίας του ευρετηρίου, καθώς το μέγεθος του μπλοκ αυξάνεται (Σχήμα 6.1).

Επίδραση στο μέσο χρόνο συγχώνευσης των ευρών

Ταυτόχρονα με την μείωση του πλήθους των μικρών και μεγάλων ευρών που συγχωνεύονται, η αύξηση του μεγέθους του μπλοκ έχει ως αποτέλεσμα να *αυξάνεται* ο μέσος χρόνος που απαιτείται για να συγχωνεύσουμε ένα εύρος (Σχήμα 6.2). Για τα μικρά εύρη, η αύξηση αυτή είναι πιο έντονη για τον εξής λόγο: αν θεωρήσουμε πως σε κάθε συγχώνευση ενός μικρού εύρους διαβάζουμε κατά μέσο όρο το μισό μπλοκ (έστω $B_p/2$ bytes), συγχωνεύουμε τις εμφανίσεις από τη μνήμη (έστω C bytes) με τις λίστες από τον δίσκο, και ξαναγράφουμε τις λίστες στο δίσκο ($B_p/2 + C$ bytes), τότε προφανώς με μεγαλύτερα μεγέθη μπλοκ αυξάνεται το πλήθος των bytes που διαβάζουμε και γράφουμε σε κάθε συγχώνευση (αφού αυξάνεται τόσο το $B_p/2$, όσο και το C, το πόσες εμφανίσεις συγκεντρώνονται στη μνήμη για ένα εύρος, εφόσον τα εύρη περιλαμβάνουν περισσότερους όρους).



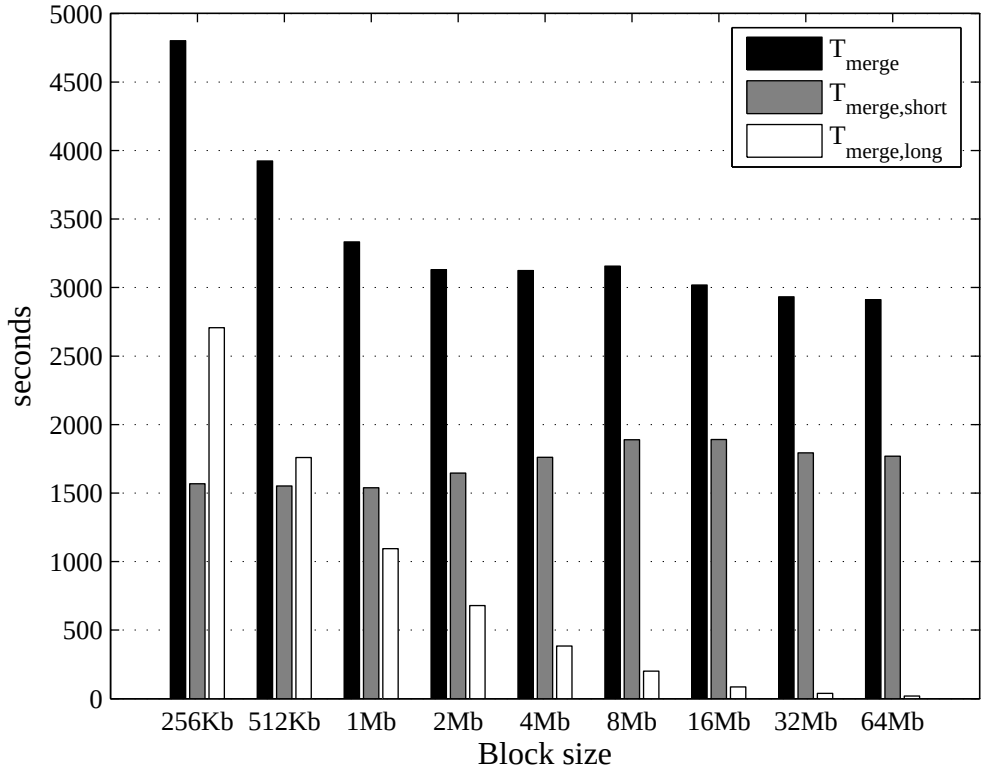
Σχήμα 6.2: Επίδραση του μεγέθους του μπλοκ στο μέσο χρόνο συγχώνευσης ενός εύρους.

Έτσι βλέπουμε πως υπάρχουν δύο τάσεις: από την μία η *μείωση του πλήθους* των ευρών

που συγχωνεύονται, και από την άλλη η *αύξηση του μέσου χρόνου* που απαιτείται για την συγχώνευση ενός εύρους (συγχωνεύουμε λιγότερα εύρη, αλλά με περισσότερες εμφανίσεις το καθένα).

Υπενθυμίζουμε ότι από την εξίσωση 6.3 προκύπτει ότι το πλήθος M_{short} και M_{long} των μικρών και μεγάλων ευρών αντίστοιχα που συγχωνεύονται, καθώς και ο μέσος χρόνος συγχώνευσης t_{short} και t_{long} ενός μικρού και ενός μεγάλου εύρους αντίστοιχα, είναι οι κύριες παράμετροι[2] που επηρεάζουν τον συνολικό χρόνο συγχώνευσης T_{merge}. Έτσι, το αν τελικά ο συνολικός χρόνος συγχώνευσης θα αυξηθεί ή θα μειωθεί εξαρτάται από τους ρυθμούς μείωσης και αύξησης των τιμών των παραμέτρων αυτών.

Επίδραση στο συνολικό χρόνο συγχώνευσης



Σχήμα 6.3: Επίδραση του μεγέθους του μπλοκ στο χρόνο συγχώνευσης των μικρών και μεγάλων ευρών, καθώς και στο συνολικό χρόνο συγχώνευσης.

Στο Σχήμα 6.3 παρατηρούμε πως ο χρόνος $T_{merge,long}$ που απαιτείται για την συγχώνευση των μεγάλων ευρών *μειώνεται* συνεχώς με την αύξηση του μεγέθους του μπλοκ, και αυτό

[2] φυσικά μαζί με τον χρόνο ενημέρωσης $T_{merge,prepare}$, ο οποίος όμως μέχρι το $B_p = 1\text{Mb}$ είναι τάξης μικρότερος του T_{merge}, ενώ μετά είναι γενικά σταθερός

γιατί το πλήθος των μεγάλων ευρών M_{long} μειώνεται με πολύ γρηγορότερο ρυθμό απ' ότι αυξάνεται ο μέσος χρόνος συγχώνευσης t_{long} ενός μεγάλου εύρους: το t_{long} αυξάνεται κατά 5%–10% σε κάθε βήμα, ενώ το M_{long} μειώνεται κατά 40%–50% (βλ. Σχήμα 6.1, 6.2).

Για τον χρόνο $T_{merge,short}$ που απαιτείται για την συγχώνευση των μικρών ευρών παρατηρούμε μία πτώση καθώς το μέγεθος του μπλοκ αυξάνεται από τα 256Kb στο 1Mb, μετά μία αύξηση καθώς το μέγεθος του μπλοκ μεταβάλλεται 1Mb έως 16Mb, και τέλος μία μείωση μέχρι τα 64Mb. Αυτό οφείλεται στο γεγονός πως κατά διαστήματα υπερισχύει η μείωση του M_{short} έναντι της αύξησης του t_{short}, ενώ σε άλλα διαστήματα ισχύει το αντίθετο. Συγκεκριμένα, ενώ από τα 256Kb στο 1Mb το πλήθος M_{short} των μικρών ευρών που συγχωνεύονται μειώνεται κατά 38% και ο μέσος χρόνος συγχώνευσης t_{short} αυξάνεται κατά 57%, και άρα έχουμε μία συνολική μείωση[3] του $T_{merge,short}$, εντούτοις όταν πηγαίνουμε από το 1Mb στα 2Mb έχουμε μία μείωση του M_{short} της τάξεως του 24% ενώ το t_{short} αυξάνεται κατά 42%, αυξάνοντας[4] έτσι το $T_{merge,short}$. Η συμπεριφορά αυτή εξακολουθεί να ισχύει καθώς αυξάνουμε το μέγεθος του μπλοκ μέχρι τα 16Mb, μετά από το οποίο σημείο υπερισχύει η μείωση του M_{short} έναντι της αύξησης του t_{short}.

Άρα, αυτό που παρατηρούμε είναι πως καθώς αυξάνει το μέγεθος του μπλοκ, ο χρόνος $T_{merge,long}$ που δαπανάται για την συγχώνευση των μεγάλων ευρών μειώνεται συνεχώς, ενώ ο χρόνος $T_{merge,short}$ που δαπανάται για την συγχώνευση των μικρών ευρών μειώνεται καθώς το μπλοκ αυξάνεται μέχρι το 1Mb, έπειτα αυξάνεται μέχρι τα 16Mb, και τέλος μειώνεται μέχρι τα 64Mb. Στις περισσότερες περιπτώσεις όπου αυξάνεται το $T_{merge,short}$, η αντίστοιχη μείωση του $T_{merge,long}$ είναι *μεγαλύτερη*, και έτσι ο συνολικός χρόνος συγχώνευσης T_{merge} ο οποίος ισούται με το άθροισμα των $T_{merge,short}$, $T_{merge,long}$ και $T_{merge,prepare}$ τελικά μειώνεται στην γενική περίπτωση, ενώ για 32Mb και 64Mb παρατηρούμε ανεπαίσθητες αλλαγές.

6.2.2 Χρόνος ενημέρωσης του B-tree

Κατά την διαδικασία συγχώνευσης ενός εύρους, αφού προσθέσουμε στις λίστες του τις νέες εμφανίσεις που συγκεντρώθηκαν στη μνήμη και αποθηκεύσουμε τις λίστες πίσω στο δίσκο, πρέπει να ενημερώσουμε το B-tree για τους όρους των οποίων οι λίστες τροποποιήθηκαν (Σχήμα 4.3, βήμα 16). Μία λίστα θεωρείται "τροποποιημένη" είτε επειδή προστέθηκαν σε αυτή κάποιες νέες εμφανίσεις, είτε επειδή απλά άλλαξε θέση στο δίσκο, αφού και στις δύο περιπτώσεις πρέπει να ενημερωθεί στο B-tree η εγγραφή που αφορά τον συγκεκριμένο όρο (Σχήμα 5.2).

[3] αν $t_{(B_p=256Kb)} = M_{short} \cdot t_{short} \;\Rightarrow\; t_{(B_p=1Mb)} = (0.62M_{short}) \cdot (1.57t_{short}) = 0.97(M_{short} \cdot t_{short})$

[4] αν $t_{(B_p=1Mb)} = M_{short} \cdot t_{short} \;\Rightarrow\; t_{(B_p=2Mb)} = (0.76M_{short}) \cdot (1.42t_{short}) = 1.08(M_{short} \cdot t_{short})$

Πίνακας 6.2: Επίδραση του μεγέθους του μπλοκ στο πλήθος των όρων που περιέχει ένα μικρό εύρος και στο συνολικό πλήθος των μικρών ευρών που συγχωνεύονται.

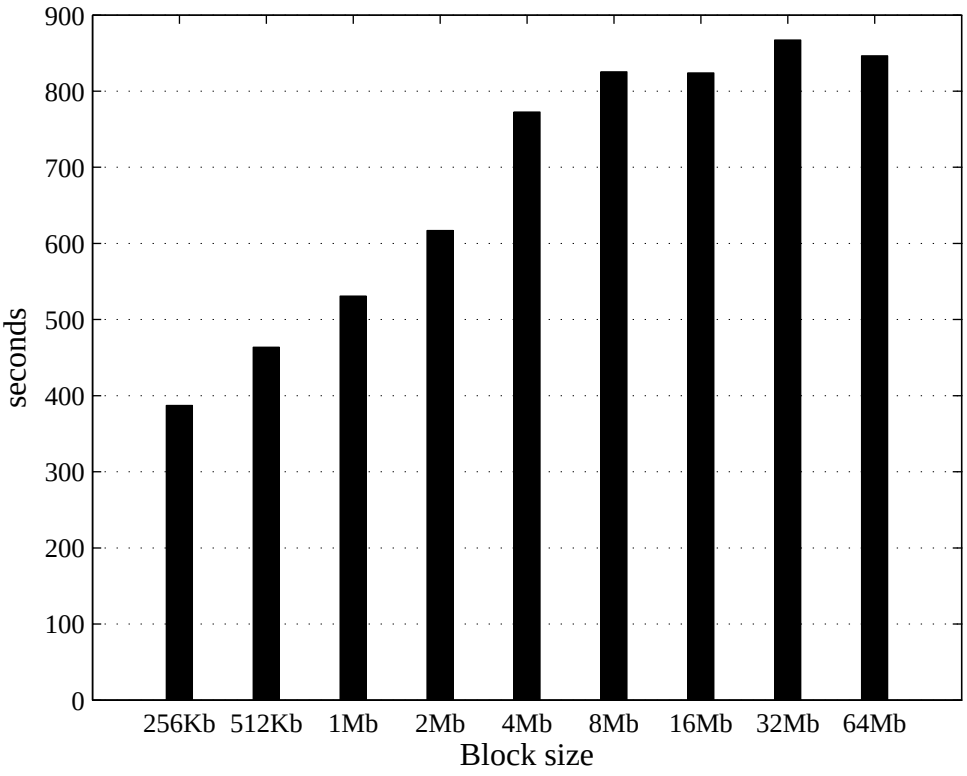
B_p	terms/range	M_{short}
256Kb	1688	20646
512Kb	2820	16535
1Mb	4724	12791
2Mb	7852	9678
4Mb	13012	7087
8Mb	23056	4802
16Mb	41828	2931
32Mb	83157	1636
64Mb	162427	883

Όταν συγχωνεύουμε ένα μεγάλο εύρος, πρέπει να ενημερώσουμε στο B-tree την εγγραφή του μοναδικού όρου που αντιστοιχεί στο συγκεκριμένο εύρος, καθώς έχει αλλάξει το μέγεθος της λίστας του. Όταν συγχωνεύουμε ένα μικρό εύρος, πρέπει να ενημερώσουμε στο B-tree τις εγγραφές *όλων* των όρων που περιέχονται σε αυτό το εύρος, ακόμα και αν σε κάποιους δεν έχει προστεθεί καμία νέα εμφάνιση στις λίστες τους. Αυτό γίνεται γιατί, παρόλο που μπορεί να μην άλλαξε το μέγεθος κάποιας λίστα, κατά πάσα πιθανότητα έχει αλλάξει η θέση στην οποία βρίσκεται αποθηκευμένη στο δίσκο, καθώς με μεγάλη πιθανότητα κάποια λίστα που ήταν αποθηκευμένη πριν από αυτή στο ίδιο μπλοκ θα επεκτάθηκε λόγω της προσθήκης κάποιων νέων εμφανίσεων. Με λίγα λόγια, η προσθήκη κάποιων εμφανίσεων σε μία μικρή λίστα έχει σαν αποτέλεσμα όλες οι λίστες που βρίσκονται αποθηκευμένες στο ίδιο μπλοκ μετά από αυτήν να μετακινηθούν κατά κάποιες bytes. Φυσικά κάποιες λίστες στην αρχή του μπλοκ μπορεί να μείνουν ανεπηρέαστες αν δεν έχουν προστεθεί νέες εμφανίσεις σε καμία λίστα πριν από αυτές, αλλά οι λίστες αυτές θα είναι ελάχιστες, και για λόγους απλότητας θεωρούμε πως μετά την συγχώνευση ενός μικρού εύρους ενημερώνουμε το B-tree για όλους τους όρους του, ακόμα και γι αυτούς που δεν μετακινήθηκαν.

