Project - Ngram Detection

Τσεσμελής Δημήτρης 1115201400208
Φλωράκης Απόστολος 1115201400217
Αγαπίου Μαρίνος 1115201400002

Project(1ο παραδοτέο)

Τρόπος εκτέλεσης:

Για τη σωστή εκτέλεση του προγράμματος χρησιμοποιούμε την εντολή ./exec -i "init-file" -q "work-file".
Προαιρετικά στη γραμμή εντολής μπορεί να δοθεί και το όρισμα -untest "unitest_input.txt" για να εκτελεστεί και το Unit Test.
Το αρχείο "unitest_input.txt" περιέχει εντολές ίδιας μορφής με αυτές του work-file και εξάγει τα αποτελέσματα του Unit Test στο αρχείο "unitest_results.txt"

Έλεγχος αποτελέσματος:

Προς διευκόλυνση στέλνουμε το αποτέλεσμα της εκτέλεσης εκτός από την κονσόλα και στο αρχείο "output.txt".
Με την εντολή "diff output.txt ./datasets/small/small.result" μπορεί να γίνει έλεγχος του αποτελέσματος.

Περιγραφή των λειτουργιών και της δομής:

Αρχικά το πρόγραμμα διαβάζει την απαιτούμενη είσοδο από τα αρχεία, η οποία πραγματοποιείται με τον εξής τρόπο.
Το αρχείο διαβάζεται γραμμή γραμμή χρησιμοποιώντας τα stream της C++ και τύπο string για την προσωρινή αποθήκευση της κάθε γραμμής, ώστε να μην υπάρξει κίνδυνος υπερχείλισης του buffer (που περιέχει τη γραμμή).
Αν πρόκειται για το αρχείο "init-file" που αρχικοποιεί την δομή Trie, το string δίνεται στη συνάρτηση Insert, η οποία αναλαμβάνει την εισαγωγή ολόκληρης της γραμμής στη δομή.

Στην περίπτωση που το αρχείο είναι "work-file", ανάλογα με τον πρώτο χαρακτήρα του string (Q, D, A) πραγματοποιείται η αντίστοιχη λειτουργία (query, delete_ngram, insert) με όρισμα το string χωρίς χαρακτήρα αυτό.

Συνάρτηση Insert():

Για την ευκολότερη διαχείριση του string και την αποτελεσματικότερη εισαγωγή των επιμέρους λέξεων στη δομή, η συνάρτηση αυτή αρχικά σπάει το string στις λέξεις από την οποίες αποτελείται και τις αποθηκεύει σε έναν πίνακα από string (sentence_split)
Για κάθε μία από τις λέξεις αυτές και ξεκινώντας από τον κόμβο-ρίζα του δέντρου γίνεται η εξής διαδικασία.
Η i-οστή λέξη μπαίνει στο i-οστό επίπεδο του δέντρου μέσω της συνάρτησης insert_to_level.
Για την εισαγωγή της στο εκάστοτε επίπεδο, γίνεται έλεγχος στον πίνακα των παιδιών που εξετάζουμε για να διαπιστωθεί αν η λέξη υπάρχει ήδη. Η αναζήτηση γίνεται με Binary Search (logn)
Αν η λέξη υπάρχει ήδη στο επίπεδο δεν πραγματοποιείται κάποια ενέργεια εκτός και αν η λέξη είναι η τελευταία της πρότασης οπότε και ο κόμβος τίθεται final.
Αν η λέξη δεν υπάρχει και αν ο χώρος του πίνακα επαρκεί, η λέξη τοποθετείται στο τέλος του πίνακα και με διαδοχικά swap μεταξύ των λέξεων η λέξη καταλήγει στη σωστή θέση, ώστε ο πίνακας να είναι τελικά ταξινομημένος.
Στην περίπτωση που ο χώρος δεν επαρκεί, γίνεται διπλασιασμός του πίνακα (double_table) και έπειτα ακολουθεί η παραπάνω διαδικασία.
Η παραπάνω διαδικασία πραγματοποιείται επαναληπτικά μέχρι να εισαχθεί και η τελευταία λέξη του string

Συναρτηση Query():

Η συνάρτηση παίρνει ένα όρισμα string και με παρόμοιο τρόπο με την insert το κάνει split (sentence_split)
Με τη χρήση δύο δεικτών start και end και περνώντας επαναληπτικά τον πίνακα που προέκυψε από την sentence_split, προσδιορίζουμε όλα τα πιθανά ngram που θα αναζητήσουμε στη δομή μας (search_sentence).
Διατρέχοντας το ngram αναζητούμε την κάθε λέξη στο αντίστοιχο επίπεδο.
Σε περίπτωση που κάποια λέξη δεν βρεθεί η αναζήτηση αποτυγχάνει, ενώ εάν βρεθούν όλες οι λέξεις στο κατάλληλο επίπεδο και η λέξη του τελευταίου επιπέδου είναι final, το ngram προστίθεται στο αποτέλεσμα.
Τελικά η συνάρτηση επιστρέφει ένα string που περιέχει όλα τα ngram που βρέθηκαν.

