Ονοματεπώνυμα και Αριθμοί Μητρώων δημιουργών του Project

Απόστολος Καρβέλας / 1115201800312

Αθανάσιος Αναγνωστόπουλος / 1115201800006

Ιωάννης Παπαδημητρίου / 1115201800150

Σκοπός

Σκοπός του project είναι η υλοποίηση μιας λειτουργικής Ανεστραμμένης Μηχανής Αναζήτησης (ΑΜΑ) . Η συγκεκριμένη ΑΜΑ δέχεται ως είσοδους ερωτήσεις (queries) και κείμενα (documents) με σκοπό την αντιστοιχίση του κάθε document σε queries . Για την επίτευξη του σκοπού αυτού ελέγχεται εαν κάθε λέξη του query είναι παρόμοια με τουλάχιστον μία λέξη του document. Ο βαθμός ομοιότητας μεταξύ λέξεων που γίνεται αποδεκτός, καθώς και ο τρόπος με τον οποίο μετριέται η ομοιότητα αυτή, εξαρτάται απο το query, που σημαίνει ότι δυό queries με τις ίδιες λέξεις δεν είναι απαραίτητο να αντιστοιχίζονται στα ίδια documents.

Για την υλοποίηση αυτής της ΑΜΑ η είσοδος έχει τη μορφή ενός αρχείου txt με τα queries και τα documents. Κάθε γραμμή εισάγει ένα καινούριο query, τερματίζει ένα υπάρχων query ή εισάγει ένα καινούριο document για να αντιστιχιστεί με τα υπάρχοντα queries.

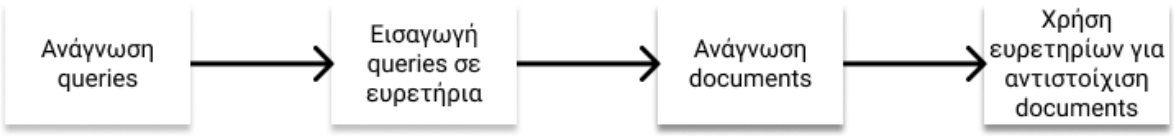
Σειριακή Υλοποίηση

1 ΤΑ QUERIES ΣΤΟ TXT

Στο αρχείο εισόδου txt τα queries έχουν τη μορφή **s queryid matchtype threshold numberofwords queryword_1 queryword_2 ... queryword_numberofwords**. Έστω για παράδειγμα η γραμμή **s 38 2 1 3 airport airlines deicing**. Το **s** σημαίνει ότι η γραμμή εισάγει query, το **38** είναι ο αντιπροσωπευτικός αριθμός του query, το **2** σημαίνει ότι χρησιμοποιείται edit distance για τη μέτρηση του βαθμού ομοιότητας, το **1** σημαίνει ότι οι λέξεις query-document για να αντιστοιχίστουν μπορούν να διαφέρουν κατά το πολύ έναν χαρακτήρα, το **3** σημαίνει ότι το query έχει τρεις λέξεις και τα υπόλοιπα είναι οι λέξεις του query.

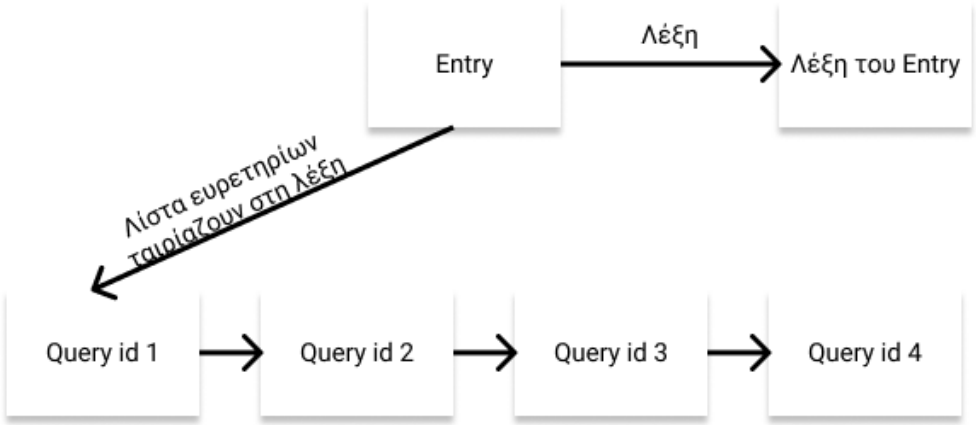
2 Η ΚΥΡΙΑ ΣΚΕΨΗ

Η γενική ιδέα είναι ότι το πρόγραμμα θα έχει κάποια ευρετήρια στα οποία θα εισάγωνται querywords και τα queryid's με σκοπό την επίτευξη της γρήγορης αντιστοίχισής τους με τα documents, ενώ τα queries τα ίδια θα αποθηκεύονται σε έναν πίνακα κατακερματισμού. Αυτή τη στιγμή το πρόγραμμα έχει τρεις τρόπους μέτρησης ομοιότητας (distances). Το κάθε distance έχει και το δικό του ευρετήριο που ελαχιστοποιεί τον χρόνο αναζήτησης με βάση τις ιδιότητές του ευρετήριου.