Γίνεται έτσι σαφές πως ο κύριος όγκος των ενημερώσεων του B-tree οφείλεται στα μικρά εύρη, καθώς κάθε φορά που συγχωνεύεται ένα τέτοιο εύρος πρέπει να ενημερώσουμε στο B-tree *όλους* τους όρους του, ενώ αντίθετα για κάθε μεγάλο εύρος που συγχωνεύεται πρέπει να ενημερώσουμε μόνο *έναν* όρο. Άρα το πλήθος των ενημερώσεων του B-tree, και συνεπακόλουθα ο χρόνος που δαπανάται για την ενημέρωση του, εξαρτάται αποκλειστικά από *το πλήθος των όρων* που περιέχει ένα μικρό εύρος και από το *πόσο συχνά συγχωνεύεται*

(δηλαδή από το πλήθος M_{short} των συγχωνεύσεων των μικρών ευρών).

Στον Πίνακα 6.2 φαίνεται η επίδραση του μεγέθους του μπλοκ στις δύο παραπάνω παραμέτρους. Καθώς το μέγεθος του μπλοκ αυξάνεται, κάθε μικρό εύρος περιέχει περισσότερους όρους στο μπλοκ του, το οποίο σημαίνει μεγαλύτερο πλήθος ενημερώσεων του B-tree ανά συγχώνευση, ενώ από την άλλη το πλήθος των ευρών που συγχωνεύονται μειώνεται για τους λόγους που αναφέραμε στην προηγούμενη ενότητα. Συγκεκριμένα, σε κάθε βήμα το πλήθος των όρων ενός μικρού εύρους φαίνεται να *διπλασιάζεται*, ενώ η μείωση του M_{short} είναι περίπου της τάξης του 25%. Η επίδραση που έχουν τελικά οι δύο αυτές τάσεις στο συνολικό χρόνο ενημέρωσης T_{btree} του B-tree παρουσιάζεται στο Σχήμα 6.4, όπου όπως φαίνεται ο χρόνος αυξάνεται συνεχώς, με μία μικρή μείωση για $B_p = 64Mb$.



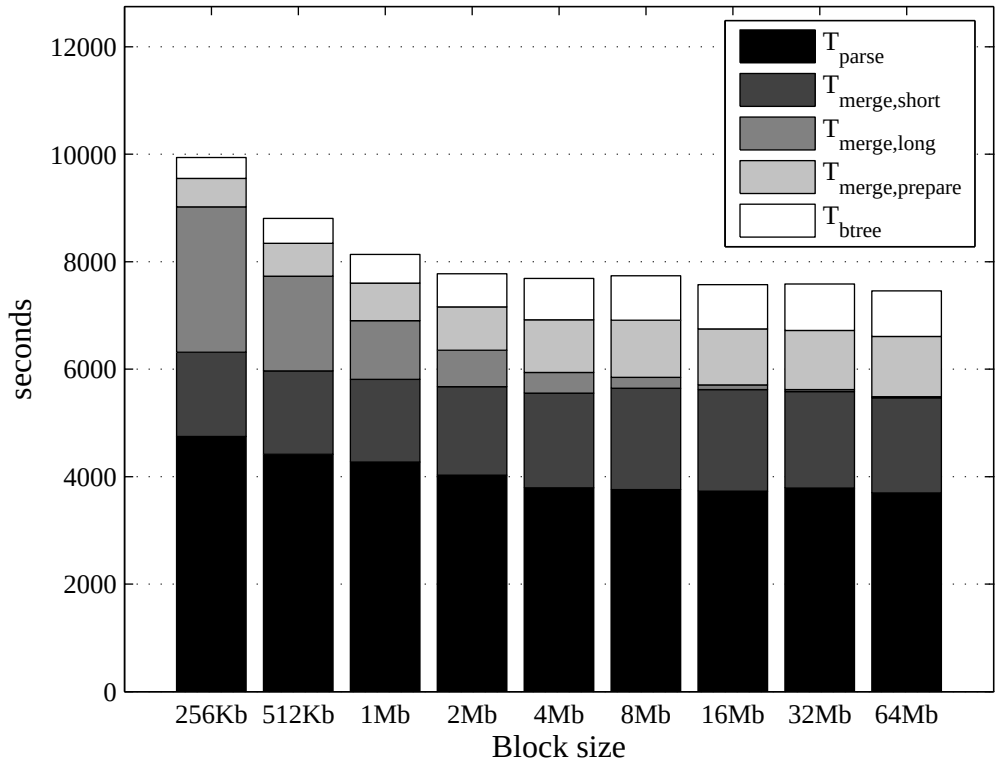
Σχήμα 6.4: Επίδραση του μεγέθους του μπλοκ στο χρόνο ενημέρωσης του B-tree.

6.2.3 Συνολικός χρόνος δημιουργίας ευρετηρίου

Ο συνολικός χρόνος T_{build} που απαιτείται για την δημιουργία του ευρετηρίου εξαρτάται, σύμφωνα με τις εξισώσεις 6.2, 6.3, από τον χρόνο T_{parse} για την λεκτική ανάλυση των κειμένων, τον χρόνο $T_{merge,prepare}$ ο οποίος περιλαμβάνει κάποιες ενέργειες που αφορούν

την προετοιμασία της διαδικασίας συγχώνευσης (ταξινόμηση όλων των ευρών ως προς το μέγεθος των εμφανίσεων τους στη μνήμη, επιλογή των ευρών των οποίων θα συγχωνευθούν, συγκέντρωση από τον πίνακα κατακερματισμού όλων των όρων που ανήκουν σε ένα εύρος το οποίο θα συγχωνευθεί, λεξικογραφική ταξινόμηση των όρων που συλλέχθηκαν), τον χρόνο T_{merge} για την συγχώνευση των ευρών και τον χρόνο T_{btree} για την ενημέρωση του B-tree.

Σύμφωνα με το Σχήμα 6.5, ο χρόνος T_{build} μειώνεται συνεχώς με την αύξηση του B_p, με σχετικά μικρότερους ρυθμούς από τα 4Mb και μετά. Το γεγονός αυτό, όπως φαίνεται και στο Σχήμα 6.3, οφείλεται κυρίως στην μείωση του χρόνου συγχώνευσης T_{merge}, και συγκεκριμένα στην μείωση του $T_{merge,long}$ ο οποίος μειώνεται σημαντικά με την αύξηση του B_p.



Σχήμα 6.5: Ανάλυση του χρόνου δημιουργίας του ευρετηρίου στις διάφορες διαδικασίες.

Ο χρόνος T_{parse} που δαπανάται για την ανάλυση των κειμένων σε εμφανίσεις είναι γενικά σταθερός, εκτός από την περίπτωση που το μέγεθος του μπλοκ είναι 256Kb όπου και παρατηρούμε μία σχετικά αυξημένη τιμή. Ο λόγος για τον οποίον η διαδικασία της λεκτικής ανάλυσης απαιτεί αυξημένο χρόνο στην περίπτωση αυτή είναι επειδή για τόσο μικρά μεγέθη μπλοκ (καθώς και για μικρότερα) παρατηρούνται φαινόμενα χρήσης του *χώρου εναλλαγής*

(swap space). Ο λόγος για τον οποίο συμβαίνει αυτό περιγράφεται στην Ενότητα 6.5.1.

Ο χρόνος T_{btree} που δαπανάται για την ενημέρωση του B-tree αυξάνεται συνεχώς, καθώς σε κάθε συγχώνευση ενός μικρού εύρους πρέπει να ενημερώσουμε όλους τους όρους του και με μεγαλύτερα μεγέθη μπλοκς κάθε μικρό εύρος περιέχει περισσότερους όρους, το οποίο συνεπάγεται αυτόματα περισσότερες ενημερώσεις του B-tree ανά συγχώνευση ενός μικρού εύρους (βλ. Πίνακα 6.2 και Σχήμα 6.4).

Τέλος, ο χρόνος $T_{merge,prepare}$ αυξάνεται με την αύξηση του μπλοκ κυρίως γιατί καθώς αυξάνεται το μπλοκ κάθε μικρό εύρος περιέχει περισσότερους όρους, και άρα κάθε φορά που επιλέγεται ένα μικρό εύρος για να αποθηκευτεί στο δίσκο απαιτείται περισσότερος χρόνος για την συλλογή και κυρίως για την *ταξινόμηση* των όρων του που βρίσκονται στη μνήμη.

6.2.4 Μέγεθος ευρετηρίου

Όπως αναφέρθηκε νωρίτερα, το ευρετήριο αποτελείται από δύο δομές: το ανεστραμμένο αρχείο, που περιέχει την ανεστραμμένη λίστα κάθε όρου που έχει εμφανιστεί μέχρι στιγμής, και το B-tree, το οποίο παρέχει για κάθε όρο την πληροφορία του που ξεκινάει η ανεστραμμένη λίστα του μέσα στο ανεστραμμένο αρχείο.

Το μέγεθος του B-tree είναι σε γενικές γραμμές ανεξάρτητο από τις παραμέτρους του συστήματος, αφού εξαρτάται κυρίως από το πλήθος των όρων που εισάγονται σε αυτό (το οποίο προφανώς είναι σταθερό για όλα τα πειράματα), και τουλάχιστον μία τάξη μικρότερο από το μέγεθος του ανεστραμμένου αρχείου. Για το λόγο αυτό θα εξετάσουμε μόνο την επίδραση του μεγέθους του μπλοκ στο μέγεθος του ανεστραμμένου αρχείου.

Το συνολικό μέγεθος όλων των λιστών που βρίσκονται αποθηκευμένες στο ανεστραμμένο αρχείο εξαρτάται αποκλειστικά και μόνο από την συλλογή των αρχείων που δεικτοδοτούμε και από την μέθοδο συμπίεσης τους, ενώ είναι ανεξάρτητο από τις παραμέτρους του συστήματος. Άρα θα περιμέναμε πως το μέγεθος του ανεστραμμένου αρχείου θα ήταν σταθερό, ανεξαρτήτως των τιμών των παραμέτρων. Όμως, λόγω της χρήσης των μπλοκς για την αποθήκευση των λιστών εμφανίζεται το φαινόμενο του *εσωτερικού κατακερματισμού*: υπάρχουν τμήματα ελεύθερου χώρου στο δίσκο τα οποία δεν μπορούν να χρησιμοποιηθούν. Το φαινόμενο αυτό εμφανίζεται στα μπλοκς, καθώς στο τέλος κάθε μπλοκ υπάρχει κάποιος *ελεύθερος χώρος* ο οποίος όμως *συνυπολογίζεται* στο μέγεθος του αρχείου (ο χώρος αυτός είναι δεσμευμένος παρόλο που ουσιαστικά δεν χρησιμοποιείται).

Έτσι, το συνολικό μέγεθος του ανεστραμμένου αρχείου τελικά εξαρτάται από τις παραμέτρους του συστήματος, και κυρίως από το μέγεθος του μπλοκ B_p, το οποίο επηρεάζει

άμεσα τον εσωτερικό κατακερματισμό των μπλοκς. Από τις υπόλοιπες παραμέτρους, η μοναδική παράμετρος που επηρεάζει σημαντικά τον κατακερματισμό των μπλοκς είναι η παράμετρος α, η οποία σε συνδυασμό με το μέγεθος του μπλοκ ορίζουν το ελάχιστο μέγεθος T_t που πρέπει να έχει μία λίστα για να θεωρηθεί μεγάλη λίστα. Η επίδραση της παραμέτρου αυτής στον κατακερματισμό των μπλοκς μελετάται στην Ενότητα 6.3.

Στο ευρετήριο υπάρχουν δύο ειδών μπλοκς, τα μπλοκς που ανήκουν σε μικρά εύρη και τα μπλοκς που ανήκουν σε μεγάλα εύρη. Λόγω του διαφορετικού αλγορίθμου συγχώνευσης που χρησιμοποιείται για κάθε είδος μπλοκ, το μέγεθος του εσωτερικού κατακερματισμού διαφέρει για ανάλογα με το είδος του μπλοκ.

Εσωτερικός κατακερματισμός στα μικρά μπλοκς

Όπως παρατηρούμε στον Πίνακα 6.3, στα μπλοκς που περιέχουν μικρές λίστες ο κατακερματισμός (F_{short}) είναι αρκετά έντονος, της τάξεως του 50%–60%. Κάθε φορά που υπερχειλίζει το μπλοκ ενός μικρού εύρους, οι λίστες του χωρίζονται –στην γενική περίπτωση– σε δύο σύνολα και αποθηκεύονται σε δύο μπλοκς, καθένα από τα οποία είναι γεμάτο τουλάχιστον 50%. Θα περίμενε λοιπόν κάποιος πως ο κατακερματισμός γι' αυτά τα μπλοκς θα πρέπει να ήταν το πολύ 50% και στην γενική περίπτωση αρκετά μικρότερος. Το πρόβλημα όμως προκύπτει από τις διασπάσεις των ευρών αυτών λόγω της εμφάνισης μεγάλων όρων, και όχι από τις διασπάσεις λόγω υπερχείλισης του μπλοκ.

Πίνακας 6.3: Επίδραση του μεγέθους του μπλοκ στον εσωτερικό κατακερματισμό των μικρών μπλοκς.

B_p	F_{short}
256Kb	63.70 %
512Kb	63.25 %
1Mb	62.45 %
2Mb	61.78 %
4Mb	60.70 %
8Mb	56.36 %
16Mb	51.10 %
32Mb	44.22 %
64Mb	40.43 %

Όπως αναφέρθηκε και στην Ενότητα 4.3.1, κάθε φορά που κατά την διαδικασία της συγχώνευσης ενός μικρού εύρους εμφανίζεται ένας όρος t με μία μεγάλη λίστα, το εύρος δια-

σπάται σε τρία μικρότερα εύρη, εκ των οποίων το πρώτο περιέχει όλους τους όρους που είναι μικρότεροι από τον t, το δεύτερο μόνο τον t, και το τρίτο όλους τους όρους που είναι μεγαλύτεροι από τον t. Έπειτα, η διαδικασία της συγχώνευσης συνεχίζεται για τις λίστες του τρίτου εύρους, το οποίο μπορεί αναδρομικά να χωριστεί επίσης σε τρία εύρη αν εμφανιστεί κάποιος μεγάλος όρος σε αυτό.

Προφανώς, οι μεγάλοι όροι μπορούν να εμφανιστούν *οπουδήποτε* μέσα σε ένα εύρος. Για παράδειγμα, στην ακραία –αλλά πιθανή– περίπτωση που ο δεύτερος όρος ενός μικρού εύρους γίνει σε κάποια στιγμή μεγάλος όρος, το πρώτο εύρος που θα δημιουργηθεί μετά την διάσπαση θα περιέχει μόνο τον πρώτο όρο, και το αντίστοιχο μπλοκ του του θα περιέχει μόνο την συγκεκριμένη λίστα. Στην περίπτωση αυτή έχουμε ένα μπλοκ με πολύ μεγάλο εσωτερικό κατακερματισμό, εφόσον ένα πολύ μεγάλο τμήμα του παραμένει άδειο, και μάλιστα με μεγάλη πιθανότητα το τμήμα αυτό θα παραμείνει άδειο αφού η λίστα είναι μικρή και ενδέχεται να μην συγκεντρώσει πολλές εμφανίσεις.

Βάσει των παραπάνω είναι προφανές πως όσο πιο συχνά εμφανίζονται μεγάλοι όροι στα μικρά εύρη, τόσο πιο έντονο θα είναι το φαινόμενου του εσωτερικού κατακερματισμού στα μπλοκς των μικρών ευρών, καθώς θα έχουμε διαδοχικές διασπάσεις τους σε πολλά μικρά εύρη. Αυξάνοντας το μέγεθος του μπλοκ αυξάνεται το όριο T_t το οποίο διαχωρίζει της μικρές από τις μεγάλες λίστες, και έτσι μειώνεται ο ρυθμός με τον οποίο παράγονται μεγάλες λίστες και άρα και ο ρυθμός με τον οποίο τα μικρά εύρη διασπώνται λόγω εμφάνισης μεγάλων λιστών. Το τελευταίο έχει σαν αποτέλεσμα ο εσωτερικός κατακερματισμός των μπλοκς που περιέχουν μικρές λίστες τελικά να μειώνεται καθώς το μέγεθος του μπλοκ αυξάνεται.