Συνάρτηση Delete_ngram():

Η συνάρτηση παίρνει ένα όρισμα string και με παρόμοιο τρόπο με την insert το κάνει split (sentence_split)
Αναζητούμε την κάθε λέξη στο αντίστοιχο επίπεδο αποθηκεύοντας ταυτόχρονα σε ένα πίνακα, εφόσον τη βρούμε, τον κόμβο στον οποίο τη βρήκαμε.
Αν κάποια λέξη δεν βρεθεί στο κατάλληλο επίπεδο δεν μπορεί να γίνει διαγραφή.
Στη συνέχεια, διατρέχουμε τον πίνακα από το τέλος προς την αρχή (δηλαδή διατρέχουμε το δέντρο από κάτω προς τα πάνω).
Αν ο κόμβος που εξετάζουμε είναι κόμβος φύλλο ή ο κόμβος βρίσκεσαι σε παραπάνω επίπεδο, δεν έχει παιδιά και δεν είναι final τον διαγράφουμε. Αν ο κόμβος διαγραφεί, τότε θέτουμε τον δείκτη του πατέρα που έδειχνε στον κόμβο αυτό σε NULL
και στη συνέχεια αναπροσαρμόζουμε τον πίνακα που περιέχει τους δείκτες-παιδιά του πατέρα (rearrange_table).

Unit Test:

Unit test υλοποιήθηκε για τις συναρτήσεις insert, delete και query της δομής Trie. Για την check_insert, εκτελείται μία insert και μία query και ελέγχουμε με τη χρήση ενός assert αν το ngram που επιθυμούμε να εισάγουμε στη δομή εισήχθει σωστά. Αντίστοιχα, για την check_delete εκτελείται μία insert, μία query, μία delete, μία query και τέλος μία asset για να διαπιστωθεί ότι το ngram δεν υπάρχει στο Trie. Τέλος, για την check_query εκτελείται μία insert, μία query και μία assert για να διαπιστώσουμε αν το ngram υπάρχει στη δομή.

Project(2ο παραδοτέο)

Τρόπος εκτέλεσης:

Για τη σωστή εκτέλεση του προγράμματος χρησιμοποιούμε την εντολή ./exec -i "init-file" -q "work-file".
Με την εντολή make δημιουργούνται 3 εκτελέσιμα προγράμματα (static, dynamic και exec) και ανάλογα με την 1η γραμμή που θα διαβαστεί από το αρχείο "init-file" κάνουμε exec αντίστοιχο εκτελέσιμο.

Περιγραφή των λειτουργιών και της δομής:

1)Bloom Filter:
Η δομή bloom filter χρησιμοποιήθηκε για την γρήγορη διαπίστωση της ύπαρξης ή μη ενός ngram σε ένα αποτέλεσμα.
Για την ελαχιστοποίηση της false positive ακολουθούμε την παρακάτω λογική.

Επιλέγοντας αρχικά το προβλεπόμενο αριθμό εισαγωγών στο bit vector (n) καθορίζουμε δυναμικά τον αριθμό m (μέγεθος πίνακα bit vector) και k (πλήθος hash function) ώστε να έχουμε το μεγαλύτερο ανεκτό
error rate βάσει του τύπου (1-e^(-kn/m))^k (πηγή: http://llimllib.github.io/bloomfilter-tutorial/).

Ως hash function χρησιμοποιούμε την murmur3 καθώς μετά από πειραματισμούς καταλήξαμε στο ότι είναι η καταλληλότερη για γρήγορο hashing έχοντας ως κύριο χαρακτηριστικό την ανεξαρτησία και ακολουθώντας
την ομοιόμορφη κατανομή, χαρακτηριστικά που έχει διαπιστωθεί ότι είναι απαραίτητα για τέτοιου είδους εφαρμογές.
Η murmur3 στο εσωτερικό της περιέχει πράξεις όπως πολλαπλασμό, ολίσθιση και xor.
Δεδομένου ότι σε πολλές περιπτώσεις χρειαζόμαστε k hash functions χρησιμοποιούμε την cassandra ώστε βάσει της murmur3 να τις παράξουμε.

Το bloom filter δημιουργείται μία φορά και γίνεται re-initialize στην αρχή κάθε query.