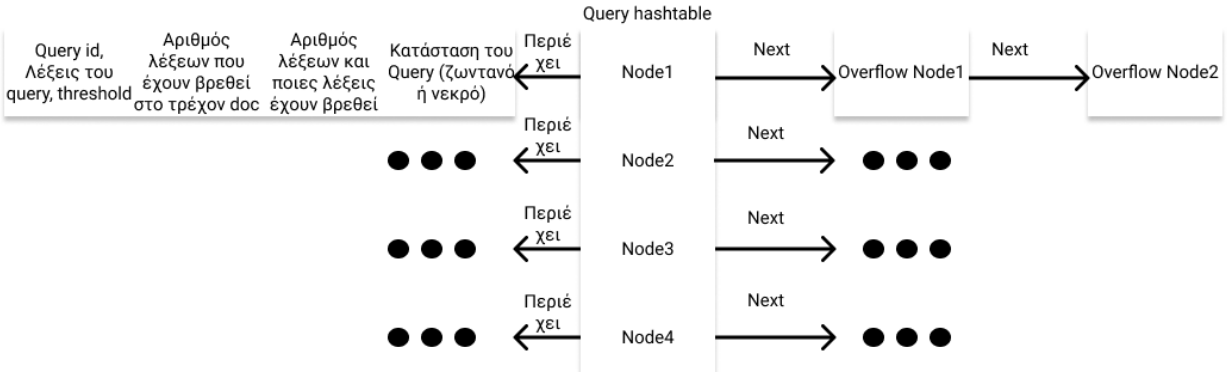
3 ENTRIES: ΟΙ ΛΕΞΕΙΣ ΣΤΑ ΕΥΡΕΤΗΡΙΑ

Η κυριότερη πληροφορία που αποθηκεύει το κάθε ευρετήριο είναι το entry. Ένα entry αντιπροσωπεύει μία λέξη που βρίσκεται μέσα σε τουλάχιστον ένα query και απαρτίζεται από τη λέξη word, καθώς και από μία λίστα payload_list με τα id των query ίδιου matchtype που περιέχουν την λέξη αυτή. Ο λόγος για τη δομή entry εξηγείται μετά από το document.



4 ΤΡΟΠΟΣ ΑΠΟΘΗΚΕΥΣΗΣ ΤΩΝ QUERIES ΚΑΙ ΕΠΙΒΕΒΑΙΩΣΗ ΑΝΤΙΣΤΟΙΧΙΣΗΣ

Ο πίνακας κατακερματισμού query_hashtable χρησιμοποιείται για να αποθηκευτούν οι λέξεις ενός query που έχουν βρεθεί σε ένα document με τη μορφή της κλάσης query_hash_node. Το query_hash_node -πέρα από τις πληροφορίες που δίνονται από το txt- κρατάει το id του τρέχοντος document (curr_doc), τον αριθμό λέξεων που έχουν ήδη αντιστοιχιστεί στο τρέχων document (words_found), εαν το query είναι ενεργό(alive) καθώς και ποιες από αυτές τις λέξεις έχουν βρεθεί είδη (word_c[ ]). Όταν το words_found είναι ίσο με το numberofwords (ή word_count στον κώδικα) το query έχει ταιριάξει με το document.