Εσωτερικός κατακερματισμός στα μεγάλα μπλοκς

Για τα μπλοκς που περιέχουν μεγάλες λίστες ο εσωτερικός κατακερματισμός (F_{long}) είναι αρκετά μικρότερος, της τάξεως του 20%–30%. Σε αυτού του είδους μπλοκς ο κατακερματισμός οφείλεται κυρίως στο τελευταίο μπλοκ κάθε μεγάλης λίστας το οποίο σχεδόν πάντα περιέχει αρκετό ελεύθερο χώρο.

Ο κατακερματισμός στα μπλοκς αυτά επηρεάζεται αντίστροφα με την αύξηση του μεγέθους του μπλοκ, σε σχέση με τα μπλοκ των μικρών λιστών, καθώς όσο αυξάνεται το μέγεθος του μπλοκ προφανώς αυξάνεται και ο ελεύθερος χώρος στο τελευταίο μπλοκ της λίστας. Έτσι, με μεγαλύτερα μεγέθη μπλοκ υπάρχει περισσότερος ελεύθερος χώρος στο τελευταίο μπλοκ κάθε μεγάλης λίστας και άρα αυξάνεται ο εσωτερικός κατακερματισμός των μπλοκ αυτών (Πίνακας 6.4).

Πίνακας 6.4: Επίδραση του μεγέθους του μπλοκ στον εσωτερικό κατακερματισμό των μεγάλων μπλοκς.

B_p	F_{long}
256Kb	12.69 %
512Kb	16.19 %
1Mb	20.57 %
2Mb	25.55 %
4Mb	31.23 %
8Mb	36.17 %
16Mb	39.65 %
32Mb	39.91 %
64Mb	37.28 %

Συνολικός κατακερματισμός ευρετηρίου

Ο συνολικός κατακερματισμός του ανεστραμμένου αρχείου εξαρτάται από το μέγεθος του κατακερματισμού των μικρών και μεγάλων μπλοκς, καθώς και από το πλήθος των μικρών και μεγάλων μπλοκς. Συγκεκριμένα, αν με F_{short} και F_{long} συμβολίσουμε το ποσοστό του ελεύθερου χώρου μέσα σε ένα μικρό και μεγάλο μπλοκ αντίστοιχα, και με N_{short} και N_{long} συμβολίσουμε το πλήθος των μικρών και μεγάλων μπλοκς, τότε ο συνολικός κατακερματισμός F_{total} του ανεστραμμένου αρχείου προκύπτει από τον τύπο:

$$F_{total} = \frac{N_{short} \times F_{short} + N_{long} \times F_{long}}{N_{short} + N_{long}} \tag{6.4}$$

δηλαδή από το λόγο του πλήθους των μπλοκς τα οποία είναι άδεια προς το συνολικό πλήθος των μπλοκς.

Η επίδραση του μεγέθους του μπλοκ B_p στις παραμέτρους N_{short}, N_{long}, F_{short}, F_{long}, F_{total} φαίνεται στον Πίνακα 6.5. Παρατηρούμε πως καθώς αυξάνεται το μέγεθος του μπλοκ, ο συνολικός εσωτερικός κατακερματισμός του ανεστραμμένου αρχείου αυξάνεται μέχρι το $B_p = 8\text{Mb}$, όπου περίπου το 49% του ανεστραμμένου αρχείου είναι άδειο, ενώ από το σημείο εκείνο και μετά ο κατακερματισμός μειώνεται. Συγκεκριμένα, ο εσωτερικός κατακερματισμός τόσο των μικρών όσο και των μεγάλων μπλοκς φαίνεται να συγκλίνει στο 40%, και ανάλογα φαίνεται να συγκλίνει και ο συνολικός κατακερματισμός.

Πίνακας 6.5: Επίδραση του μεγέθους του μπλοκ στον κατακερματισμό του ευρετηρίου.

B_p	N_{short}	N_{long}	F_{short}	F_{long}	F_{total}
256Kb	18698	35490	63.70 %	12.69 %	30.29 %
512Kb	11425	17524	63.25 %	16.19 %	34.76 %
1Mb	6958	8595	62.45 %	20.57 %	39.31 %
2Mb	4260	4158	61.78 %	25.55 %	43.88 %
4Mb	2592	1951	60.70 %	31.23 %	48.05 %
8Mb	1479	839	56.36 %	36.17 %	49.05 %
16Mb	819	314	51.10 %	39.65 %	47.93 %
32Mb	415	106	44.22 %	39.91 %	43.34 %
64Mb	212	34	40.43 %	37.28 %	39.99 %

6.3 Κατώφλι μεγάλων λιστών

Η παράμετρος T_t του συστήματος ονομάζεται "κατώφλι μεγάλων λιστών", καθώς ορίζει το ελάχιστο μέγεθος που πρέπει να έχει μία λίστα ώστε να θεωρηθεί μεγάλη. Ορίζεται σαν ένα ποσοστό επί το μέγεθος του μπλοκ B_p (π.χ. μπορούμε να ορίσουμε πως θα θεωρούμε μεγάλη κάθε λίστα που καταλαμβάνει περισσότερο από 30% του μπλοκ: $T_t = 0.3 \times B_p$). Στην Ενότητα 6.2 μελετήσαμε την επίδραση του μεγέθους του μπλοκ στη δημιουργία του ευρετηρίου. Στις επόμενες ενότητες θα μελετήσουμε την επίδραση της παραμέτρου T_t, δεδομένου ενός συγκεκριμένου και σταθερού μεγέθους μπλοκ, στο συνολικό χρόνο δημιουργίας του ευρετήριο καθώς και στο μέγεθος του ευρετηρίου. Επειδή όπως είπαμε και νωρίτερα η παράμετρος T_t μπορεί να οριστεί σαν το γινόμενο ενός ποσοστού α επί το μέγεθος του μπλοκ B_p ($T_t = \alpha \times B_p$), και θεωρώντας πως το μέγεθος του μπλοκ είναι σταθερό, ουσιαστικά στη συνέχεια θα μελετήσουμε την επίδραση της παραμέτρου α.

6.3.1 Χρόνος δημιουργίας ευρετηρίου

Αύξηση της τιμής της παραμέτρου α συνεπάγεται αυτόματα μείωση του πλήθους των μεγάλων λιστών, αφού λιγότερες λίστες θα ξεπερνάν σε μέγεθος το T_t. Επειδή κάθε μεγάλη λίστα αντιστοιχεί σε ένα μεγάλο εύρος, μείωση των μεγάλων λιστών συνεπάγεται μείωση του πλήθους R_{long} των μεγάλων ευρών που βρίσκονται στον πίνακα ευρών. Άρα, καθώς αυξάνεται η τιμή του α *μειώνεται* και το πλήθος M_{long} των μεγάλων ευρών που συγχωνεύονται κατά την διάρκεια της δημιουργίας του ευρετηρίου, και προφανώς μειώνεται και ο συνολικός χρόνος $T_{merge,long}$ που απαιτείται για την συγχώνευση των μεγάλων ευρών.

Θα περιμέναμε λοιπόν πως, καθώς αυξάνεται η τιμή της παραμέτρου α και άρα και η

τιμή της παραμέτρου T_t, το πλήθος M_{short} των μικρών ευρών θα αυξάνεται καθώς οι μεγάλοι όροι θα παραμένουν στα μικρά μπλοκς αντί να πηγαίνουν σε δικά τους μπλοκς, κάνοντας έτσι τα μικρά μπλοκς να υπερχειλίζουν πιο συχνά και άρα να συμβαίνουν πιο συχνά διασπάσεις (αφού κάθε φορά που υπερχειλίζει ένα μικρό εύρος διασπάται δημιουργώντας ένα παραπάνω μικρό εύρος). Αυτό που παρατηρούμε όμως είναι πως το πλήθος των μικρών ευρών μειώνονται επίσης με την αύξηση του α.

Πίνακας 6.6: Επίδραση της παραμέτρου α στο πλήθος των ευρών που συγχωνεύονται.

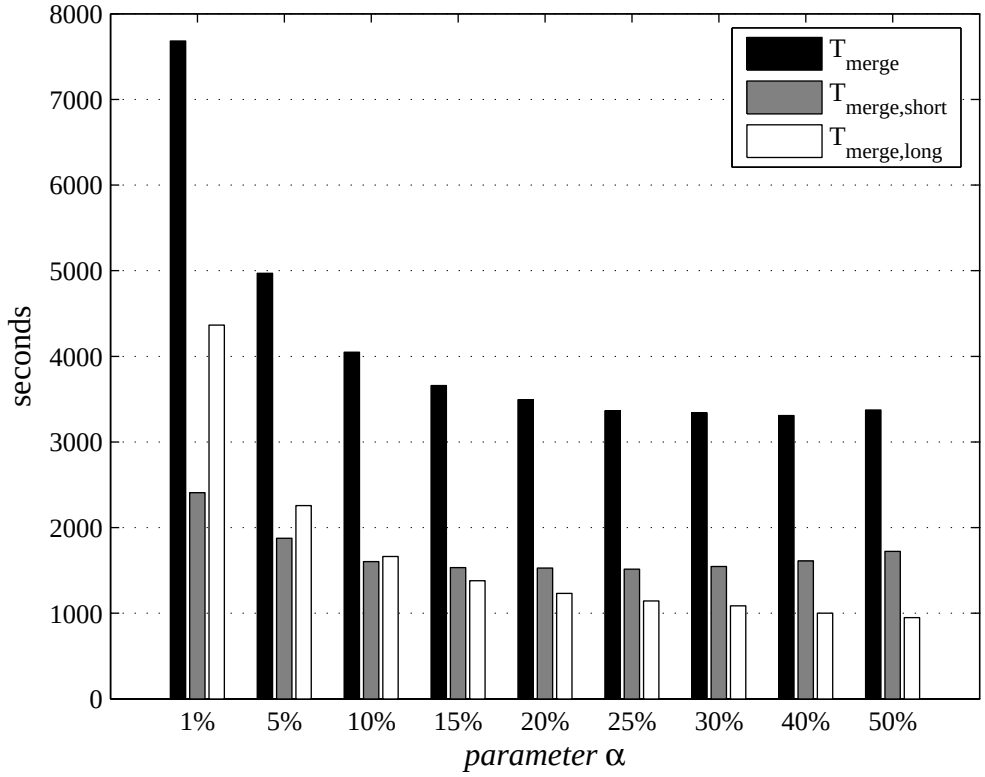
α	M_{short}	M_{long}
1%	23541	221143
5%	14969	92806
10%	12811	62895
15%	12362	51439
20%	12341	45323
25%	12521	41512
30%	12791	38936
40%	13580	35757
50%	14488	33677

Για να καταλάβουμε γιατί παρατηρούμε μείωση του πλήθους των μικρών ευρών με την αύξηση του α αρκεί να παρατηρήσουμε το εξής: όταν το μέγεθος κάποιας λίστας ξεπεράσει το κατώφλι T_t και μετατραπεί σε μεγάλη λίστα, αυτό θα έχει σαν αποτέλεσμα το αρχικό εύρος στο οποίο ανήκε να διασπαστεί σε τρία εύρη, δύο μικρά εύρη και ένα μεγάλο. Έτσι για κάθε μετατροπή μιας λίστας σε μεγάλη λίστα αυξάνονται τόσο τα μεγάλα εύρη όσο και τα μικρά εύρη, κατά 1 αντίστοιχα. Θέτοντας μεγαλύτερες τιμές στην παράμετρο α αυξάνουμε την τιμή του T_t και μειώνουμε τον αριθμό των λιστών που μετατρέπονται σε μεγάλες λίστες, με αποτέλεσμα να μειώνουμε και το πλήθος των διασπάσεων που θα προκαλούσαν αυτές. Αυτό έχει σαν τελικό αποτέλεσμα να μειωθούν και τα μικρά εύρη.

Συνοψίζοντας τα παραπάνω, η αύξηση της τιμής της παραμέτρου α –και συνεπακόλουθα της τιμής του T_t– συνεπάγεται μείωση του πλήθους των μεγάλων ευρών, ενώ υπάρχουν δύο τάσεις που επηρεάζουν την αύξηση ή μείωση του πλήθους των μικρών ευρών: όσο αυξάνεται η τιμή του T_t, από την μία τα μικρά μπλοκς υπερχειλίζουν πιο συχνά και άρα διασπώνται πιο συχνά επειδή οι μεγάλες λίστες καθυστερούν να απομακρυνθούν από τα μικρά μπλοκς (αύξηση του πλήθους των μικρών ευρών), και από την άλλη επειδή απομακρύνονται λιγότερο συχνά μεγάλες λίστες από τα μικρά μπλοκς μειώνονται οι διασπάσεις που προκαλούνται από την μετατροπή μιας μικρής λίστας σε μεγάλη λίστα (μείωση του πλήθους των μικρών

ευρών). Ανάλογα με το ποια από τις δύο τάσεις υπερισχύει, θα έχουμε μείωση ή αύξηση στο πλήθος M_{short} των ευρών που συγχωνεύονται κατά την διάρκεια της δημιουργίας του ευρετηρίου.

Παρατηρώντας τα αποτελέσματα του Πίνακα 6.6, βλέπουμε πως τελικά το M_{short} μειώνεται καθώς το α αυξάνεται από 1% σε 20% (υπερτερεί η μείωση των μικρών ευρών), ενώ από το 20% και μετά το M_{short} αυξάνεται (υπερτερεί η αύξηση των μικρών ευρών).



Σχήμα 6.6: Επίδραση της παραμέτρου α στο χρόνο συγχώνευσης των ευρών.

Όσον αφορά τον χρόνο T_{merge} για την συγχώνευση των ευρών, παρατηρούμε στο Σχήμα 6.6 δύο συμπεριφορές: ο χρόνος $T_{merge,long}$ για την συγχώνευση των μεγάλων ευρών μειώνεται συνεχώς (επειδή όπως είπαμε μειώνεται το πλήθος των μεγάλων ευρών M_{long}), ενώ ο χρόνος $T_{merge,short}$ για την συγχώνευση των μικρών ευρών μειώνεται μέχρι το $\alpha = 25\%$ και μετά αυξάνεται. Παρόλα αυτά, ο βέλτιστος χρόνος συγχώνευσης T_{merge} παρατηρείται για $\alpha = 40\%$ και όχι για $\alpha = 25\%$, και αυτό γιατί παρόλο που από το 25% στο 40% ο χρόνος για την συγχώνευση των μικρών ευρών $T_{merge,short}$ αυξάνεται, την ίδια στιγμή ο χρόνος για την συγχώνευση των μεγάλων ευρών $T_{merge,long}$ μειώνεται με μεγαλύτερο ρυθμό. Συγκεκριμένα, το $T_{merge,short}$ αυξάνεται κατά 100 sec ενώ το $T_{merge,long}$ μειώνεται κατά 140 sec. Άρα, στην μετάβαση από 25% σε 40% η μείωση του χρόνου $T_{merge,long}$ είναι

μεγαλύτερη από την αντίστοιχη αύξηση του χρόνου $T_{merge,short}$. Στο ακριβώς επόμενο βήμα από 40% σε 50%, η μείωση του $T_{merge,long}$ είναι μικρότερη από την αύξηση του $T_{merge,short}$, και έτσι παρατηρείται μία συνολική αύξηση στο T_{merge}.

6.3.2 Μέγεθος ευρετηρίου

Έχοντας μελετήσει στην Ενότητα 6.2.4 την επίδραση του μεγέθους του μπλοκ στον εσωτερικό κατακερματισμό των μπλοκς, θα μελετήσουμε τώρα την επίδραση της παραμέτρου α στον εσωτερικό κατακερματισμό των μπλοκς, και συνεπακόλουθα, την επίδραση της στο μέγεθος του ανεστραμμένου αρχείου.