2)Linear Hashing:
Για την υλοποίηση του linear hashing έχει ακολουθηθεί η λογική που παρουσιάζεται στο παρακάτω link: http://cgi.di.uoa.gr/~ad/MDE515/e_ds_linearhashing.pdf με μόνη διαφοροποίηση το διπλασιασμό
των θέσεων του bucket όταν συμπληρωθεί η χωρητικότητά του αντί τη δημιουργία αλυσίδας (λίστας) από bucket.
Το linear hashing μας παρέχει γρήγορη πρόσβαση στους κόμβους του 1ου επιπέδου (O(1)).
Με την προσθήκη της δομής αυτή έγιναν ορισμένες μικρές αλλαγές στις συναρτήσεις insert, query και delete.
Για την ακρίβεια, για την πρώτη λέξη ενός ngram που θέλουμε να εισάγουμε, αναζητήσουμε ή διαγράψουμε αντίστοιχα αντί της binary search στα παιδιά της ρίζα του Trie, χρησιμοποιείται το hash table.

3)TopK:
Για την εμφάνιση των topk ngrams έχει χρησιμοποιηθεί ένα linear hash table και ένα binary max heap.
Το hashtable χρησιμοποιείται για την ταχύτερη αναζήτηση λέξεων που υπάρχουν ήδη στο heap. Συγκεκριμένα, επιταχύνει την εύρεση κάποιου ngram στο heap από O(n) σε O(1).
Το binary max heap χρησιμοποιείται για τη γρηγορότερη εύρεση (O(k)) των topk ngrams καθώς και για τη διατήρηση της δομής σε ταξινομιμένη σειρά (heapify) σε χρόνο O(logn).
Σε σχέση με πιναθή υλοποίηση με στατικό πίνακα η χρήση binary heap ελαχιστοποιεί την κατανάλωση μνήμης καθώς κάθε φορά που θέλουμε να εισάγουμε ένα ngram που δεν έχουμε ξανά βρει στη ριπή, εισάγουμε στη δομή ακριβώς ένα κόμβο και δεν κάνουμε realloc που θα δημιουργούσε κενές θέσεις στον πίνακα.

Η εισαγωγή διακρίνεται σε δύο περιπτώσεις.

Είτε το στοιχείο δεν υπάρχει στον hash table οπότε και το εισάγουμε εκ νέου τόσο στο hash table όσο και στο heap στην κατάλληλη θέση, είτε το στοιχείο υπάρχει ήδη οπότε το αναζητούμε γρήγορα
στο hash table, έπειτα αυξάνουμε κατά ένα την τιμή εμφάνισής του στο heap και κάνουμε heapify.

Για την εύρεση των topk ngrams διατρέχουμε το heap από τη ρίζα προς τα κάτω k φορές και προσθέτουμε στο αποτέλεσμα κάθε ngram που σε προηγούμενη επανάληψη δεν είχε επιλεγεί.

Για κάθε batch δημιουργείται ένα νέο binary max heap και ένα hashtable.

4)Compress:
Για την υλοποίηση της compress έχουμε προσθέσει στην κλάση Trie_node τα πεδία char* words_array και short int * words_length για την αποθήκευση των λέξεων και του αριθμού των χαρακτήρων αυτών
καθώς και κάποιους int για να ξέρουμε ανά πάσα στιγμή το πόσες λέξεις έχουμε αποθηκευμένες και ποιά η χωρητικότητα των char*.
Η συνάρτηση compress εφαρμόζεται σε κάθε κόμβο του 1ου επιπέδου τον οποίο βρίσκουμε μέσω του hash table.
Έπειτα επισκέπτεται όλους τους κόμβους του υποδένδρου μέχρι να φτάσει σε κόμβο φύλλο όπου και ξεκινάει να συγχωνεύει τον κόμβο αυτό με τον κόμβο πατέρα του εφόσον ο τελευταίος δεν έχει άλλο παιδί,
τροποποιώντας κατάλληλα τα δεδομένα και τους πίνακες του νέου συγχωνευμένου κόμβου.
Στη συνέχεια, διαγράφει τον παλιό κόμβο παιδί και συνεχίζει την ίδια διαδικασία με παιδί τον συγχωνευμένο κόμβο.

Για τη σωστή αναζήτηση στη νέα compressed δομή χρειάστηκαν να γίνουν οριμένες τροποποιήσεις στη συνάρτηση search_sentence του 1ου παραδοτέου.

Αντί της binary search που αναζητούσε μια λέξη στον πίνακα children ενός επιπέδου πλέον ψάχνουμε αυτή τη λέξη και στον πίνακα words_array του παιδιού που περιέχει πιθανώς παραπάνω από μία λέξεις.