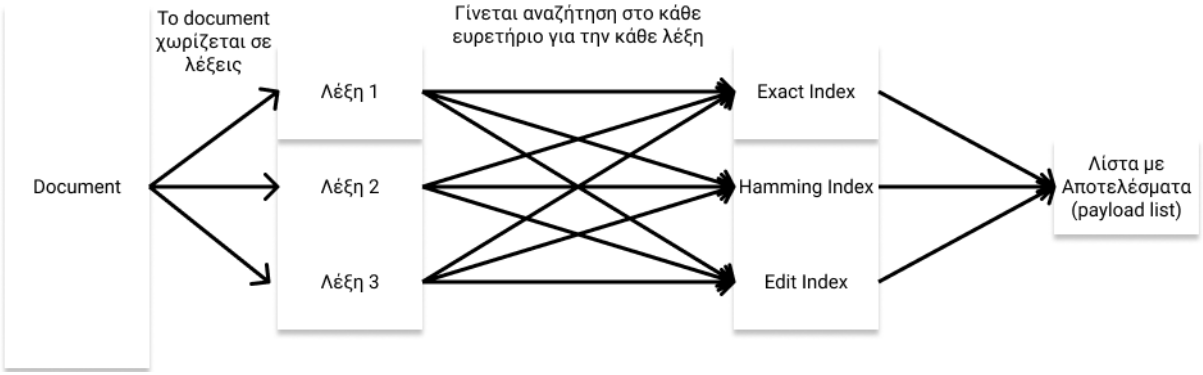


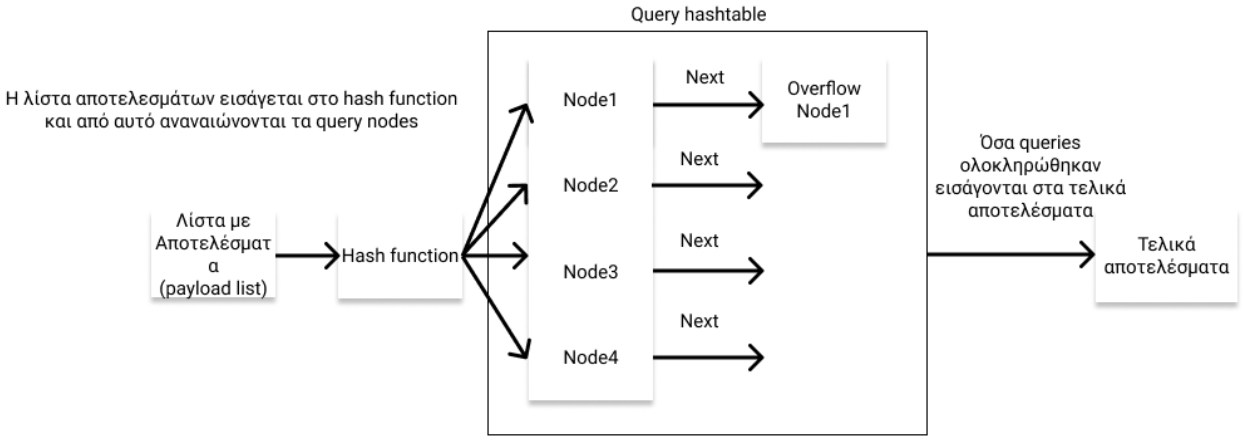
5 ΤΑ DOCUMENTS ΣΤΟ TXT

Στο txt τα documents έχουν τη μορφή **m documentid numberofwords word_1 word_2 ... word_numberofwords** (π.χ**. m 2 6243 http dbpedia resource list people ... minnesota**). Η γραμμή αρχίζει με m που σημαίνει ότι ακολουθεί document, το οποίο έχει id 2 και 6243 λέξεις που ακολουθούν ύστερα.

6 ΕΠΕΞΕΡΓΑΣΙΑ ΤΟΥ DOCUMENT

Όταν εισάγεται ένα document το πρόγραμμα το χωρίζει σε λέξεις. Για κάθε λέξη γίνεται αναζήτηση σε όλα τα ευρετήρια για κάθε threshold. Εαν βρεθεί η λέξη/entry σε κάποιο ευρετήριο τότε το query_hashtable ανανεώνει όλα τα queries με αυτή τη λέξη. Για την επίτευξη αυτού το πρόγραμμα διατρέχει το payload_list του entry -που έχει όλα τα id των query με τη λέξη- και τα ανανεώνει. Ολά τα query_hash_node's που ταιριάζουν όλες τους τις λέξεις με το τρέχων document επιστρέφουν το id τους στη λίστα αποτελεσμάτων. Εαν το currdoc ενός query δεν ταιριάζει με το id του τρέχοντος document τα words_found και word_c μηδενίζονται και ύστερα γίνεται η ανανέωση για την καινούρια λέξη. Αφότου ο parser διατρέξει όλες τις λέξεις το id του document, ο αριθμός των queries που ταίριαξαν και τα id's των queries αυτών βρίσκονται σε ένα αντικείμενο κλάσης doc.