Η παράμετρος α ορίζει μαζί με την παράμετρο B_p την τιμή του T_t, το πότε δηλαδή μία λίστα θεωρείται μεγάλη. Η μετάβαση της ανεστραμμένης λίστας ενός όρου t από την κατηγορία των μικρών λιστών στην κατηγορία των μεγάλων λιστών συνεπάγεται αυτόματα πως το αρχικό εύρος στο οποίο ανήκε ο t θα διασπαστεί σε τρία εύρη, ενώ παράλληλα, οι λίστες του εύρους που μέχρι στιγμής ήταν αποθηκευμένες σε ένα μπλοκ θα αποθηκευτούν σε τρία μπλοκς, κατ' αντιστοιχία με τα εύρη στα οποία ανήκουν πλέον.

Το τελευταίο σημαίνει πως αν το συνολικό μέγεθος των λιστών του αρχικού εύρους είναι S, και άρα έχουμε έναν εσωτερικό κατακερματισμό $\frac{B_p-S}{B_p}$ για το μπλοκ στο οποίο βρίσκονται αποθηκευμένες, μετά την διάσπαση του εύρους και την αποθήκευση των λιστών σε τρία μπλοκς ο συνολικός κατακερματισμός για τα μπλοκς αυτά αυτόματα αυξάνεται σε $\frac{3 \cdot B_p-S}{3 \cdot B_p}$, αφού έχουμε το ίδιο μέγεθος λιστών, αλλά πλέον βρίσκονται αποθηκευμένες σε τρία μπλοκς.

Γίνεται έτσι προφανές πως η παράμετρος α επηρεάζει άμεσα τον κατακερματισμό των μπλοκς, εφόσον έμμεσα (μέσω του T_t) ελέγχει τον ρυθμό με τον οποίον οι λίστες μετατρέπονται σε μεγάλες λίστες. Θέτοντας μία μικρή τιμή στην παράμετρο αυτή, για παράδειγμα 1%, αυτόματα ορίζουμε πως κάθε λίστα της οποίας το μέγεθος ξεπερνάει το 1% του μεγέθους του μπλοκ θεωρείται μεγάλη λίστα και αποθηκεύεται σε ένα ξεχωριστό μπλοκ, δημιουργώντας έτσι μεγάλο εσωτερικό κατακερματισμό στα μεγάλα μπλοκς. Επίσης, έχουμε πολύ συχνά διασπάσεις των μικρών ευρών με αποτέλεσμα να δημιουργούνται εύρη τα οποία περιέχουν ελάχιστους όρους, καθένα από τα οποία εύρη δεσμεύει επίσης ένα μπλοκ, προκαλώντας έτσι μεγάλο εσωτερικό κατακερματισμό και στα μικρά μπλοκς.

Σύμφωνα με τα παραπάνω, αύξηση της παραμέτρου α συνεπάγεται αυτόματα μείωση του κατακερματισμού και στα δύο είδη μπλοκς, και άρα και μείωση του συνολικού κατακερματισμού. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 6.7 (υπενθυμίζουμε πως με N_{short}

Πίνακας 6.7: Επίδραση της παραμέτρου α στον κατακερματισμό του ευρετηρίου.

α	N_{short}	N_{long}	F_{short}	F_{long}	F_{total}
1%	32118	38737	96.78 %	78.29 %	86.67 %
5%	12025	17150	86.99 %	54.03 %	67.61 %
10%	8598	12747	77.94 %	40.75 %	55.73 %
15%	7577	10834	71.82 %	32.53 %	48.70 %
20%	7157	9796	67.59 %	27.24 %	44.27 %
25%	6982	9046	64.36 %	23.07 %	41.05 %
30%	6958	8595	62.45 %	20.57 %	39.31 %
40%	7130	7942	60.18 %	16.89 %	37.37 %
50%	7352	7490	58.66 %	14.51 %	36.38 %

και N_{long} συμβολίζουμε το πλήθος των μικρών και μεγάλων μπλοκς, με F_{short} και F_{long} τον κατακερματισμό –το ποσοστό του ελεύθερου χώρου– στα μικρά και μεγάλα μπλοκς, ενώ ο συνολικός κατακερματισμός του ευρετηρίου F_{total} προκύπτει από την εξίσωση 6.4).

6.4 Αναλογία κόστους συγχώνευσης

Η παράμετρος K_t εκτιμά την σχέση που υπάρχει μεταξύ του κόστους της συγχώνευσης ενός μικρού εύρους και ενός μεγάλου εύρους. Συγκεκριμένα, μεταξύ ενός μικρού εύρους R_s και ενός μεγάλου εύρους R_l επιλέγουμε να συγχωνεύσουμε το R_s μόνο αν ισχύει:

$$\frac{R_s.mem\_postings}{R_l.mem\_postings} \geq K_t$$

ή ισοδύναμα, μόνο αν:

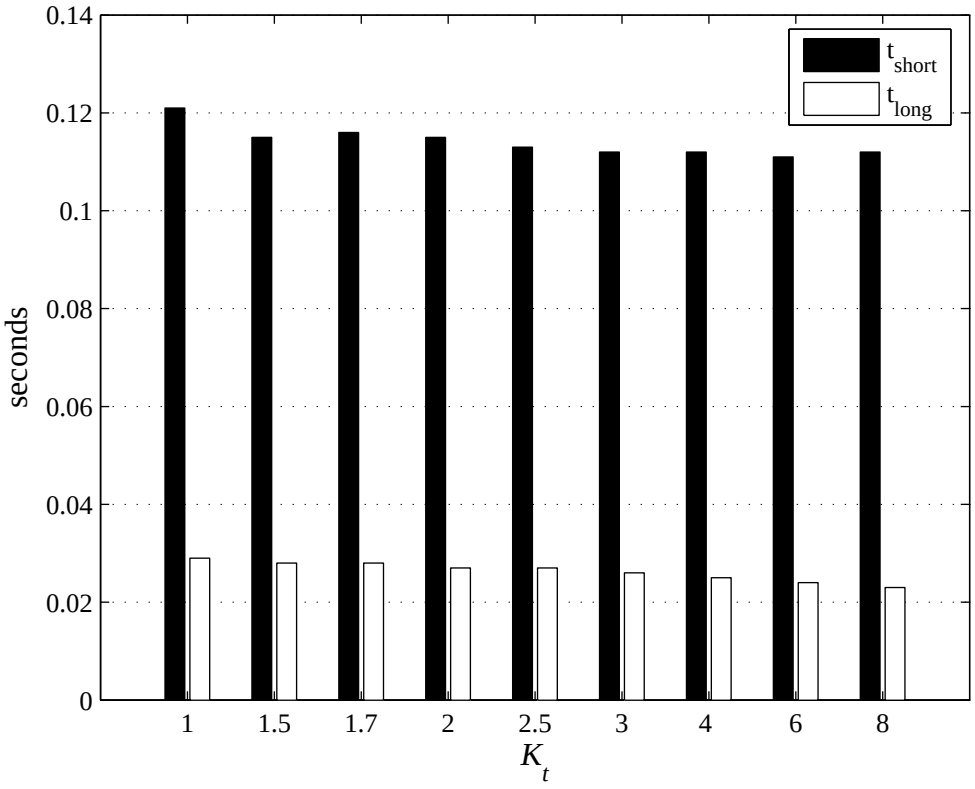
$$R_s.mem\_postings \geq K_t \times R_l.mem\_postings$$

Αυτό πρακτικά σημαίνει πως όσο μεγαλύτερο είναι το K_t, τόσες περισσότερες εμφανίσεις θα πρέπει να έχει ένα μικρό εύρος για να το προτιμήσουμε έναντι ενός μεγάλου εύρους, ή αλλιώς, θεωρούμε πιο συμφέρον να συγχωνεύσουμε ένα μεγάλο εύρος από ένα μικρό εύρος. Η επίδραση αυτή του K_t απεικονίζεται στο Σχήμα 6.8, όπου βλέπουμε πως καθώς αυξάνουμε το K_t από 1 σε 8, το πλήθος M_{short} των μικρών ευρών που συγχωνεύουμε μειώνεται ενώ αντίστοιχα το πλήθος M_{long} των μεγάλων ευρών αυξάνεται.

Σύμφωνα με την εξίσωση 6.3, ο συνολικός χρόνος συγχώνευσης T_{merge} ισούται με:

$$T_{merge} = T_{merge,prepare} + (M_{short} \times t_{short}) + (M_{long} \times t_{long})$$

όπου ο χρόνος $T_{merge,prepare}$ είναι γενικά σταθερός για δεδομένο μέγεθος μπλοκ.

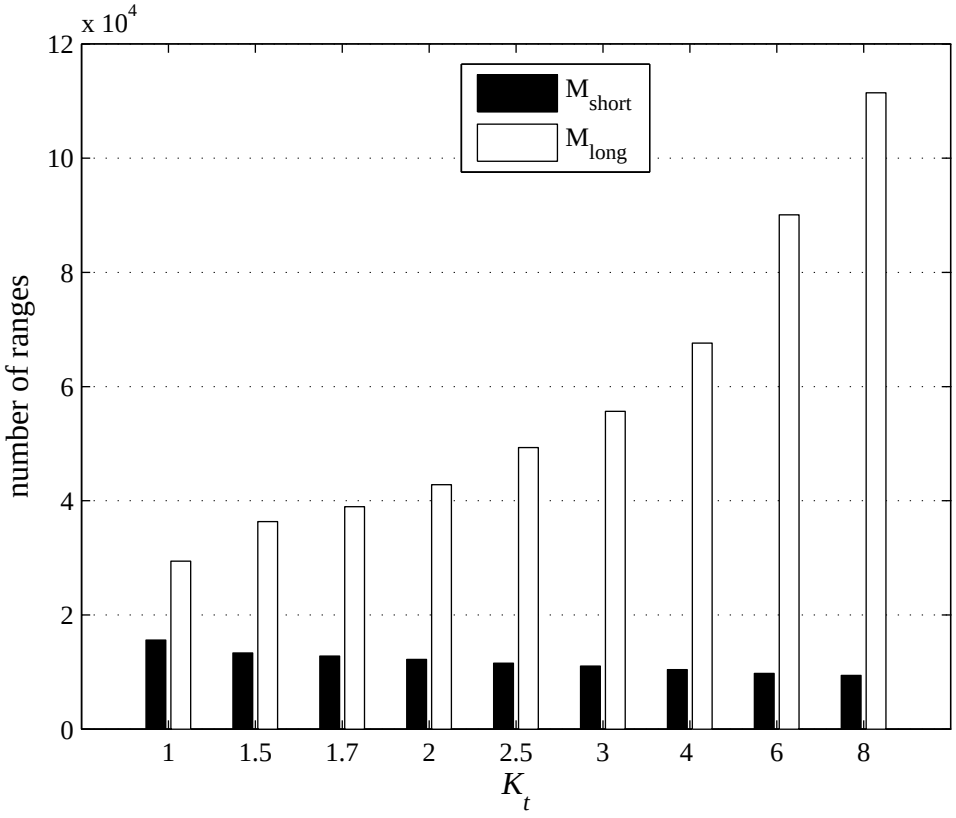


Σχήμα 6.7: Επίδραση της παραμέτρου K_t στο μέσο χρόνο συγχώνευσης των ευρών.

Όπως προκύπτει από το Σχήμα 6.7, το t_{short} παραμένει σχετικά αμετάβλητο για $K_t \geq 2$ ενώ το t_{long} μειώνεται ελάχιστα. Το γεγονός αυτό αρχικά επιβεβαιώνει την υπόθεση που κάναμε στον αλγόριθμο Πρωτέας πως ο χρόνος για να αποθηκεύσουμε ένα εύρος στο δίσκο είναι γενικά ανεξάρτητος του μεγέθους των εμφανίσεων που έχει συλλέξει στη μνήμη. Για παράδειγμα, αυξάνοντας την τιμή του K_t αναβάλουμε την συγχώνευση των μικρών ευρών, με αποτέλεσμα αυτά να συλλέγουν περισσότερες εμφανίσεις στη μνήμη, και άρα όταν θα αποθηκευτούν στο δίσκο θα έχουν περισσότερες εμφανίσεις.

Θα περιμέναμε λοιπόν πως εφόσον η συγχώνευση των μεγάλων ευρών είναι πολύ πιο αποδοτική από αυτή των μικρών (το t_{long} είναι συνήθως 3 με 4 φορές μικρότερο από το t_{short}) και εφόσον μειώνεται το πλήθος M_{short} των μικρών ευρών και αυξάνεται το πλήθος M_{long} των μεγάλων ευρών, πως θα υπάρχει και μία συνεχής μείωση στο συνολικό χρόνο για την συγχώνευση. Κάτι τέτοιο θα συνέβαινε μόνο αν ο ρυθμός μείωσης του M_{short} ήταν παρόμοιος ή μεγαλύτερος από τον ρυθμό αύξησης του M_{long}. Για παράδειγμα, αν το M_{short} μειωνόταν κατά N και το M_{long} αυξανόταν κατά N (ή λιγότερο), τότε ο συνολικός χρόνος θα μειωνόταν κατά $N \cdot t_{short}$ ενώ θα αυξανόταν κατά $N \cdot t_{long}$ (όπου προφανώς

$N \cdot t_{short} \geq N \cdot t_{long}$), και άρα τελικά ο συνολικός χρόνος θα μειωνόταν.

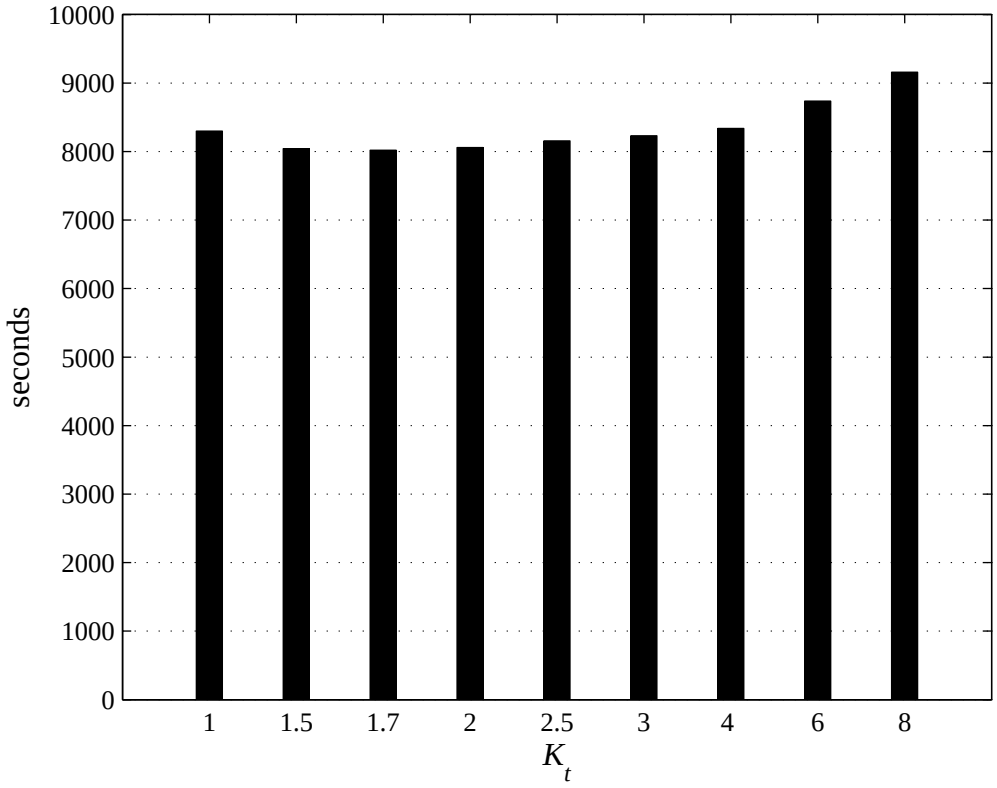


Σχήμα 6.8: Επίδραση της παραμέτρου K_t στο πλήθος των ευρών που συγχωνεύονται.