Project(3ο παραδοτέο)

Σκοπός του 3ου part την εργασίας ήταν η χρήση threads για την παραλληλοποίηση των Q (query) ερωτημάτων.
Για να επιτευχθεί αυτό έχει χρησιμοποιηθεί η δομή JobScheduler η οποία περιέχει μία ουρά από Jobs που θα ανατεθεί αργότερα σε κάποιο thread.
Η διαδικασία που ακολουθείται είναι η εξής:

Γενική περιγραφή:

Αρχικά δημιουργούμε στην αρχή της main τον JobScheduler με κενή ουρά από Jobs.
Κάθε φορά που έρχεται ένα νέο Query, δημιουργούμε μία νέα Job με το Query αυτό και την κάνουμε submit στον JobScheduler.
Κάθε φορά που έρχεται ένα νέο F, καλούμε τη συνάρτηση execute_all_jobs η οποία εκκινεί τα thread και στη συνέχεια την συνάρτηση wait_all_tasks_finish η οποία αναμένει για την ολοκλήρωση όλων των thread που δημιουργήθηκαν.
Όταν ολοκληρωθεί η παραπάνω διαδικασία, εκτυπώνουμε τα αποτελέσματα των queries και του top στο αρχείο και αδειάζουμε την ουρά του JobScheduler ώστε να φιλοξενηθούν στην επόμενη ριπή εργασιών οι επόμενες εργασίες. Με τον τρόπο αυτό αποφεύγουμε την συνεχή δημιουργία και καταστροφή του JobScheduler που πρόσθετε αρχικά καθυστερήσεις της τάξης των 2 δευτερολέπτων.

Περιγραφή λεπτομερειών υλοποίησης:

Για να επιτευχθεί ο συγχρονισμός των thread ώστε να μην υπάρχει κοινή προσπέλαση του critical section την ίδια χρονική στιγμή έχουν χρησιμοποιηθεί 2 conditional variables και 2 mutexes.
Πιο συγκεκριμένα, το 1 conditional variable και οι 1 mutex χρησιμοποιούνται για τον συγχρονισμό των threads πάνω στο queue ενώ τα υπόλοιπα για τον συγχρονισμό των thread στην

εισαγωγή στοιχείων στον BinaryHeap και στο HashTableHeap που χρησιμοποιούνται για το top.

Αλλαγές στην Query:

Από τη στιγμή που το κάθε query μπορεί να εκτελεστεί παράλληλα με κάποιο άλλο, είναι αναγκαίο στο σώμα της συνάρτησης "query" να μην γίνεται προσπέλαση κοινής μνήμης. Παρόλα αυτά, στο part 2 της εργασίας κατά την εκτέλεση των query ερωτημάτων προσθέταμε δεδομένα στο BinaryHeap ώστε να μπορούμε στο τέλος της ριπής να βγάζουμε τα αποτελέσματα του top και χρησιμοποιούσαμε ένα κοινό BloomFilter για όλα τα query μιας ριπής.

Για την αντιμετώπιση του προβλήματος με το BloomFilter, εφαρμόσαμε την στρατηγική όπου κάθε query φτιάχνει το δικό του BloomFilter.
Για την αντιμετώπιση των εισαγωγών στο BinaryHeap η αρχική μας προσέγγιση ήταν να κάνουμε την εισαγωγή των δεδομένων 1 φορά πριν την εκτέλεση του top, δηλαδή ακριβώς μετά την ολοκλήρωση των queries της ριπής. Παρόλα αυτα, με πειραματισμούς διαπιστώσαμε πως κάνοντας παράλληλα το κάθε thread την εισαγωγή στο BinaryHeap, με τον απαραίτητο συγχρονισμό, κερδίσαμε στον συνολικό χρόνο εκτέλεσης του medium dataset έως και 3 δευτερόλεπτα.

Δεδομένου πως η strtok() δεν είναι thread-safe συνάρτηση, την αντικαταστήσαμε με την αντίστοιχη safe version strtok_r() στα σημεία εκείνα που κληθήκαμε να εκτελέσουμε κώδικα παράλληλα και υπήρχε, σε σχέση με τα part1,part2

Συγχρονισμός στο Queue (job_to_execute):

Όλη η παρακάτω διαδικασία περικλείεται από ένα while(1) loop.
Αρχικά γίνεται lock ο mutex.

Ακολουθεί ένα while που εξετάζει αν υπάρχει κάποιο νήμα που διαβάζει ήδη κάποια Job από την ουρά ή αν η ουρά είναι άδεια. Αν η ουρά είναι άδεια τότε το νήμα τερματίζει και στέλνει ένα cond_broadcast σήμα στην cond var ώστε να "προκαλέσει" τον τερματισμό και κάποιου άλλου νήματος. Αν η ουρά δεν είναι άδεια αλλά υπάρχει κάποιο άλλο νήμα που διαβάζει ήδη τότε το νήμα αυτό αναμένει στην cond var ώστε να μην προκαλεί το φαινόμενο του busy waiting στο πρόγραμμα. Αν δεν ισχύει τίποτα από τα δύο τότε το νήμα συνεχίζει παρακάτω.
Έπειτα, γίνεται unlock ο mutex και pop μίας Job από την ουρά.
Στη συνέχεια, γίνεται ξανά lock ο mutex ώστε να γίνουν οι απαραίτητες τροποποιήσεις των κοινά διαμοιραζόμενων μεταβλητών καθώς και να σταλεί ένα cond_signal για να αφυπνιστεί κάποιο άλλο νήμα.
Τέλος γίνεται unlock ο mutex, ξενικά να εκτελείται το Job από το νήμα και γράφεται το αποτέλεσμα του Job στην κατάλληλη θέση του πίνακα query_results_buffer.