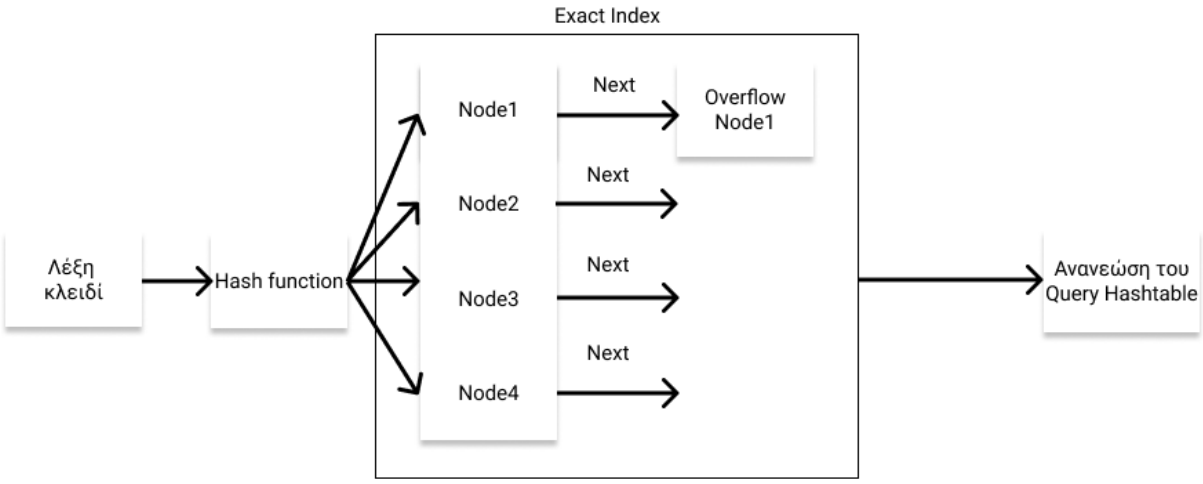




7 ΕΥΡΕΤΗΡΙΑ

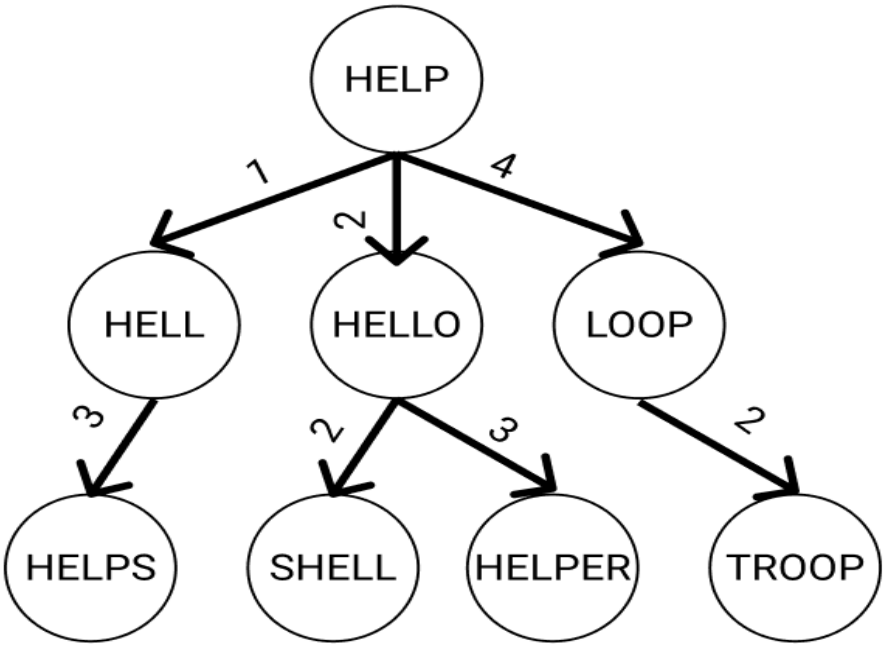
7.1 EXACT INDEX

Το Exact Index είναι ένας πίνακας στατικού κατακερματισμού που χρησιμοποιεί ως κλειδί-είσοδο μία λέξη. Όταν γίνει αναζήτηση στο ευρετήριο αυτό και βρεθεί η λέξη τα queries τύπου Exact Match που έχουν τη λέξη αυτή ανανεώνονται, αλλιώς επιστρέφει κωδικό λάθους. Σε περίπτωση που τουλάχιστον το 90% του πίνακα γεμίσει, γίνεται rehash, δηλαδή δημιουργείται καινούριος πίνακας διπλάσιου μεγέθους και όλα τα στοιχεία μεταβιβάζονται σε αυτόν.



7.2 EDIT INDEX

Το Edit Index είναι ένα BKTree. Το BKTree είναι ένα δέντρο όπου ο κάθε κόμβος περιέχει μία λέξη και έναν ακαθόριστο αριθμό παιδιών. Ο κόμβος έχει μία διαφορετική απόσταση για το κάθε του παιδί, ανάλογα με την διαφορά των λέξεων του κόμβου με τον κόμβο-παιδί (η συνάρτηση για τον υπολογισμό απόστασης καθορίζεται κατά την κατασκευασία του δένδρου και παραμένει σταθερή κατά την ζωή του δέντρου). Σε περίπτωση που δύο κόμβοι έχουν την ίδια απόσταση από τον κόμβο-γονέα τότε ο πρώτος θα εισαχθεί κανονικά ενώ ο δεύτερος θα εισαχθεί κάτω από τον πρώτο.



7.3 HAMMING INDEX

Το Hamming Index αποτελείται από έναν πίνακα από BKTrees, όπου το δέντρο i περιέχει μόνο λέξεις μεγέθους i+4 (καθώς το ελάχιστο μέγεθος λέξης είναι 4).

Παράλληλη Υλοποίηση

Για την υλοποίηση της ανεστραμμένης μηχανής αναζήτησης με παραλληλία θα χρησιμοποιήθουν οι προαναφερθήσες δομές με την διαφορά ότι η εισαγωγή των στοιχείων στα ευρετήρια, καθώς και η εύρεση λέξεων από τα έγγραφα σε αυτά εκτελούνται πολυνηματικά.

8 ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΤΗΣ ΕΡΓΑΣΙΩΝ

Ο χρονοπρογραμματιστής (Job scheduler) αποθηκεύει ορισμένες διεργασίες (Jobs) οι οποίες εκτελούνται από τα νήματα με σειρά first-in-first-out. Αφότου ένα νήμα εκτελέσει μία διεργασία λαμβάνει την επόμενη στην ουρά μέχρις ότου να μην υπάρχουν διεργασίες ή να λάβει διεργασία τερματισμού. Όταν η ουρά δεν έχει άλλο στοιχείο για να λάβουν τα νήματα τότε περιμένουν μέχρι να ικανοποιηθεί η συνθήκη cond_nonempty, δηλαδή το κύριο νήμα να προσθέσει καινούργιο στοιχείο στην ουρά, όπου και στέλει σήμα . Η ουρά είναι μια απλή συνδεδεμένη λίστα με job_nodes, όπου κάθε job_node έχει ένα από τα 4 ακόλουθα JobType:

- QUERY που εισάγεται κατά το start_query του κυρίου νήματος και προσθέτει ένα query στο job list ώστε να τα εισάχθει στα ευρετήρια,
- DOCUMENT το οποίο προστίθεται κατά το match document και αναζητάει τις λέξεις του δοσμένου document στα ευρετήρια,
- END_QUERY που εισάγεται κατά το start_query του κυρίου νήματος με σκοπό την απενεργοποίηση ενός query,
- BARRIER η διεργασία αυτή προστίθεται στην λίστα όταν πρέπει στο σημείο αυτό να εκτελέσουν τα νήματα φράγμα συγχρονισμού.