Αυτό που παρατηρούμε όμως είναι πως ο ρυθμός αύξησης των μεγάλων ευρών είναι πολύ μεγαλύτερος από την αντίστοιχη μείωση των μικρών ευρών. Για παράδειγμα, αυξάνοντας το K_t από 1 σε 3 το M_{short} μειώνεται κατά 30% ενώ την ίδια στιγμή το M_{long} αυξάνεται κατά 90%. Αυξάνοντας το K_t από 3 σε 5 έχουμε μία περαιτέρω μείωση του M_{short} κατά 15%, την ίδια στιγμή που το M_{long} αυξάνεται κατά 60%.

Φτάνουμε έτσι σε ένα σημείο όπου το πλήθος των μεγάλων ευρών είναι πολύ μεγαλύτερο από το πλήθος των μικρών ευρών. Για παράδειγμα, για $K_t = 4$ το M_{long} είναι περίπου 70,000 ενώ το M_{short} είναι 10,000: τα μεγάλα εύρη είναι σε πλήθος 7 φορές περισσότερα από τα μικρά, την ίδια στιγμή που ο χρόνος που απαιτούν για την συγχώνευση είναι μόλις 4 φορές μικρότερος από τον αντίστοιχο των μικρών ευρών. Αυτός είναι και ο λόγος για τον οποίο παρατηρούμε αύξηση του χρόνου συγχώνευσης T_{merge} για τιμές του K_t μεγαλύτερες από 1.7, λόγω ακριβώς της μεγάλης αύξησης του αριθμού των μεγάλων ευρών.

Η επίδραση της παραμέτρου K_t στο συνολικό χρόνο δημιουργίας του ευρετηρίου παρουσιάζεται στο Σχήμα 6.9. Η περιοχή τιμών από 1.5 μέχρι 2 φαίνεται να είναι μία ασφαλής επιλογή για το K_t (με βέλτιστη τιμή το 1.7).



Σχήμα 6.9: Επίδραση της παραμέτρου K_t στο χρόνο δημιουργίας του ευρετηρίου.

6.5 Μνήμη

Καθώς τα αρχεία της συλλογής αναλύονται λεκτικά, οι εμφανίσεις των όρων συγκεντρώνονται στη μνήμη. Για την συλλογή των εμφανίσεων επιτρέπουμε στο σύστημα να χρησιμοποιήσει ένα τμήμα μνήμης μεγέθους P_t. Κάθε φορά που η προσωρινή αυτή μνήμη εξαντλείται, αποθηκεύουμε ένα πλήθος εμφανίσεων από τη μνήμη στο δίσκο, συνολικού μεγέθους P_f.

Για κάθε όρο για τον οποίο υπάρχουν κάποιες εμφανίσεις στη μνήμη διατηρούμε επίσης μία βοηθητική δομή, τη δομή εμφανίσεων, η οποία περιέχει όλες τις πληροφορίες που είναι απαραίτητες για την κωδικοποίηση των εμφανίσεων του. Το σύνολο των δομών εμφανίσεων *δεν προσμετρώνται* στην προσωρινή μνήμη μεγέθους P_t.

Στο εξής θα αναφερόμαστε με τον όρο *μνήμη* του συστήματος στην προσωρινή μνήμη που είναι διαθέσιμη για την συγκέντρωση των εμφανίσεων (χωρίς τους περιγραφείς τους). Θεωρούμε πως η υπόλοιπη μνήμη του συστήματος είναι αρκετή για τους περιγραφείς των όρων (το οποίο μπορεί να μη συμβαίνει αν έχουμε πολύ μικρά μπλοκ και ταυτόχρονα μεγάλα τιμές του P_t, όπως αναφέρουμε παρακάτω).

6.5.1 Μέγεθος προσωρινής μνήμης

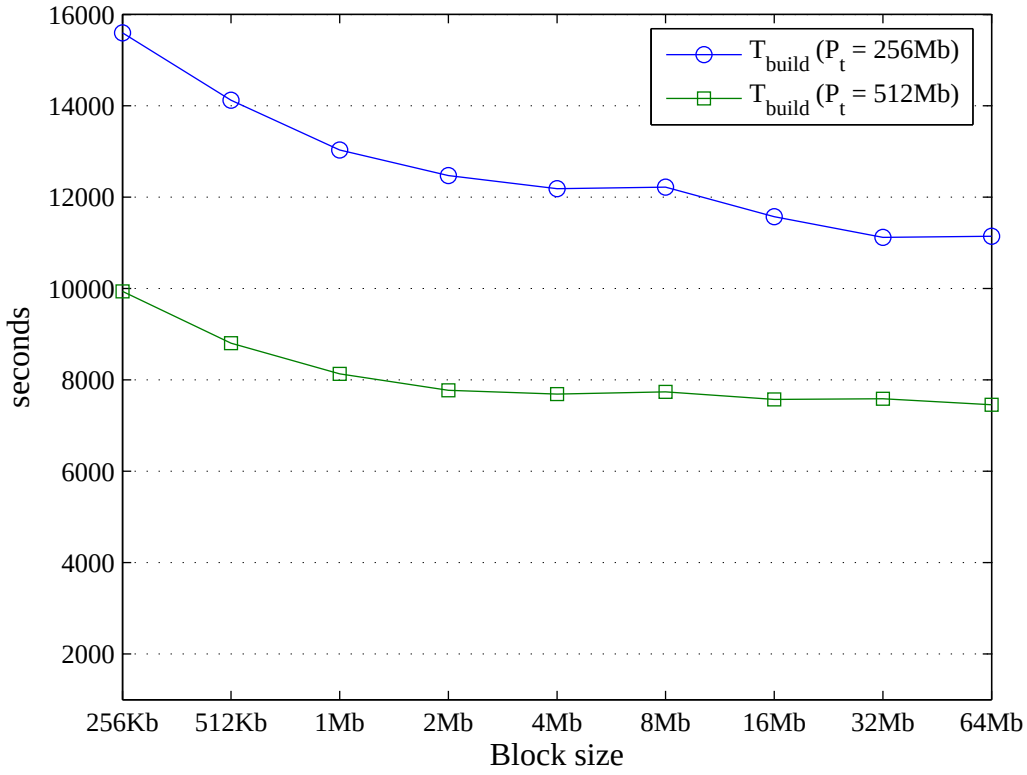
Η κατανομή του μεγέθους των λιστών φαίνεται να ακολουθεί κατανομή Zipf [3]. Αυτό σημαίνει πως υπάρχει ένας μικρός αριθμός από όρους που είναι πολύ δημοφιλείς και έχουν πολύ μεγάλες λίστες, και ένας μεγάλος αριθμός από όρους που έχουν πολύ μικρές λίστες. Αυτό έχει σαν αποτέλεσμα το μεγαλύτερο μέρος του ανεστραμμένου αρχείου να καταλαμβάνεται από λίγες αλλά μεγάλες λίστες, ενώ όλες οι υπόλοιπες λίστες καταλαμβάνουν ένα μικρό μέρος του αρχείου.

Μία παρόμοια συμπεριφορά παρατηρείται και στη μνήμη κατά την διάρκεια της λεκτικής ανάλυσης των αρχείων: υπάρχει ένας μεγάλος αριθμός από λίστες που έχουν συγκεντρώσει ελάχιστες εμφανίσεις, και ένας μικρός αριθμός δημοφιλών λιστών που έχουν μαζέψει αρκετές εμφανίσεις. Βάσει του αλγορίθμου Πρωτέας, κάθε φορά που εξαντλείται η μνήμη επιλέγουμε να συγχωνεύσουμε ένα σύνολο από εύρη τα οποία περιέχουν συνολικά P_f εμφανίσεις. Προφανώς, επιλέγουμε να συγχωνεύσουμε τα εύρη που έχουν συγκεντρώσει τις περισσότερες εμφανίσεις, ώστε η συγχώνευση τους να είναι αποτελεσματική.

Ο τρόπος λειτουργίας του αλγορίθμου είναι τέτοιος που επιλέγει σε κάθε βήμα να συγχωνεύσει μόνο εκείνα τα εύρη τα οποία έχουν συγκεντρώσει αρκετές εμφανίσεις στη μνήμη. Αυτό φυσικά έχει σαν αποτέλεσμα να υπάρχει στη μνήμη ένα πλήθος από εύρη που περιέχουν σπάνιους όρους και έχουν συγκεντρώσει ελάχιστες εμφανίσεις, και τα οποία γι' αυτόν ακριβώς τον λόγο πιθανόν να μην επιλεγούν ποτέ να συγχωνευθούν.

Έτσι, ένα τμήμα της μνήμης είναι μονίμως δεσμευμένο από αυτά τα εύρη, των οποίων το συνολικό μέγεθος αυξάνεται, εφόσον με κάποιο σταθερό (αλλά χαμηλό) ρυθμό εμφανίζονται συνεχώς νέοι σπάνιοι όροι. Φυσικά, όσο μικρότερο είναι το μέγεθος P_t της προσωρινής μνήμης, τόσο συχνότερα θα γεμίζει η μνήμη και τόσο συχνότερα θα εκτελείται η διαδικασία συγχώνευσης, με αποτέλεσμα κάθε εύρος να προλαβαίνει να μαζέψει λιγότερες εμφανίσεις μεταξύ δύο διαδοχικών διαδικασιών συγχώνευσης και οι συγχωνεύσεις των ευρών να είναι έτσι λιγότερο αποτελεσματικές.

Περιμένουμε λοιπόν πως μεγαλύτερα μεγέθη προσωρινής μνήμης θα βελτιώνουν την απόδοση του αλγορίθμου, καθώς θα υπάρχει περισσότερη μνήμη διαθέσιμη για την συγκέντρωση των νέων εμφανίσεων. Επειδή το ποσοστό της μνήμης που δεσμεύεται από τα εύρη με τους σπάνιους όρους εξαρτάται και από το μέγεθος του μπλοκ (μεγαλύτερα μεγέθη μπλοκ επιτρέπουν σε κάθε εύρος να περιέχει περισσότερους όρους και άρα να συγκεντρώνει περισσότερες εμφανίσεις), εκτελέσαμε μία σειρά πειραμάτων για μεγέθη προσωρινής μνήμης 256Mb και 512Mb όπου το μέγεθος του μπλοκ B_p κυμαίνονταν από 256Kb μέχρι 64Mb.



Σχήμα 6.10: Επίδραση του μεγέθους της προσωρινής μνήμης στη δημιουργία του ευρετηρίου.

Από το Σχήμα 6.10 είναι προφανές πως μεγαλύτερα μεγέθη προσωρινής μνήμης βελτιώνουν σημαντικά τους χρόνους δημιουργίας του ευρετηρίου, και άρα μία επέκταση της φυσικής μνήμης του συστήματος συνεπάγεται αυτόματα βελτίωση της απόδοσης του συστήματος.

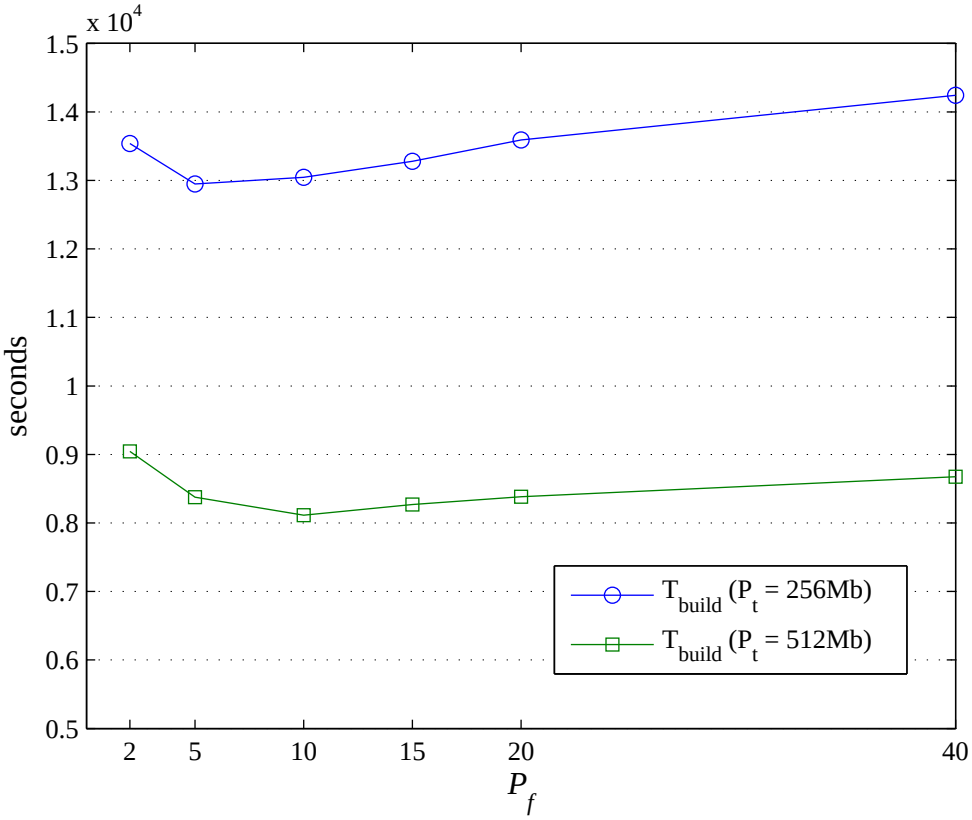
Από τα αναλυτικά αποτελέσματα που παρουσιάζονται στο Παράρτημα παρατηρούμε πως για $B_p = 256$ Kb και $P_t = 512$ Mb έχουμε αυξημένο χρόνο για την *λεκτική ανάλυση* των κειμένων. Αυτό οφείλεται στο γεγονός πως για μικρά μεγέθη μπλοκ B_p και μεγάλα μεγέθη προσωρινής μνήμης P_t, η υπόλοιπη μνήμη που παραμένει διαθέσιμη στο σύστημα φαίνεται να γεμίζει με τις δομές εμφανίσεων των όρων, προκαλώντας όπως αναφέραμε και νωρίτερα φαινόμενα χρήσης χώρου εναλλαγής (swap space), τα οποία με την σειρά τους επιφέρουν μεγάλες καθυστερήσεις.

6.5.2 Μέγεθος μνήμης συγχώνευσης

Όταν η προσωρινή μνήμη εξαντληθεί, επιλέγουμε κάποια εύρη των οποίων οι εμφανίσεις στη μνήμη καταλαμβάνουν συνολικά χώρο P_f και τα συγχωνεύουμε με τον δίσκο. Έτσι,

η διαδικασία επιλογής και συγχώνευσης των ευρών λαμβάνει χώρα κάθε P_f εμφανίσεις που συγκεντρώνονται στη μνήμη.

Αν θέσουμε μία μικρή τιμή στην παράμετρο P_f αυτό θα σημαίνει πως η διαδικασία της επιλογής και συγχώνευσης των ευρών θα συμβαίνει πολύ συχνά, με αποτέλεσμα τα εύρη να μην προλαβαίνουν να συγκεντρώσουν αρκετές εμφανίσεις και οι συγχωνεύσεις τους να είναι λιγότερο αποτελεσματικές. Από την άλλη, μία μεγάλη τιμή επιτρέπει να επιλέγονται προς συγχώνευση και εύρη με σχετικά λίγες εμφανίσεις, προκειμένου να συγκεντρωθούν συνολικά P_f εμφανίσεις, το οποίο προφανώς δημιουργεί προβλήματα καθώς γίνονται διάφορες διαδικασίες ανάγνωσης και εγγραφής στο δίσκο για ελάχιστα δεδομένα.