Συγχρονισμός στο BinaryHeap::insert:

Για να επιτευχθεί παράλληλη εισαγωγή δεδομένων στο BinaryHeap έχει ακολουθηθεί ακριβώς ίδια λογική με το συγχρονισμό στο Queue, δηλαδή έχει χρησιμοποιηθεί ένα ακόμη ζευγάρι conditional variable-mutex καθώς και η παραπάνω λογική για την αποφυγή του busy waiting.

Dynamic Version:

Όπως και στο κομμάτι του Static λειτουργούμε με τις ίδιες καινούργιες δομές δηλαδή με Job_scheduler, jobs , Queue και για την παράλληλη λειτουργία και τον σωστό συγχρονισμό των thread έχουμε τις ίδιες λειτουργίες που αναφέρθηκαν προηγουμένως .

Αρχικά έχουμε κατά την εκκίνηση της main μία ακέραια μεταβλητή current_version η οποία αρχικοποιείται σε μηδέν και αυξάνεται κάθε φορά όπως απαιτεί η εκφώνηση.

Για την υλοποίηση του Versioning προσθέσαμε στις δομές μας τα εξής :

Κάθε κόμβος Trie_node έχει δύο επιπλέον πεδία τα A_version, D_version που είναι ακέραιοι που αναπαριστούν την version εισαγωγής και διαγραφής αντίστοιχα.

Δηλαδή καθε ngram εισάγεται μαζί με την αντίστοιχη τιμή εισαγωγής του A_version και την τιμή του D_version αρχικοποιημένη σε μηδέν.

Εισάγωγή ngram :

Για εισάγωγή ngram που αποτελούνται από παραπάνω από μία λέξεις , η λέξη που θα εισαχθεί στον τελικό κόμβο δηλαδή η final θα έχει ως A_version = current_version , D_version = -1 και οι υπόλοιπες που θα εισαχθούν σε κόμβους σε πιο ψηλά επίπεδα θα έχουν ως A_version = -1 , D_version = -1. Δηλαδή για παράδειγμα αν γίνει εισαγωγή του ngram : "the cat is" μόνο ο τελικός-final κόμβος της λέξης "is" θα έχει τιμές: A_version = current_version , D_version = -1 ενώ οι υπόλοιποι κόμβοι θα έχουν A_version = -1 , D_version = -1.

Αν μία λέξη κάποιου ngram που εισάγουμε υπάρχει ήδη τότε δεν αλλάζουμε τις τιμές των A_version και D_version της, εκτός αν η καινούργια λέξη που εισάγουμε είναι final τότε σε αυτή την περίπτωση αλλάζουμε, δηλαδή κάνουμε update την τιμή του A_version του κόμβου σε αυτή του A_version της καινούργιας λέξης.

Query:

Για την λειτουργία query ερωτημάτων ακολουθούμε τα ίδια βήματα που γίνονται και στην έκδοση Static. Για την Dynamic έκδοση αλλάξαμε την συνάρτηση search_sentence (η οποία δέχεται ένα ngram και μας δείχνει αν αυτό υπάρχει στο trie) έτσι ώστε να λειτουργεί με βάση τα A_version και D_version του κάθε κόμβου. Συγκεκριμένα στην έκδοση του 2ου Part για κάθε ngram που δέχεται η search_sentence μας επέστρεφει 1 αν βρει το ngram στο trie και η τελευταία λέξη είναι σε κόμβο final. Για το 3ο Part για κάθε query κρατάμε την μεταβλητή query_version στην οποία αποθηκεύουμε το version που έγινε το query και την παιρνάμε στο αντίστοιχο job για τον Job_scheduler. Έπειτα για κάθε ngram που δέχεται η search_sentence μας επιστρέφει 1 αν βρεί το ngram στο trie και η τελευταία λέξη είναι σε κόμβο final και αν για την μεταβλητή query_version ισχύει :

A_version <= query_version && D_version > query_version. Διαφορετικά επιστρέφει μηδέν και στην περίπτωση αυτή δεν το εισάγουμε στο αποτέλεσμα.

Διαγραφη ngram :

Για την διαγραφή ενός ngram από την δομή trie αρχικά ψάχνουμε για το ngram στην δομή trie και στην συνέχεια (αν το βρούμε) πάμε μόνο στον final κόμβο του ngram και κάνουμε update την τιμή της μεταβλητής D_version του κόμβου. Για παράδειγμα αν γίνει διαγραφή του ngram : "the cat is" μόνο ο τελικός-final κόμβος της λέξης "is" θα πάρει D_version = current_version ενώ οι υπόλοιποι κόμβοι δεν θα υποστούν αλλαγή στην μεταβλητή D_version τους.