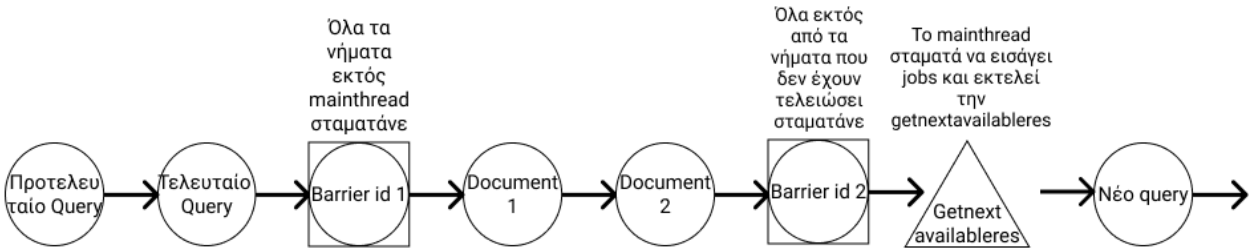
Για την υλοποίηση όλων των δομών αυτών έχει φτιαχτεί Job node με στοιχεία το id, την συμβολοσειρά, match_type, match_dist, το είδος του Job, και έναν δείκτη στον επόμενο κόμβο.

Υπάρχουν τριών ειδών barrier που έχουν ξεχωριστό id το καθένα, και χρησιμοποιούνται για δημιουργία φραγμάτων.

Το πρώτο barrier με id 1 προστίθεται στην job list πριν από το πρώτο document για κάθε batch, ώστε να έχουν ολοκληρωθεί οι εισαγωγές των στοιχείων πριν από τις αναζητήσεις.

Το δεύτερο barrier έχει id 2 και προστίθεται στην job list πριν από το πρώτο GetNextAvailRes ώστε να έχουν προστεθεί στην λίστα που με τα αποτελέσματα πριν τα δεχτεί η συνάρτηση αυτή. Οπότε για το barrier αυτό σταματάει και το main thread.

Τέλος το barrier με id 3 βρίσκεται στο τέλος του άπειρου βρόγχου για τα νήματα ώστε να έχει ολοκληρωθεί η εξέταση των αποτελεσμάτων από το getnextavailres πριν τερματίσει το πρόγραμμα.



9 ΚΥΡΙΟ ΝΗΜΑ

Κατά την εκτέλεση του προγράμματος καλείται η συνάρτηση InitializeIndex η οποία εκτός από την δημιουργία των δομών, αρχικοποιεί τα mutexes, τα barriers καθώς και τα νήματα που θα εκτελέσει το πρόγραμμα τα κατά την pthread_create καλεί την συνάρτηση consumer για την υλοποίηση του job scheduler. Ο αριθμός των νημάτων καθορίζεται μέσω του define NUM_THREADS στο core.h.

Στην συνέχεια, κατά το start query το κύριο νήμα προσθέτει την εργασία στην λίστα με στοιχεία του query ενώ πρώτα έχει κλειδώσει τον mutex br_mutex ώστε να μην μπορεί ένα νήμα να αφαιρέσει στοιχείο της λίστας την ίδια στιγμή. Αφού έχει εισάγει στοιχείο τότε στέλνει σήμα cond_nonempty για να περάσει το νήμα που περιμένει να προστεθεί εργασία.

Στο end query προσθέτει έναν κόμβο end_query στην λίστα.

Για το match document χρησιμοποιείται η μεταβλητή flag_q η οποία γίνεται 0 μόνο για το πρώτο document του κάθε batch και προσθέτει έναν κόμβο φράγματος στην job list. Στην συνέχεια, όπως με τα queries προσθέτει έναν κόμβο Document στην λίστα ώστε να τα εκτελέσει τα νήματα.

Η συνάρτηση GetNextAvailRes έχει και αυτή μεταβλητή flag_q η οποία ενεργοποιείται για την πρώτη εκτέλεση της συνάρτησης για κάθε batch με την διαφορά ότι αφού εισάγει φραγμό στην λίστα τότε κάνει wait μέχρι τα νήματα να ολοκληρώσουν τις εργασίες και περάσουν το νέο barrier. Αφού συνεχίσει το κύριο νήμα, κλειδώνει τον mutex mutexdoc για να εξετάσει το αποτέλεσμα από την λίστα και ύστερα τον ξεκλειδώνει.

10 ΝΗΜΑΤΑ

Αφού κληθεί το pthread_create από το κύριο νήμα τότε όλα τα νήματα που δημιουργούνται θα καλέσουν την συνάρτηση consumer. Για την υλοποίηση των barrier υπάρχει η μεταβλητή br_type.