Σχήμα 6.11: Επίδραση του μεγέθους της μνήμης συγχώνευσης στη δημιουργία του ευρετηρίου.

Από τα πειραματικά αποτελέσματα (Σχήμα 6.11), δοκιμάζοντας τιμές μεταξύ 2Mb και 40Mb για την μνήμη συγχώνευσης, καταλήγουμε στο συμπέρασμα πως όταν η μνήμη συγκέντρωσης είναι 512Mb η βέλτιστη τιμή για τη μνήμη συγχώνευσης φαίνεται να είναι τα 10Mb, ενώ μειώνοντας την μνήμη συγκέντρωσης στο μισό (P_t=256Mb) παρατηρούμε πως η βέλτιστη τιμή για τη μνήμη συγχώνευσης μειώνεται ανάλογα (P_f=5Mb). Άρα, μία καλή τιμή για την μνήμη συγχώνευσης P_f είναι το 2% του μεγέθους της προσωρινής μνήμης P_t.

6.6 Χρόνος ανάκτησης λιστών

Ανεξάρτητα από τον αλγόριθμο που χρησιμοποιείται για την αποτίμηση ενός ερωτήματος αναζήτησης, ένα μεγάλο μέρος του χρόνου δαπανάται για την ανάκτηση από τον δίσκο των ανεστραμμένων λιστών των όρων του ερωτήματος. Ο χρόνος για την ανάκτηση μιας λίστας εξαρτάται κυρίως από το μέγεθος της λίστας και από το πλήθος των τμημάτων από τα οποία αποτελείται. Για παράδειγμα, μια μικρή λίστα λίγων Kb η οποία είναι αποθηκευμένη σε πολλά τμήματα στο δίσκο θα απαιτήσει πάρα πολύ χρόνο για την ανάκτηση της λόγω της μετακίνησης της κεφαλής του δίσκου μεταξύ των τμημάτων της.

Στο σύστημά μας όλες οι μικρές λίστες αποθηκεύονται συνεχόμενες στο δίσκο, ενώ οι μεγάλες λίστες αποθηκεύονται σε έναν αριθμό από μπλοκς τα οποία στην γενική περίπτωση είναι μη συνεχόμενα. Το πλήθος των μπλοκς από τα οποία αποτελείται μία μεγάλη λίστα εξαρτάται αποκλειστικά από το μέγεθος του μπλοκ B_p: μεγαλύτερα μεγέθη μπλοκ οδηγούν προφανώς σε λιγότερα μπλοκς ανά λίστα. Παράλληλα, η αύξηση του B_p μειώνει το πλήθος των μεγάλων λιστών (εφόσον αυξάνεται το κατώφλι T_t), και άρα ένας αριθμός από λίστες που πριν θεωρούνταν μεγάλες και αποθηκεύονταν σε ένα σύνολο από μπλοκς πλέον αποθηκεύονται συνεχόμενες στο δίσκο.

Για να μετρήσουμε την αποτελεσματικότητα της ανάκτησης των λιστών όπως αυτές αποθηκεύονται στο σύστημα μας, κάναμε κάποιες μετρήσεις σχετικά με τον συνολικό χρόνο που απαιτείται για να ανακτήσουμε τις λίστες όλων των όρων (συνολικά 27,934) των 10,000 πρώτων ερωτημάτων από το *Efficiency Topic* του TREC Terabyte Track 2005. Για κάθε όρο βρίσκουμε με την βοήθεια του λεξικού σε ποια μπλοκς βρίσκονται αποθηκευμένες οι εμφανίσεις του και έπειτα διαβάζουμε σειριακά τα μπλοκς αυτά. Τέλος, συγκρίνουμε τον χρόνο αυτό ('fragmented' - `frg`) με την περίπτωση όπου όλες λίστες είναι αποθηκευμένες συνεχόμενες στο δίσκο ('contiguous' - `cnt`), την οποία θεωρούμε ως ιδανική περίπτωση. Οι λίστες των όρων ανακτώνται με την σειρά με την οποία οι όροι εμφανίζονται στα ερωτήματα (για παράδειγμα, δεν γίνεται κανενός είδους ταξινόμηση των όρων βάσει της διάταξης τους στο δίσκο ή βάσει του μεγέθους τους).

Θεωρούμε μεγέθη μπλοκ από 512Kb μέχρι 16Mb. Για κάθε μέγεθος μπλοκ μετράμε πόσες από τις 27,934 λίστες που ανακτούμε είναι μικρές και πόσες μεγάλες[5], καθώς και πόσα μπλοκς συνολικά ανακτούμε για τις μεγαλές λίστες. Επίσης, μετράμε τον συνολικό χρόνο ανάκτησης όλων των λιστών στις δύο περιπτώσεις (T_{cnt} και T_{frg}), καθώς και τους επιμέρους χρόνους ανάκτησης όλων των μικρών λιστών ($T_{cnt,short}$ και $T_{frg,short}$) και όλων

[5]το μέγεθος του μπλοκ B_p επηρεάζει το κατώφλι T_t που ορίζει πότε μία λίστα θεωρείτει μεγάλη

των μεγάλων λιστών ($T_{cnt,long}$ και $T_{frg,long}$). Η επίδραση αυτή του μεγέθους του μπλοκ B_p στις παραπάνω παραμέτρους παρουσιάζεται στους Πίνακες 6.8 και 6.9.

Πίνακας 6.8: Επίδραση μεγέθους του μπλοκ στο πλήθος των μικρών και μεγάλων λιστών που ανακτούμε, καθώς και στο πλήθος των μπλοκς που ανακτούμε.

B_p	short lists	long lists	long blocks	blocks/long list
512Kb	11492	16442	580181	35.2
1Mb	13577	14357	292038	20.3
2Mb	16096	11838	147869	12.5
4Mb	18565	9369	75276	8.0
8Mb	21128	6806	38344	5.6
16Mb	23356	4578	19027	4.2

Πίνακας 6.9: Επίδραση του μεγέθους του μπλοκ στο χρόνο ανάκτησης μικρών και μεγάλων λιστών.

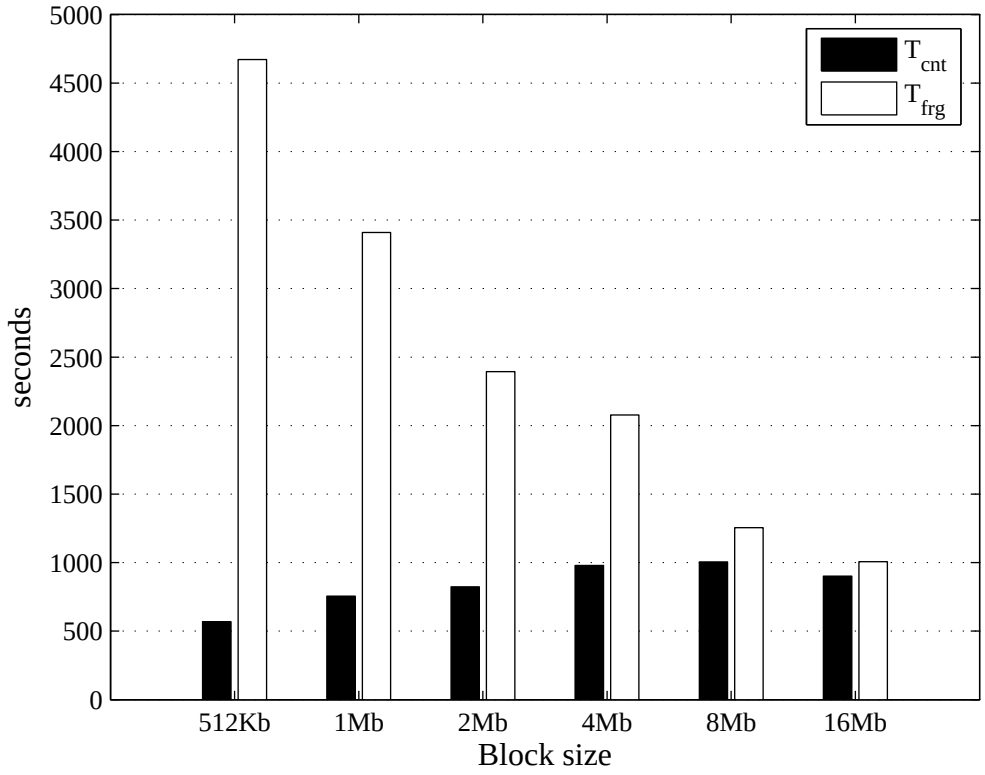
B_p	T_{cnt}	T_{frg}	incr.	$T_{cnt,short}$	$T_{frg,short}$	incr.	$T_{cnt,long}$	$T_{frg,long}$	incr.
512Kb	568	4671	721%	84	135	61%	484	4535	835%
1Mb	756	3410	351%	124	176	41%	631	3234	412%
2Mb	823	2394	190%	160	213	32%	662	2181	229%
4Mb	979	2078	112%	235	297	26%	743	1781	139%
8Mb	1005	1255	24%	453	530	17%	551	724	31%
16Mb	902	1006	11%	410	454	10%	492	552	12%

Από τις μετρήσεις προκύπτει καταρχήν πως όσο μεγαλύτερο είναι το μέγεθος του μπλοκ, τόσο μικρότερη επιβάρυνση έχουμε κατά την ανάκτηση των λιστών. Για πολύ μικρά μεγέθη μπλοκ οι λίστες αποτελούνται από πάρα πολλά μπλοκς με αποτέλεσμα η ανάκτηση μίας λίστας να απαιτεί πολλές μετακινήσεις της κεφαλής του δίσκου. Με ικανοποιητικά μεγάλο μέγεθος μπλοκ (για παράδειγμα, 16Mb) ο χρόνος ανάκτησης του συστήματος μας είναι μόλις 10% υψηλότερος από την *ιδανική* περίπτωση όπου οι λίστες είναι αποθηκευμένες συνεχόμενες. Σε αυτό φυσικά συμβάλει το γεγονός πως με μεγαλύτερα μεγέθη μπλοκ οι μεγάλες λίστες αποτελούνται από *ελάχιστα μπλοκς* (για B_p=16Mb κάθε μεγάλη λίστα κατά μέσο όρο αποτελείται μόνο από 4.2 μπλοκς), ενώ ταυτόχρονα μειώνεται το *πλήθος των μεγάλων λιστών*, οι οποίες ουσιαστικά δημιουργούν τις διάφορες επιβαρύνσεις λόγω του ότι βρίσκονται κατακερματισμένες στο δίσκο (βλ. Πίνακα 6.8).

Η αύξηση που παρατηρείται στο χρόνο ανάκτησης των μικρών λιστών $T_{frg,short}$ καθώς αυξάνεται το μεγέθος του μπλοκ οφείλεται απλά στο γεγονός πως ανακτούνται *περισσότερες* μικρές λίστες, όπως παρατηρούμε και στον Πίνακα 6.8, και όχι στο ότι επιβαρύνεται η ανάκτηση τους: αυξάνοντας το B_p, ένα σύνολο από λίστες που θεωρούνταν μεγάλες πλέον θεωρούνται μικρές, και άρα ο χρόνος τους συνυπολογίζεται στο χρόνο ανάκτησης των μικρών λιστών.

Αρκετά ενδιαφέρον είναι το γεγονός πως οι χρόνοι $T_{frg,short}$ και $T_{cnt,short}$ έχουν μία σημαντική διαφορά. Υπενθυμίζουμε πως και στην περίπτωση του συστήματος μας οι μικρές λίστες αποθηκεύονται πάντα συνεχόμενες στο δίσκο, και άρα δε θα περιμέναμε κάποια σημαντική διαφορά μεταξύ των δύο παραπάνω χρόνων. Εικάζουμε πως η συμπεριφορά αυτή οφείλεται στην μετακίνηση της κεφαλής του δίσκου μεταξύ διαδοχικών λιστών. Επίσης παρατηρούμε πως με την αύξηση του μεγέθους του μπλοκ αυξάνεται και ο χρόνος ανάκτησης των λιστών στην περίπτωση που αυτές είναι συνεχόμενα αποθηκευμένες στο δίσκο (T_{cnt}). Εικάζουμε πως και αυτή η συμπεριφορά οφείλεται στη μετακίνηση της κεφαλής του δίσκου από το τέλος της μίας λίστας στην αρχή της επόμενης και όχι στην ανάκτηση των λιστών, εφόσον το μόνο πράγμα που αλλάζει όσον αφορά τις λίστες αυτές καθώς αυξάνεται το μέγεθος μπλοκ είναι η *θέση* των λιστών στο δίσκο (συμβαίνουν διαφορετικές διασπάσεις και οι λίστες αποθηκεύονται σε διαφορετικά μπλοκς).

Εφόσον οι μετρήσεις μας αφορούν την απλή σειριακή ανάκτηση των λιστών, χωρίς κάποιου είδους βελτιστοποιήσεις στη σειρά με την οποία ανακτώνται οι λίστες και τα μπλοκς, θεωρούμε την επιβάρυνση του 10% σε σχέση με την ιδανική περίπτωση όπου όλες οι λίστες είναι αποθηκευμένες συνεχόμενες στο δίσκο αρκετά μικρή. Να τονίσουμε πως στις περιπτώσεις που χρησιμοποιούνται αλγόριθμοι βάσει συγχώνευσης, όπως ο αλγόριθμος γεωμετρικής διαμέρισης ή ο αλγόριθμος λογαριθμικής συγχώνευσης, οι μεγάλες λίστες είναι επίσης αποθηκευμένες σε διάφορα τμήματα στο δίσκο, ενώ επιπλέον σε αυτούς τους αλγορίθμους είναι πιθανό και οι *μικρές λίστες* να είναι αποθηκευμένες σε πολλά τμήματα στον δίσκο. Επιπλέον, για να είναι οι αλγόριθμοι αυτοί σε θέση να διατηρούν τις μεγάλες λίστες κατακερματισμένες σε ένα τόσο μικρό αριθμό τμημάτων όπως στο σύστημα μας (για μέγεθος μπλοκ 16Mb κάθε λίστα αποτελείται μόνο από 4.2 μπλοκς) θα πρέπει να εκτελούν πολύ συχνά συγχωνεύσεις των διαφόρων τμημάτων από τα οποία αποτελείται το ευρετήριο, ώστε να διατηρούν μόνο 4-5 τμήματα κάθε φορά ταυτόχρονα στο σύστημα. Στην περίπτωση αυτή όμως οι *χρόνοι δημιουργίας* του ευρετηρίου θα είναι αρκετά αυξημένοι λόγω ακριβώς των πολλαπλών και συχνών συγχωνεύσεων των τμημάτων.



Σχήμα 6.12: Επίδραση του μεγέθους του μπλοκ στο χρόνο ανάκτησης των λιστών.

6.7 Εκ των προτέρων δέσμευση του ευρετηρίου στο δίσκο

Για το σύνολο των πειραμάτων που εκτελέσαμε επιλέξαμε να δεσμεύσουμε εκ των προτέρων τα αρχεία του ευρετηρίου στο δίσκο. Συγκεκριμένα, επειδή ανάλογα με το σύστημα αρχείων υπάρχουν κάποιοι περιορισμοί στο μέγιστο μέγεθος ενός αρχείου (συνήθως το μέγιστο μέγεθος αρχείου είναι 2Gb ή 4Gb), δεσμεύσαμε εκ των προτέρων στο δίσκο έναν αριθμό αρχείων (των 4Gb στην περίπτωση μας), ώστε να αποφύγουμε τυχόν επιβαρύνσεις που προκύπτουν από τον κατακερματισμό των αρχείων λόγω των διαδοχικών επεκτάσεων τους. Με τον τρόπο αυτό τα μπλοκς κάθε αρχείου είναι όντως συνεχόμενα στο δίσκο (με ένα ανυπολόγιστο ποσοστό κατακερματισμού), προσομοιώνοντας έτσι την ιδανική περίπτωση στην οποία θα έπρεπε να υλοποιήσουμε το δικό μας σύστημα αρχείων βάσει της άμεσης διεπαφής (raw interface) που παρέχει ο σκληρός δίσκος, για να εξασφαλίσουμε την απαιτούμενη συνέχεια.