Σχεδιαστικές Επιλογές και Βελτιστοποιήσεις

- Ως γενικότερη αρχή και φιλοσοφία στην υλοποίηση μας, δίνουμε μεγαλύτερη σημασία στη μείωση του χρόνου εκτέλεσης δηλαδή στην ταχύτητα του προγράμματος μας σε σχέση με το χώρο που καταλαμβάνει στη μνήμη. Αυτό δε σημαίνει πως στην πλειοψηφία των περιπτώσεων που κληθήκαμε να κάνουμε μια σχεδιαστική επιλογή στην οποία τα δυο προαναφερόμενα χαρακτηριστικά ερχόντουσαν σε σύγκρουση, επιλέχθηκε η “ταχύτερη” τακτική σε σχέση με την “πιο αργή” αλλά με μικρότερη κατανάλωση μνήμης. Σαφώς και και όπου μπορούμε διαχειριζόμαστε τη μνήμη όσο το δυνατό “οικονομικότερα” αλλά επιλέγουμε να τη θυσιάσουμε ώστε να μειώσουμε το χρόνο εκτέλεσης όπου αυτό είναι εφικτό.

- Ενώ υπήρχε η αρχική ιδέα στη δομή trie ο πίνακας των παιδιών κάθε κόμβου να είναι της μορφής trie_node** children και ενώ η αρχική μας υλοποίηση ακολούθησε την προσέγγιση αυτή, επιλέχθηκε τελικά να αναπαρασταθεί ως trie_node* children με τους κόμβους παιδία να δημιουργούνται όλα μαζί σε συνεχόμενες θέσεις μνήμης και όχι δυναμικά ένα ένα, σε πιθανώς διαφορετικές θέσεις όπως συνέβαινε βάσει της αρχικής μας σκέψης. Επιτυγχάνουμε λοιπόν locality όσον αφορά τη μνήμη, δηλαδή όλα τα παιδιά του εκάστοτε κόμβου βρίσκονται σε συνεχόμενες θέσεις μνήμης, όποτε και μπορούν να φορτωθούν με αποδοτικότερο τρόπο όλα μαζί ώστε να ακολουθήσει οποιαδήποτε λειτουργία επιθυμούμε. Στο σημείο αυτό να τονιστεί πως με τον τρόπο αυτό δημιουργούμε κατευθείαν κόμβους πεδία που πότε δεν θα μας απασχολήσουν “φιλοξενώντας” πληροφορία για λέξη κάποιου ngram, παραμένοντας ουσιαστικά κενά. “Θυσιάζουμε” λοιπόν κάποιο μη χρησιμοποιούμενο χώρο μνήμης για να επιτύχουμε locality μνήμης και ταχύτητα εκτέλεσης.

- Σαν γενικότερη φιλοσοφία πραγματοποιούμε για την επέκταση και αντιγραφή μνημής τις συναρτήσεις realloc() και memmove() ώστε να επιτύχουμε locality μνήμης και ταχύτητα αντίστοιχα, αντικαθιστώντας σε αρκετά σημεία του κωδικά μας για παράδειγμα κλήσεις της συνάρτησης memcpy() που χρησιμοποιούσαμε εκτενώς αρχικά .

- Σε πολλά σημεία της άσκησης που απαιτούσαν δυναμική υλοποίηση ως προς την διαχειριση μνήμης είτε αφορόυσε char* arrays είτε άλλους τύπους πινάκων αντικειμένων, επιλέχθηκε η πολιτική του διπλασιασμού του του εκάστοτε χώρου σε σχέση με το χώρο που καταλάμβανε η δομή όταν ο χώρος καταλήφθηκε, αντί για παράδειγμα να πραγματοποιείται κάποιο realloc συγκεκριμένου πιο μικρού μεγέθους σε σχέση με το διπλασιασμό. Προτιμούμε πιθανώς μετα από κάποια double να έχουμε πιο πολλές άδειες θέσεις μνήμης παρά να πραγματοποιούμε πολλά realloc κατά τακτά χρονικά διαστήματα. Και αυτό γιατί μετά από κάποιους διπλασιασμούς θα δημιουργηθεί επαρκής χώρος οπότε και ο επόμενος διπλασιασμός θα αργήσει να χρειαστεί ενώ τα realloc που θα απαιτούνταν στο ίδιο διάστημα ναι μεν θα άφηναν πολύ λιγότερη αχρησιμοποίητη μνήμη θα επιβράδυναν όμως την εκτέλεση.

- Ως προς το Linear-Hashing επιλέγουμε αρχικά ένα μέγεθος hashtable και όταν αυτό γεμίσει και χρειαστούμε μια ακόμα θέση την οποία δεν έχουμε στη διάθεση μας επιλέγουμε να διπλασιάσουμε τον πίνακα αυτόν. Η επιλογή αυτή βασίστηκε πάνω στην πολιτική που ακολουθήσαμε και περιγράφηκε παραπάνω.