Η br_type ισούται με 0 όταν δεν υπάρχει φράγμα, οπότε τα νήματα συνεχίζουν κανονικά.

Η br_type ισούται με 1 όταν κάποιο νήμα έχει βρει κόμβο barrier με id 1 οπότε πρέπει στήλει σήμα cond_nonempty ώστε να συνεχίσουν τα νήματα που έχουν σταματήσει στο wait του obtain και στην συνέχεια κάνουν barrier wait ώστε να συγχρονιστούν. Μετά το φράγμα αλλάζει το br_type πάλι σε 0.

Η br_type ισούται με 2 όταν κάποιο νήμα βρει ένα barrier με id 2 και δουλεύει όπως το προηγούμενο barrier με την διαφορά ότι αφού τα νήματα συγχρονιστούν στο barrier στέλνουν σήμα cond_br ώστε να συνεχίσει το κύριο νήμα με την getnextavailres.

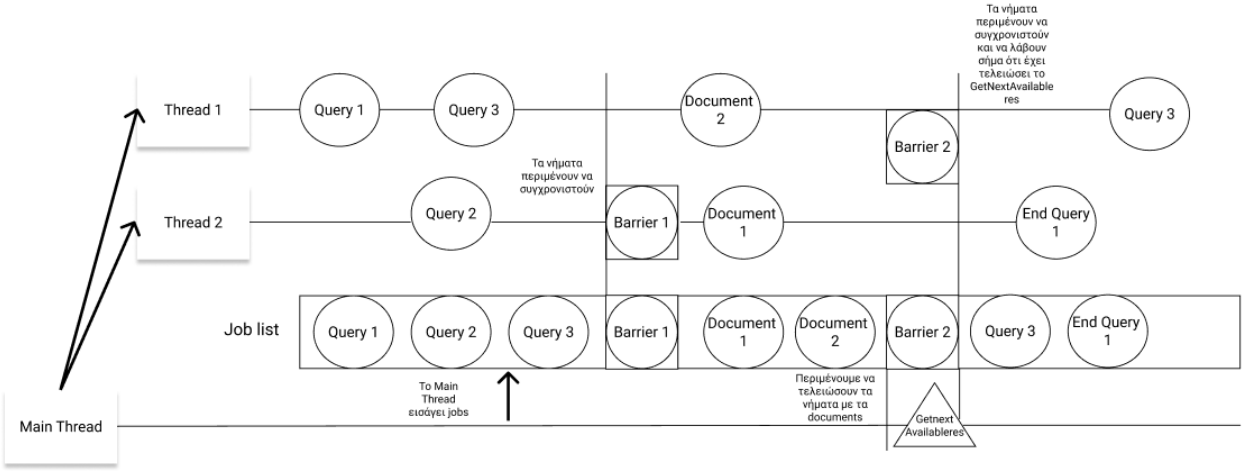
Τέλος, όταν το br_type είναι 3 κάνει το νήμα κάνει break οπότε και βγαίνει από τον άπειρο βρόγχο ώστε να καταστραφει με pthread_exit. Το br_type γίνεται 3 όταν ένα νήμα εντοπίσει και το τελευταίο doc id που εισάγεται από τον χρήστη μέσω define END_DOC στο core.cpp. Για παράδειγμα για το small_test.txt πρέπει να ισούται με 960, για το input30m.txt πρέπει να είναι 60 κ.λ.π.

Η συνάρτηση consumer είναι ένας άπειρος βρόγχος ο οποίος πρώτα βλέπει αν πρέπει να εκτελεσθεί κάποιο barrier και στην συνέχεια καλεί την συνάρτηση obtain που επιστρέφει ένα job node από την λίστα. Ύστερα από το obtain αν τα δεδομένα είναι null ή μορφής barrier το νήμα κάνει continue ώστε να μην υπολογισθούν και ξανά ελέγχει τα br_types για μερικά edge cases, όπως αν έχει ήδη δεχτεί εργασία από την λίστα και στην συνέχεια γίνει εφαρμοστεί το barrier. Οπότε σε αυτή την περίπτωση πρώτα υπολογίζουμε την εργασία που βρίσκεται στο νήμα και μετά γίνεται το φράγμα. Για τον υπολογισμό των εργασιών υπάρχουν τρεις συναρτήσεις που καλούνται ανάλογα το είδος της εργασίας ενώ στο τέλος ελέγχει αν είναι το τελευταίο document id ώστε να κάνει break από τον βρόγχο.

Για την εισαγωγή των queries στις δομές υπάρχει η συνάρτηση start_q, η οποία δουλεύει όπως και στην δεύτερη εργασία με την διαφορά ότι πριν από κάθε εισαγωγή σε ευρετήριο κλειδώνουμε έναν ξεχωριστό mutex για κάθε ένα και αφού γίνει η εισαγωγή τον ξεκλειδώνουμε, με αυτόν τον τρόπο η εισαγωγή γίνεται παράλληλα αφού πολλά νήματα μπορούν να εισάγουν ταυτόχρονα σε διαφορετικές δομές.