Στην περίπτωση της μεθόδου της υβριδικής ενημέρωσης για την δημιουργία ευρετηρίων, η προσέγγιση που είχε ακολουθηθεί στην εργασία η οποία μελετούσε την μέθοδο ήταν πως για κάθε μεγάλη λίστα δημιουργούνταν ένα ξεχωριστό αρχείο στο δίσκο, ενώ κάθε φορά

που κάποιες νέες εμφανίσεις συλλέγονταν για μία μεγάλη λίστα αυτές προστίθονταν στο τέλος του αντίστοιχου αρχείου. Με τον τρόπο αυτό οι εμφανίσεις των μεγάλων λιστών αποθηκεύονται *λογικά* συνεχόμενες μέσα σε ένα αρχείο, αλλά μπορεί να μην βρίσκονται και *φυσικά* συνεχόμενες στο δίσκο, καθώς το αρχείο μπορεί να είναι κατακερματισμένο και τα μπλοκς του να είναι αποθηκευμένα σε διάφορα σημεία του δίσκου. Η υλοποίηση αυτή (ένα ξεχωριστό αρχείο για κάθε μεγάλη λίστα, προσθήκη νέων εμφανίσεων στο τέλος του αρχείου) βασίζεται εξ ολοκλήρου στο σύστημα αρχείων για την διατήρηση μικρού κατακερματισμού στα διάφορα αρχεία, και άρα στις μεγάλες λίστες. Όπως παρατηρούμε όμως από τα αποτελέσματα των πειραμάτων μας, ο φυσικός κατακερματισμός των αρχείων επιφέρει υπολογίσιμες επιβαρύνσεις, κυρίως κατά την απάντηση των ερωτημάτων, και άρα μας οδηγεί στο συμπέρασμα πως δεν μπορούμε να βασιζόμαστε στο εκάστοτε σύστημα αρχείων για την διατήρηση της φυσικής συνέχειας των μπλοκς των αρχείων στο δίσκο.

Για τους σκοπούς των πειραμάτων αυτών δεικτοδοτήσαμε μία συλλεγή κειμένων 100Gb, με τις προεπιλεγμένες παραμέτρους (B_p = 1Mb, P_t = 512Mb, P_f = 10Mb, α = 30%), αρχικά χωρίς να έχουμε δεσμεύσει εκ των προτέρων το ευρετήριο στο δίσκο, και έπειτα έχοντας δεσμεύσει εκ των προτέρων το ευρετήριο. Μετρήσαμε τόσο τους χρόνους δημιουργίας του ευρετηρίου όσο και τους χρόνου αναζήτησης κειμένων σε κάθε περίπτωση. Τα αποτελέσματα παρουσιάζονται στους Πίνακες 6.10, 6.11[6] και επιβεβαιώνουν τις προσδοκίες μας: κατά την δημιουργία του ευρετηρίου δεν παρατηρούμε κάποια αλλαγή στους χρόνους δημιουργίας στην περίπτωση που το ευρετήριο είναι δεσμευμένο εκ των προτέρων. Στην περίπτωση όμως της αποτίμησης ερωτημάτων βλέπουμε μία αρκετά σημαντική βελτίωση των χρόνων, και κυρίως στους χρόνους ανάκτησης των μεγάλων λιστών.

Πίνακας 6.10: Χρόνοι δημιουργίας ευρετηρίου με και χωρίς εκ των προτέρων δεσμευμένο ευρετήριο.

index preallocation	T_{build}
no	7621
yes	7640

Όπως προκύπτει από τα αποτελέσματα του Πίνακα 6.11, για μέγεθος μπλοκ 1Mb η σχετική επιβάρυνση του συστήματος μας σε σχέση με την ιδανική περίπτωση όπου όλες οι λίστες

[6] η διαφορά στις τιμές των πειραμάτων αυτών από τις τιμές που παρουσιάστηκαν στον Πίνακα 6.9 έγγυται στο γεγονός πως τα πειράματα εκείνα εκτελέστηκαν σε υπολογιστή με υποδεέστερα χαρακτηριστικά, χωρίς αυτό όμως να επηρεάζει την *σχετική* βελτίωση που παρατηρούμε στα αποτελέσματα με την χρήση ενός εκ των προτέρων δεσμευμένου ευρετηρίου.

Πίνακας 6.11: Χρόνοι ανάκτησης λιστών με και χωρίς εκ των προτέρων δεσμευμένο ευρετήριο.

index preallocation	T_{cnt}	T_{frg}	incr.	$T_{cnt,short}$	$T_{frg,short}$	incr.	$T_{cnt,long}$	$T_{frg,long}$	incr.
no	606	1333	120%	87	112	29%	519	1221	135%
yes	628	1153	83%	179	232	29%	449	920	105%

αποθηκεύονται συνεχόμενες στο δίσκο μειώνεται από 120% στο 83%, αν δεσμεύσουμε εκ των προτέρων το ανεστραμμένο αρχείο στο δίσκο. Παρατηρούμε πως για τις μικρές λίστες δεν έχουμε καμία επίδραση, ενώ για τις μεγάλες λίστες η εκ των προτέρων δέσμευση μειώνει πάρα πολύ τους σχετικούς χρόνους ανάκτησης.

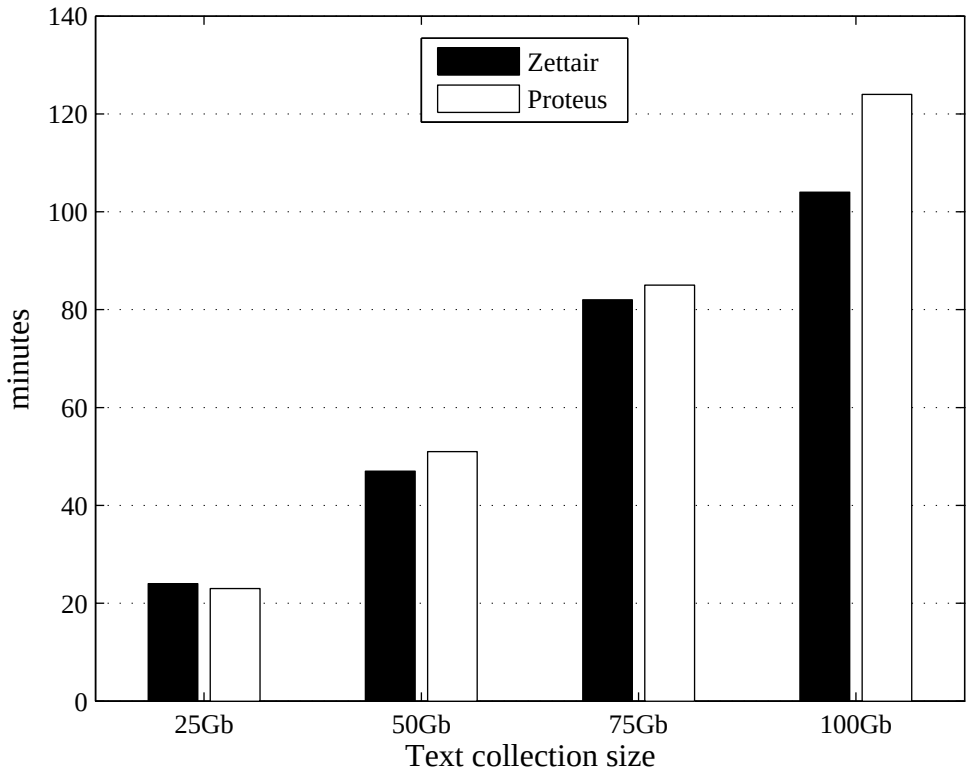
Στην περίπτωση του αλγορίθμου Πρωτέας, σε κάθε χρονική στιγμή υπάρχουν μόνο δύο αρχεία τα οποία μπορούν να επεκτείνονται: το αρχείο που αντιστοιχεί στο λεξικό και το ανεστραμμένο αρχείο. Το ανεστραμμένο αρχείο ξεκινά από μηδενικό μέγεθος και επεκτείνεται διαδοχικά ανά B_p bytes (μέγεθος μπλοκ εμφανίσεων), μέχρι να φτάσει τα 4Gb, οπότε και δημιουργείται ένα νέο αρχείο το οποίο με την ίδια λογική επεκτείνεται μέχρι να φτάσει και αυτό τα 4Gb, κτλ. Κάθε αρχείο που φτάνει τα 4Gb δεν επεκτείνεται περαιτέρω (τα μπλοκ εμφανίσεων που περιέχει όμως μπορούν να τροποποιούνται). Το λεξικό δεν ξεπερνά τα 4Gb για την συγκεκριμένη συλλογή, ενώ δεσμεύει μπλοκς με πολύ μικρότερο ρυθμό απ' ότι το ανεστραμμένο αρχείο. Άρα, συνεχώς ζητάμε από το σύστημα μπλοκς κυρίως για το ανεστραμμένο αρχείο, και άρα ο κατακερματισμός αναμένουμε να είναι αρκετά μικρός, δεδομένου πως δεν εκτελούνται άλλες διεργασίες στο σύστημα που να δεσμεύουν ταυτόχρονα μπλοκς και άρα τα μπλοκς που δεσμεύονται διαδοχικά από την διαδικασία δεικτοδότησης θα είναι με μεγάλη πιθανότητα και διαδοχικά στο δίσκο.

Στην περίπτωση της υβριδικής ενημέρωσης, σε κάθε χρονική στιγμή μπορεί να επεκτείνονται πάρα πολλά αρχεία, καθώς σε κάθε μεγάλη λίστα αντιστοιχεί και ένα ξεχωριστό αρχείο το οποίο μπορεί να επεκτείνεται σε κάθε διαδικασία συγχώνευσης λόγω της προσθήκης νέων εμφανίσεων σε αυτό. Άρα αναμένουμε οι επιπτώσεις του κατακερματισμού να είναι αρκετά πιο εμφανείς σε αυτή την περίπτωση, καθώς αναμένουμε πολύ μεγαλύτερα ποσοστά κατακερματισμού ανά αρχείο.

6.8 Σύγκριση με το Zettair

Για να είναι η μελέτη μας ολοκληρωμένη, συγκρίναμε το σύστημα μας με την αρχική έκδοση του Zettair. Συγκεκριμένα, συγκρίναμε τους χρόνους δημιουργίας του ευρετηρίου για συλλογές κειμένων μεγέθους 25Gb, 50Gb, 75Gb και 100Gb.

Το Zettair δημιουργεί το B-tree στο τέλος της εκτέλεσης του, κατά την διαδικασία της τελικής συγχώνευσης όλων των τμημάτων. Από την άλλη, το σύστημα μας διατηρεί το B-tree ενημερωμένο καθ' όλη την διάρκεια της δημιουργίας του ευρετηρίου. Με τον τρόπο αυτό το σύστημα μας επιτρέπει την αναζήτηση κειμένων καθ' όλη διάρκεια της δημιουργίας του ευρετηρίου, λαμβάνοντας υπ' όψιν του όλα τα αρχεία τα οποία έχουν δεικτοδοτηθεί μέχρι στιγμής, σε αντίθεση με το Zettair το οποίο υλοποιεί μία μέθοδο αντιστροφής στατικών συλλογών.



Σχήμα 6.13: Σύγκριση χρόνου δημιουργίας ευρετηρίου με το Zettair.

Από τα αποτελέσματα που παρουσιάζονται στο Σχήμα 6.13 προκύπτει καταρχήν πως για και τα δύο συστήματα ο χρόνος δημιουργίας του ευρετηρίου αυξάνεται περισσότερο από γραμμικά με την αύξηση του μεγέθους της συλλογής αρχείων. Για παράδειγμα, το zettair απαιτεί μόλις 24 λεπτά για να δεικτοδοτήσει 25Gb κειμένων, ενώ απαιτεί 103 λεπτά ($\geq$

4×24) για την δεικτοδότηση 100Gb ($= 4 \times 25$Gb). Από την άλλη, το σύστημα μας απαιτεί 22 λεπτά για τα 25Gb, αλλά 124 λεπτά για τα 100Gb. Έτσι, το σύστημα μας φαίνεται να απαιτεί 20% περισσότερο χρόνο για την δεικτοδότησει 100Gb κειμένων σε σχέση με το Zettair. Θεωρούμε το αποτέλεσμα αυτό αρκετά ενθαρρυντικό, δεδομένου πως συγκρίνουμε μία μέθοδο δεικτοδότησης δυναμικών συλλογών κειμένων με μία μέθοδο δεικτοδότησης στατικών συλλογών [7], ενώ επίσης στο σύστημα μας δεν έχουμε κάνει κάποια βέλτιση παραμετροποίηση.

Κεφαλαιο 7

Συμπερασματα - Μελλοντικη εργασια

7.1 Συμπεράσματα

Στην παρούσα εργασία μελετήσαμε το πρόβλημα της δημιουργίας και ενημέρωσης ενός ανεστραμμένου ευρετηρίου για δυναμικές συλλογές. Ο αλγόριθμος δεικτοδότησης που προτείνουμε βασίζεται στο διαχωρισμό των λιστών ανάλογα με το μέγεθος τους σε μικρές και μεγάλες. Τόσο οι μικρές όσο και οι μεγάλες λίστες αποθηκεύονται σε μπλοκς προεπιλεγμένου μεγέθους. Αποθηκεύουμε πολλές μικρές λίστες σε ένα μπλοκ, ενώ κάθε μεγάλη λίστα αποθηκεύεται σε αποκλειστικά δικά της μπλοκς. Οι μικρές λίστες ενημερώνονται χρησιμοποιώντας τον αλγόριθμο άμεσης συγχώνευσης, ενώ οι μεγάλες λίστες βάσει του αλγορίθμου επιτόπου ενημέρωσης.

Ορίσαμε την έννοια του εύρους, η οποία αναφέρεται στο σύνολο των μικρών λιστών που είναι αποθηκευμένες σε ένα μπλοκ ή σε μία μεγάλη λίστα που είναι αποθηκευμένη σε πολλά μπλοκς. Για κάθε εύρος διατηρούμε την πληροφορία του ποιες λίστες βρίσκονται αποθηκευμένες στο μπλοκ του και πόσος ελεύθερος χώρος υπάρχει σε αυτό, καθώς και ποιες λίστες έχουν συλλεχθεί στη μνήμη για το συγκεκριμένο εύρος και πόσο μέγεθος καταλαμβάνουν. Οι πληροφορίες αυτές διατηρούνται στον πίνακα ευρών. Χρησιμοποιώντας τις πληροφορίες αυτές και τον προτεινόμενο αλγόριθμο συγχώνευσης, κάθε φορά που

εξαντλείται η μνήμη επιλέγουμε να συγχωνεύσουμε με το δίσκο ένα υποσύνολο των λιστών της μνήμης, και συγκεκριμένα επιλέγουμε να συγχωνεύσουμε τις λίστες των ευρών εκείνων των οποίων η συγχώνευση προσφέρει το μέγιστο κέρδος, με κριτήριο την ελαχιστοποίηση των μετακινήσεων του δίσκου και την απελευθέρωση του μεγαλύτερου ποσοστού μνήμης.