- Για το διάβασμα της εισόδου του προγράμματος προτιμήθηκε η επιλογή αυτή παρότι έγινε και δοκιμή με τη συνάρτηση getline της C. Με τα κατάλληλα ορίσματα η συνάρτηση αυτή δημιουργεί δυναμικά buffer οπού αποθηκεύει την κάθε γραμμή εισόδου. Πραγματοποιώντας ελέγχους η επιλογή αυτή απορρίφθηκε, καθώς υστερούσε χρονικά σε σχέση με το διάβασμα μέσω Streams και String της C++ το οποίο και τελικά προτιμήθηκε.

- Δοκιμάσαμε να εκτυπώνουμε το αποτέλεσμα ως άθροισμα επιμέρους μικρότερων αποτελεσμάτων, πραγματοποιώντας λιγότερες δηλαδή συνολικά εκτυπώσεις μεγαλύτερου μεγέθους string η κάθε μια. Παρατηρήθηκε μια χρονική βελτίωση, πολύ μικρή, σχεδόν αμελητέα με την τεχνική αυτή.

- Πειραματιστήκαμε με τα αρχικά μεγέθη των buffers και των ορισμένων μέσω define σταθερών ώστε όλα τα input να εκτελούνται ομαλά και σωστά.

- Παρατηρήθηκε πως μεγάλο μέρος του χρόνου εκτέλεσης καταλαμβάνει η συνάρτηση query όποτε και η προσοχή μας επικεντρώθηκε σε αυτή για να επιτύγχουμε μείωση του χρόνου εκτέλεσης. Η αρχική υλοποίηση της query, δεδομένου πως θέλαμε να εξασφαλίσουμε την ορθότητα των αποτελεσμάτων προτού προβούμε σε χρονικές βελτιστοποιήσεις ήταν εξαντλητική καθώς για κάθε δυνατό n-gram της εισόδου κάθε ερωτήματος πραγματοποιούνταν αναζήτηση στη δομή trie. Η δεύτερη προσέγγιση η οποία βελτίωσε σημαντικά τον χρόνο εκτέλεσης (απο 30 λεπτα για το small dataset σε 1 λεπτό στο part1) βασίστηκε στη παρατήρηση πως άμα η τελευταία λέξη ενός n-gram δεν υπάρχει στη δομή, δεν έχει νόημα να αναζητήσουμε n-gram το οποίο έχει ως αρχικό τμήμα το προηγούμενο n-gram και στο τέλος του

οσεσδήποτε λέξεις. Αφού γνωρίζουμε πως το αρχικό τμήμα δεν υπάρχει μπορούμε εκ των υστέρων να συμπεράνουμε πως οποιοδήποτε n-gram με την προσθήκη λέξεων στο τέλος του δεν θα υπάρχει επίσης.

- Επιπλέον, αναφορικά με την Query και τον τρόπο που διαπερνάμε και αναζητούμε τα ngram στο trie προσθέσαμε την εξής αλλαγή που μείωσε το χρόνο εκτέλεσης του medium dataset κατά 10 δευτερόλεπτα.
  Στην περίπτωση που θέλουμε να ψάξουμε το ngram “this is a dog”, αναζητούμε αρχικά στο HashTable τη λέξη “this”. Στη συνέχεια, για κρατάμε ένα δείκτη στη λέξη αυτή ώστε όταν θα αναζητήσουμε το “is” να μην ξανά ψάξουμε για το “this” αλλά να συνεχίσουμε από τον Pointer που δείχνει σε αυτό.

- Μια βελτιστοποίηση που απέφερε καλά αποτελέσματα χρονικά ήταν να διατηρούμε ένα Linear-Hashtable και να αποθηκεύουμε σε αυτό κάθε λέξη που εμφανίζεται έστω και μια φορά σε κάθε ριπή, διατηρώντας παράλληλα και ένα δείκτη στο αντίστοιχο heap-node του Binary-heap, με το οποίο επιλύουμε το ερώτημα top. Με τον τρόπο αυτό για λέξεις οι οποίες έχουν ξαναεμφανιστεί στη ριπή, μπορούμε πολύ εύκολα σε χρόνο O(1) να εντοπίσουμε το αντίστοιχο Heap-node που τις αφορά και να αυξήσουμε την τιμή του counter εμφάνισής τους, γλιτώνοντας την αναζήτηση της λέξης στο Heap. Μονο στην περίπτωση που μια λέξη εμφανίζεται για πρώτη φορα δεν έχουμε διαφορετική επιλογή απο το να την ανάζητήσουμε στο Heap και να την εισάγουμε στο Hashtable, ώστε να τη βρούμε “γρήγορα” σε μελλοντική αναζήτηση.
  Η παραπάνω προσθήκη επιτάχυνε την εκτέλεση κατά 25 δευτερόλεπτα.