Η υλοποίηση του match document έχει γίνει μέσω της συνάρτησης match_doc, που είναι παρόμοια με της δεύτερης εργασίας αλλά η strtok έχει αντικατασταθεί με την strtok_r και υπάρχει mutex που κλειδώνει για την εισαγωγή των αποτελεσμάτων στην D_list. Ταυτόχρονα, μέσα σε κάθε μια συνάρτηση των ευρετηρίων για search, πριν κληθεί η add_one και η add_one_payload του qhashtable κλειδώνουμε τον συγκεκριμένο mutex. Έτσι τα searches εκτελούνται παράλληλα.

Για την υλοποίηση των end query υπάρχει η συνάρτηση end_q η οποία αρχικά ελέγχει αν το query που θα τερματίσει έχει εισαχθεί στα ευρετήρια. Αν δεν υπάρχει τότε αποθηκεύει το id σε έναν πίνακα και κάνει wait μέχρι ένα start_q βρει το id αυτό και εκτελέσει signal. Ενώ αν το βρει απλά κάνει την συνάρτηση end_query.



Αλλαγές κατά την παραλληλοποίηση

Για να κάνουμε παράλληλα τις αναζητήσεις έπρεπε να αλλάξουμε την υλοποίηση του Hash Table και συγκεκριμένα στο αρχείο q_hashtable.cpp και q_hashtable.h :

Επίσης δημιουργήθηκε το αρχείο q_satisfied.h με τα εξής στοιχεία:

doc_id: Το id του συγκεκριμένου document στο οποίο αναφέρεται το query_sat_node.

words_found: Δείχνει πόσες από τις λέξεις του query (MAX_QUERY_WORDS = 5) έχουν βρεθεί.

word_c: Ένας πίνακας με 1:1 αντιστοιχία με τον πίνακα word_arr του query_hash_node που δείχνει αν έχει βρεθεί ή όχι μία λέξη του query. Για παράδειγμα, αν έχει βρεθεί η πρώτη λέξη του query (word_arr[0]) τότε θα ισχύει ότι word_c[0] = 1.

Σειριακή υλοποίηση

Για την σειριακή υλοποίηση, όπως στην δεύτερη εργασία, δεν ήταν απαραίτητη η χρήση του query_sat_node μέσα στο query_hash_node καθώς τα documents έτρεχαν με την σειρά και δεν χρειαζόταν να αποθηκεύουμε τις πληροφορίες (σχετικά με το αν ικανοποιείται το query) για το κάθε ένα document.

Κάθε φορά που αλλάζαμε document καλούσαμε την συνάρτηση reset_val έτσι ώστε να μηδενίσει το words_found καθώς και τον πίνακα word_c.

Έτσι δεν χρειαζόταν η διαγραφή και η επαναδημιουργία όλου του κόμβου αλλά μόνο η αρχικοποίηση κάποιων τιμών του, με αποτέλεσμα να μειώνεται σημαντικά ο χρόνος εκτέλεσης.

Χρόνος εκτέλεσης: [2s:576ms], για το αρχείο small_test.txt στα linux της σχολής.

Βέβαια η υλοποίηση με αυτό τον τρόπο, παρόλο που είναι αρκετά γρήγορη δεν επιτρέπει την παράλληλη εκτέλεση των documents καθώς προϋποθέτει ότι όλα τρέχουν στην σειρά και δεν αποθηκεύει τις πληροφορίες για το κάθε ένα.

Παράλληλη υλοποίηση με λίστα query_sat_list:

Η πρώτη σκέψη ήταν η δημιουργία λίστας από query_sat_nodes μέσα σε κάθε query_hash_node ώστε να μπορούμε να κρατάμε πληροφορίες σχετικά με το αν ικανοποιείται το συγκεκριμένο query αλλά για πολλά διαφορετικά documents.

Χρόνος εκτέλεσης: [17s:328ms], με 4 threads και για το αρχείο small_test.txt στα linux της σχολής.

Το πρόβλημα ήταν ο χρόνος που απαιτεί η αναζήτηση σε λίστα, ειδικά όσο προσθέταμε περισσότερους κόμβους για περισσότερα documents. Αυτό είχε ως αποτέλεσμα την αύξηση του χρόνου εκτέλεσης του προγράμματος σε μεγάλο βαθμό παρόλο που το πρόγραμμα έτρεχε πλέον παράλληλα.

Ακόμα δεν υπήρχε κάποιος αποδοτικός τρόπος για την εύκολη διαγραφή των κόμβων από την λίστα εφόσον πλέον δεν χρησιμοποιούνταν με αποτέλεσμα την αύξηση του πλήθους των στοιχείων της λίστας και κατά συνέπεια του χρόνου αναζήτησης σε κάθε επανάληψη.

Παράλληλη υλοποίηση με πίνακα query_arr:

Στη συνέχεια αντικαταστήσαμε την λίστα με δυναμικό πίνακα αλλά με την διαφορά ότι το μέγεθος του πίνακα θα είναι όσο ο αριθμός των threads που χρησιμοποιούμαι για την εκτέλεση του προγράμματος (NUM_THREADS).

Αυτό γιατί ο μέγιστος αριθμός των documents που τρέχουν παράλληλα είναι όσο ο αριθμός των threads, αφού κάθε thread αναλαμβάνει από ένα document την φορά.