Από τα αποτελέσματα των πειραμάτων προκύπτει πως η μέθοδος είναι πολύ αποδοτική για μεγάλα μεγέθη μπλοκ, με μία μικρή επιβάρυνση στο ποσοστό του αχρησιμοποίητου χώρου μέσα σε κάθε μπλοκ. Συγκεκριμένα, οι χρόνοι δημιουργίας του ευρετηρίου και ανάκτησης των λιστών είναι συγκρίσιμοι με τους αντίστοιχους χρόνους των αποδοτικότερων μέχρι στιγμής αλγορίθμων ενημέρωσης ευρετηρίου: η ανάκτηση των ανεστραμμένων λιστών απαιτεί μόλις 10% περισσότερο χρόνο από την ιδανική περίπτωση όπου όλες οι λίστες αποθηκεύονται συνεχόμενες στον δίσκο, ενώ η διαδικασία δημιουργίας του ευρετηρίου απαιτεί για 100Gb κειμένων 20% περισσότερο χρόνο από το Zettair, με την διαφορά όμως πως το Zettair κάνει δεικτοδότηση *στατικών* συλλογών, ενώ το σύστημα μας υποστήριζει ερωτήματα αναζήτησης και κατά την διαδικασία δεικτοδότησης της συλλογής.

7.2 Μελλοντική εργασία

Στα μελλοντικά μας σχέδια εντάσσεται καταρχήν η μελέτη της περίπτωση όπου χρησιμοποιούνται διαφορετικά μεγέθη μπλοκ για τα μικρά και μεγάλα εύρη, όπως επίσης και η μελέτη διαφορετικών μοντέλων υπολογισμού του κόστους συγχώνευσης που χρησιμοποιείται από τον αλγόριθμο μας στην επιλογή των ευρών που θα συγχωνεύσουμε, με σκοπό την αποδοτικότερη συγχώνευση των ευρών.

Στην κατηγορία της μελλοντικής εργασίας εμπίπτει επίσης ο σχεδιασμός και η μελέτη ενός αλγορίθμου αποτίμησης ερωτημάτων στον οποίο τα μπλοκς των μεγάλων λιστών θα ανακτώνται επιλεκτικά: διατηρώντας για κάθε μεγάλο μπλοκ το εύρος των αναγνωριστικών κειμένων τα οποία περιέχει, είναι πιθανό να έχουμε την δυνατότητα να ανακτούμε μόνο εκείνα τα μπλοκ των μεγάλων λιστών τα οποία περιέχουν τα αναγνωριστικά κειμένων που αναζητούμε.

Ενδιαφέρον επίσης παρουσιάζει το γεγονός πως κάθε εύρος μπορεί να συγχωνευτεί στο δίσκο ανεξάρτητα από τα υπόλοιπα εύρη κατά την διάρκεια της διαδικασίας συγχώνευσης, εφόσον κάθε εύρος αναφέρεται σε ένα διαφορετικό μπλοκ. Μάλιστα, κάτι τέτοιο μπορεί να γίνει χωρίς να χρειάζεται κάποιου είδους κλειδώματος στα μπλοκς του ευρετηρίου, εφόσον δεν υπάρχει η πιθανότητα δύο εύρη να χρησιμοποιούν το ίδιο μπλοκ. Φαίνεται έτσι αρκετά

ελκυστική η ιδέα της παράλληλης συγχώνευσης ευρών, δίνοντας έτσι την δυνατότητα στο δίσκο να χρονοπρογραμματίσει και να εξυπηρετήσει καλύτερα τις αιτήσεις ανάγνωσης και εγγραφής που υποβάλλονται σε αυτόν.

Μία μεγάλη κατηγορία μεθόδων που δεν καλύφθηκε στην παρούσα εργασία είναι η κατανεμημένη δημιουργία ευρετηρίων [23, 1, 25, 17, 16, 22]. Καθώς η πλειοψηφία των μεθόδων αυτών ακολουθούν την προσέγγιση της περιοδικής ανακατασκευής του ευρετηρίου, έχει αρκετό ενδιαφέρον η περίπτωση της επέκτασης του αλγορίθμου που προτείνουμε για κατανεμημένα περιβάλλοντα.

Τέλος, στα μακροπρόθεσμα σχέδια μας ανήκει και η μελέτη των διαφόρων θεμάτων που σχετίζονται με την ενσωμάτωση μιας μηχανής αναζήτησης στο λειτουργικό σύστημα [21, 6, 4].

Βιβλιογραφια

[1] Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[2] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB Conference*, pages 192–202, September 1994.

[3] Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems.

[4] Stefan Büttcher and Charles L. A. Clarke. Operating system support for full-text search in file systems.

[5] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 317–318, New York, NY, USA, 2005. ACM.

[6] Stefan Buttcher and Charles L. A. Clarke. A security model for full-text file system search in multi-user environments. In *USENIX Conference on File and Storage Technologies*, pages 169–182, San Francisco, CA, December 2005.

[7] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *ACM SIGIR*, pages 356–363, Seattle, Washington, USA, August 2006.

[8] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. Technical report, SUNY at Stony Brook, NY, USA, 1998.

[9] Ruijie Guo, Xueqi Cheng, Hongbo Xu, and Bin Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Conference on*

Information and Knowledge Management, pages 751–759, Lisboa, Portugal, November 2007.

[10] Steffen Heinz and Justin Zobel. Performance of data structures for small sets of strings. In Michael J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[11] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8):713–729, 2003.

[12] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[13] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Conference on Information and Knowledge Management*, pages 776–783, Bremen, Germany, October 2005.

[14] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[15] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *Australasian Computer Science Conference*, pages 15–23, Dunedin, New Zeland, January 2004.

[16] Maxim Lifantsev and Tzi cker Chiueh. I/o-conscious data preparation for large-scale web search engines. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 382–393. VLDB Endowment, 2002.

[17] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 396–406, New York, NY, USA, 2001. ACM.

[18] Alistair Moffat and Timothy A. H. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[19] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[20] Gonzalo Navarro, Edleno Silva de Moura, Marden Neubert, Nivio Ziviani, and Ricardo Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[21] Kyle Peltonen. Adding full text indexing to the operating system. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 386–390, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Berthier A. Ribeiro-Neto, Joao Paulo Kitajima, Gonzalo Navarro, Cláudio R. G. Sant'Ana, and Nivio Ziviani. Parallel generation of inverted files for distributed text collections. In *SCCC '98: Proceedings of the XVIII International Conference of the Chilean Computer Science Society*, page 149, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Berthier A. Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *ACM SIGIR*, pages 105–112, Berkeley, CA, August 1999.

[24] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, New York, NY, USA, 2002. ACM.

[25] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 211–224, San Francisco, CA, March 2004.

[26] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *ACM SIGMOD Conference*, pages 289–300, Minneapolis, Minnesota, May 1994.

[27] Anh Ngoc Vo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 290–297, New York, NY, USA, 1998. ACM.

[28] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[29] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[30] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.

[31] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

Παραρτημα: Πινακεσ αποτελεσματων

B_p=256Kb–64Mb, P_t=512Mb, P_f=10Mb, K_t=1.7, α=30%

B_p	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
256Kb	9937.404	4801.699	1567.327	2706.553	387.050	123623	20646	102977	0.076	0.026	30.29	63.70	12.69
512Kb	8804.757	3923.944	1552.194	1759.119	463.469	82022	16535	65487	0.094	0.027	34.76	63.25	16.19
1Mb	8134.493	3333.738	1539.159	1093.503	530.880	51727	12791	38936	0.120	0.028	39.31	62.45	20.57
2Mb	7774.200	3131.067	1646.249	678.311	616.889	31424	9678	21746	0.170	0.031	43.88	61.78	25.55
4Mb	7690.028	3124.506	1761.360	383.414	772.508	18042	7087	10955	0.249	0.035	48.05	60.70	31.23
8Mb	7738.180	3155.764	1888.421	200.692	825.516	9520	4802	4718	0.393	0.043	49.05	56.36	36.17
16Mb	7571.479	3017.708	1890.907	85.432	824.036	4625	2931	1694	0.645	0.050	47.93	51.10	39.65
32Mb	7585.754	2931.649	1793.722	38.318	867.065	2222	1636	586	1.096	0.065	43.34	44.22	39.91
64Mb	7455.010	2910.959	1769.713	18.564	846.535	1089	883	206	2.004	0.090	39.99	40.43	37.28

B_p=256Kb–64Mb, P_t=256Mb, P_f=10Mb, K_t=1.7, α=30%

B_p	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
256Kb	15597.748	10390.125	5242.263	4151.598	826.334	284990	51962	233028	0.101	0.018	29.33	60.68	13.33
512Kb	14124.455	9056.930	5081.894	2878.898	915.602	181124	39312	141812	0.129	0.020	33.75	60.62	16.79
1Mb	13031.921	8192.574	4967.877	1992.026	1029.875	113237	29782	83455	0.167	0.024	38.23	60.28	20.88
2Mb	12473.041	7812.993	5052.179	1350.356	1173.775	68950	22316	46634	0.226	0.029	43.09	60.26	25.93
4Mb	12187.237	7726.460	5290.752	776.740	1399.488	39474	16120	23354	0.328	0.033	47.19	59.36	31.41
8Mb	12219.143	7769.794	5618.209	402.286	1494.557	20822	10852	9970	0.518	0.040	49.03	56.28	36.24
16Mb	11572.419	7131.857	5228.657	180.356	1449.332	9869	6383	3486	0.819	0.052	46.60	49.40	39.56
32Mb	11122.665	6677.129	4844.038	75.150	1479.244	4724	3540	1184	1.368	0.063	41.06	41.61	38.90
64Mb	11146.711	6652.758	4832.095	34.352	1493.643	2392	1962	430	2.463	0.080	38.91	39.01	38.30

B_p=1Mb, P_t=512Mb, P_f=10Mb, K_t=1.7, a=1%–50%

α	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
1%	13462.682	7681.042	2407.665	4365.545	726.162	244684	23541	221143	0.102	0.020	86.67	96.78	78.29
5%	9971.599	4970.191	1875.885	2258.015	661.318	107775	14969	92806	0.125	0.024	67.61	86.99	54.03
10%	8928.260	4049.117	1601.865	1661.372	608.979	75706	12811	62895	0.125	0.026	55.73	77.94	40.75
15%	8543.563	3659.265	1532.091	1379.858	569.336	63801	12362	51439	0.124	0.027	48.70	71.82	32.53
20%	8351.264	3493.365	1528.121	1232.162	556.837	57664	12341	45323	0.124	0.027	44.27	67.59	27.24
25%	8172.516	3363.088	1514.018	1142.242	533.084	54033	12521	41512	0.121	0.028	41.05	64.36	23.07
30%	8118.188	3339.799	1546.925	1087.552	534.639	51727	12791	38936	0.121	0.028	39.31	62.45	20.57
40%	8085.749	3307.840	1610.507	1001.407	528.225	49337	13580	35757	0.119	0.028	37.37	60.18	16.89
50%	8151.960	3371.845	1723.024	949.314	532.263	48165	14488	33677	0.119	0.028	36.38	58.66	14.51

B_p=1Mb, P_t=512Mb, P_f=10Mb, K_t=1.0–8.0, α=30%

K_t	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
1.0	8296.626	3551.268	1887.803	866.307	615.858	45006	15577	29429	0.121	0.029	39.32	61.90	20.98
1.5	8042.748	3264.957	1533.960	1016.906	544.998	49694	13339	36355	0.115	0.028	39.31	62.49	20.38
1.7	8019.487	3275.954	1487.312	1087.539	531.526	51727	12791	38936	0.116	0.028	39.31	62.45	20.57
2.0	8058.927	3244.116	1396.651	1166.375	511.339	55008	12195	42813	0.115	0.027	39.09	62.45	20.14
2.5	8156.907	3289.299	1308.889	1315.874	494.338	60850	11534	49316	0.113	0.027	38.85	62.30	20.03
3.0	8229.394	3329.977	1232.164	1443.789	482.954	66688	11019	55669	0.112	0.026	38.91	62.56	19.90
4.0	8336.177	3487.455	1170.245	1692.607	455.936	78019	10419	67600	0.112	0.025	39.01	62.90	19.68
6.0	8737.647	3833.563	1083.376	2136.611	448.702	99836	9758	90078	0.111	0.024	39.23	63.40	19.57
8.0	9156.792	4188.634	1051.134	2531.521	442.160	120847	9401	111446	0.112	0.023	38.91	63.07	19.40

B_p=1Mb, P_t=512Mb, P_f=2Mb–40Mb, K_t=1.7, α=30%

P_f	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
2Mb	9045.749	3654.001	1778.895	1147.627	510.281	52696	12660	40036	0.141	0.029	39.38	62.57	20.55
5Mb	8375.287	3432.185	1656.374	1076.157	515.600	52345	12727	39618	0.130	0.027	39.38	62.58	20.54
10Mb	8114.003	3318.940	1535.272	1081.543	531.910	51727	12791	38936	0.120	0.028	39.31	62.45	20.57
15Mb	8270.024	3593.140	1654.573	1218.103	551.730	51323	12899	38424	0.128	0.032	39.39	62.54	20.60
20Mb	8381.512	3799.655	1794.816	1283.624	554.120	50884	12982	37902	0.138	0.034	39.30	62.46	20.51
40Mb	8675.128	4182.447	2063.782	1366.570	581.790	50035	13351	36684	0.155	0.037	39.10	62.21	20.37

B_p=1Mb, P_t=256Mb, P_f=2Mb–40Mb, K_t=1.7, α=30%

P_f	T_{build}	T_{merge}	$T_{merge,short}$	$T_{merge,long}$	T_{btree}	M_{total}	M_{short}	M_{long}	t_{short}	t_{long}	F_{total}	F_{short}	F_{long}
2Mb	13538.510	8444.772	5216.797	2031.854	948.588	117782	29233	88549	0.178	0.023	38.20	60.29	20.78
5Mb	12946.236	8117.665	5018.369	1916.294	974.893	116020	29464	86556	0.170	0.022	38.30	60.39	20.86
10Mb	13046.407	8221.754	5005.027	1988.631	1023.403	113237	29782	83455	0.168	0.024	38.23	60.28	20.88
15Mb	13278.398	8615.324	5300.911	2057.759	1046.299	111671	30140	81531	0.176	0.025	38.12	60.13	20.81
20Mb	13589.869	9035.136	5622.565	2123.647	1073.380	111203	30689	80514	0.183	0.026	38.25	60.30	20.86
40Mb	14242.387	9793.681	6300.989	2152.778	1115.728	109729	32487	77242	0.194	0.028	38.08	60.04	20.80

Retrieval time (B_p=1Mb, P_t=512Mb, P_f=10Mb, K_t=1.7, α=30%)

B_p	T_{cnt}	T_{frg}	incr	$T_{cnt,short}$	$T_{frg,short}$	incr	$T_{cnt,long}$	$T_{frg,long}$	incr	short terms	long terms	long blocks	blocks/long list
512Kb	568	4671	721.0	84	135	61.2	484	4535	835.8	11492	16442	580181	35.2
1Mb	756	3410	351.1	124	176	41.4	631	3234	412.3	13577	14357	292038	20.3
2Mb	823	2394	190.9	160	213	32.5	662	2181	229.3	16096	11838	147869	12.5
4Mb	979	2078	112.3	235	297	26.5	743	1781	139.5	18565	9369	75276	8.0
8Mb	1005	1255	24.8	453	530	17.0	551	724	31.2	21128	6806	38344	5.6
16Mb	902	1006	11.5	410	454	10.7	492	552	12.2	23356	4578	19027	4.2

Βιογραφικο

Ο Γιώργος Μαργαρίτης γεννήθηκε το 1983 στην Θεσσαλονίκη, όπου και ολοκλήρωσε τις λυκειακές του σπουδές το 2001. Το 2001 ξεκίνησε τις σπουδές του στο Τμήμα Πληροφορικής του Πανεπιστημίου Ιωαννίνων, τις οποίες και ολοκλήρωσε το 2005. Από τον Σεπτέμβριο του 2005 είναι μεταπτυχιακός φοιτητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων. Τα ερευνητικά του ενδιαφέροντα εστιάζονται στο χώρο του κατανεμημένου διαμοιρασμού αρχείων, και συγκεκριμένα στην αναζήτηση αρχείων σε τοπικά και κατανεμημένα συστήματα.