Επίσης χρησιμοποιώντας την ίδια υλοποίηση με το σειριακό πρόγραμμα, κάθε φορά που καλείται νέα ομάδα από documents (batch) αντί να διαγράφουμε και να ξαναδημιουργούμε τον πίνακα, καλείται η συνάρτηση reset_val για την αρχικοποίηση των μεταβλητών που μας δείχνουν αν έχει ικανοποιηθεί το συγκεκριμένο query.

Η υλοποίηση αυτή μας οδήγησε στον καλύτερο χρόνο εκτέλεσης καθώς η αναζήτηση στον πίνακα είναι άμεση, επειδή γνωρίζουμε το id του thread και κατά συνέπεια την θέση του query_sat_node στον πίνακα.

Επίσης επειδή το μέγεθος του πίνακα είναι συγκεκριμένο, όσο ο αριθμός των threads, σε αντίθεση με την περίπτωση της λίστας.

Χρόνος εκτέλεσης: [957ms], με 4 threads και για το αρχείο small_test.txt στα linux της σχολής.

11 ΣΥΓΚΡΙΣΗ ΧΡΟΝΩΝ (SMALL_TEST.TXT)

Threads / Optimizations	Με παράλληλα searches + inserts	Με παράλληλα inserts	Με παράλληλα searches	Σειριακή Εκτέλεση (Homework2)
1 Thread	2256[2s:256ms]	3882[3s:882ms]	2253[2s:253ms]	2576[2s:576ms]
2 Threads	1300[1s:300ms]	2310[2s:310ms]	1303[1s:303ms]	
3 Threads	1021[1s:21ms]	2303[2s:303ms]	1058[1s:58ms]	
4 Threads	957[957ms]	2326[2s:326ms]	1179[1s:179ms]	
8 Threads	1013[1s:13ms]	2363[2s:363ms]	1030[1s:30ms]	

Παρατηρούμε ότι η προσθήκη έστω και ενός νήματος επιταχύνει την εκτέλεση του προγράμματος σε σύγκριση με το σειριακό. Έτσι όσο ο αριθμός των νημάτων αυξάνεται ο χρόνος μειώνεται, με εξαίρεση τα 8 νήματα όπου ο χρόνος φαίνεται να ξανααυξάνεται. Τα αποτέλεσματα αυτά είναι αναμενόμενα, καθώς το περιβάλλον στο όποιο έτρεξε το πρόγραμμα (Linux της σχολής μέσω Putty) είχε την δυνατότητα παράλληλης εκτέλεσης με εως 4 νήματα, όποτε αυξανόταν το κόστος του συγχρονισμού χωρίς να γίνοταν κάποια βελτίωση ως προς την παραλληλοποίηση.

Ως προς την επιλογή μεθόδων παραλληλοποίησης, τα αποτελέσματα δείχνουν πως μία μίξη παράλληλης αναζήτησης και εισαγωγής στοιχείων αποφέρει τα καλυτερα αποτελέσματα. Και αυτά τα αποτελέσματα ήταν αναμενόμενα, καθώς το κόστος του συγχρονισμού (χρονοδιατριβή λόγω Barriers, Wait και κλειδώμενης μνήμης από mutexes) είναι ασήμαντο σε σχέση με την πολλαπλάσια ταχύτητα που αποφέρει η ταυτόχρονη εκτέλεση πολλών μεγάλων διεργασιών.

Επίσης αξιοσημείωτο είναι πως η παράλληλη αναζήτηση είναι ταχύτερη της παράλληλης εισαγωγής στοιχείων. Μπορούμε να υποθέσουμε ότι για αυτό ευθύνεται το συγκριτικά μεγαλύτερο βάρος της αναζήτησης στο πρόγραμμα, σε σχέση με το βάρος της εισαγωγής.

12 ΣΥΓΚΡΙΣΗ ΜΝΗΜΗΣ(SMALL_TEST.TXT)

Threads / Optimizations	Με παράλληλα searches + inserts	Με παράλληλα inserts	Με παράλληλα searches	Σειριακή Εκτέλεση (Homework2)
1 Thread	615,878,613	615,878,613	615,878,613	-
2 Threads	615,955,701	615,955,701	615,955,701	
3 Threads	616,032,728	616,032,915	616,032,789	
4 Threads	616,109,821	616,109,877	616,109,877	
8 Threads	616,418,229	616,418,229	616,418,229	

Ανεξαρτήτως της μεθόδους παραλληλισμού η μνήμη που δεσμεύεται παραμένει σχεδόν η ίδια, που είναι λογικό καθώς ανεξάρτητα της υλοποίησης η δέσμευση πόρων δεν αλλάζει.

Από την άλλη όσο αυξάνονται τα νήματα ο αριθμός bytes φαίνεται να αυξάνεται κατά ~70,000 ανά νήμα. Όσο αυξάνονται τα νήματα οι απαιτήσεις στη μνήμη απί τις συναρτήσεις συγχρονισμού (Barrier, Pthread_create etc.) αυξάνεται μαζί τους. Ακόμα για κάθε Query δεσμεύονται Query_sat_node's ίσα με τον αριθμό των νημάτων, οπότε για τα χιλιάδες Queries δεσμεύνται χιλιάδες κόμβοι για κάθε νήμα που προστίθεται.