- Στη στατική υλοποίηση της δομής έχει διατηρηθεί σε κάθε συμπιεσμένο κόμβο η λέξη που υπήρχε στον κόμβο πριν από τη συμπίεση. Σε κάθε κόμβο αποθηκεύουμε δηλαδή σε ένα char* τη λέξη αυτή η οποία μάλιστα συμπίπτει με την πρώτη λέξη του string που περιέχει όλες τις λέξεις που συμπιέστηκαν (ακολουθώντας πιστά την υλοποίηση της εκφώνησης). Με την προσέγγιση αυτή κατά την αναζήτηση ενός ngram και για κάθε λέξη που αναζητάμε σε κόμβο επόμενο επιπέδου πραγματοποιούμε μια "γρήγορη" binary search και αφού εντοπίσουμε τον κόμβο στον οποίο υπάρχει πραγματοποιούμε έλεγχο στο "συμπιεσμένο" string του κόμβου αυτού για πιθανώς επόμενες λέξεις του ngrmm. Σε διαφορετική περίπτωση θα έπρεπε να "αποσπούμε " την πρώτη λέξη κάθε "συμπιεσμένου" String για κάθε ένα κόμβο παιδί διαδικασία μη αποδοτική χρονικά.
  Προτιμούμε να δαπανούμε επιπλέον χώρο char* σε κάθε κόμβο του συμπιεσμένου trie ώστε να επιλέγουμε αποδοτικότερα και αποτελεσματικότερα κάθε φορά τον κόμβο που θα συνεχίσουμε την αναζήτηση του ngram που έχουμε ως στόχο.

Προσπάθειες Βελτιστοποίησης στην Query

- Ως προς την βελτιστοποίηση των λειτουργιών query και delete και σκεπτόμενοι πως άμα η τελική λέξη ενός ngram δεν υπάρχει στη δομή μας η αναζήτηση ή διαγραφή της είναι άσκοπη, κατά την εισαγωγή κάθε ngram αποθηκεύεται σε ένα Bloomfilter η τελευταία του λέξη. Όταν λοιπόν καλούμαστε να αναζητήσουμε η διαγράψουμε ένα ngram ελέγχουμε αρχικά άμα η τελευταία λέξη του υπάρχει στο Bloomfilter που αναφέρθηκε παραπάνω οπότε και συνεχίζουμε τη διαδικασία αναζήτησης η διαγραφής αντίστοιχα ενώ σε αντίθετη περίπτωση συμπεραίνουμε πως και ολόκληρο το

ngram δεν υπάρχει οπότε και δεν έχει νόημα οποιδίποτε περαιτέρω ενέργεια η οποία θα μας επιβάρυνε χρονικά.

- Την ίδια διαδικασία πραγματοποιήσαμε και για τις αρχικές λέξεις κάθε ngram. Παρατηρώντας πως η εύρεση των αρχικών λέξεων του n-gram στο Linear-hashtable απαιτούσε σε συγκεκριμένες περιπτώσεις σημαντικό χρόνο, αποθηκεύαμε κτά την εισαγωγή σε ένα Bloomfilter όλες τις αρχικές λέξεις των n-grams. Σε αυτό το Bloom-filter πραγματοποιούσαμε έλεγχο πριν από κάθε query-delete και σε περίπτωση που διαπιστώναμε πως το εν λόγω n-gram δεν υπάρχει διακόπταμε τη διαδικασία, γλιτώνοντας ουσιαστικά το χρόνο που απαιτείται για την αναζήτηση της πρώτης λέξης του n-gram στο hashtable. Αποδείχθηκε (όπως και για την παραπάνω περίπτωση με το bloom-filter των final λέξεων) όμως πως για τα n-gram που βρίσκουμε τελικά γίνεται διπλή αναζήτηση, πρώτα στο bloomfilter για να διαπιστωθεί η ύπαρξη τους και έπειτα στο hashtable για να βρούμε το δείκτη που αντιστοιχεί σε αυτά κια να συνεχίσουμε την εργασία μας. Άρα εν τέλει το αποτέλεσμα δεν βελτιώθηκε χρονικά.

- Μια άλλη ιδέα που υλοποιήσαμε βασίστηκε στο γεγονός οτι το βάθος του trie είναι συγκεκριμένο και γνωστό καθώς μπορούμε εύκολα να υπολογίσουμε το μέγιστο μέγεθος των λέξεων που εισάγονται στο δέντρο, το οποίο θα αποτελεί και το βάθος. Έτσι τα n-gram τα οποία θα αναζητήσουμε στη δομή trie μέσω της query θα μπορουν να έχουν μήκος το πολύ όσο το βάθος του δέντρου περιορίζοντας κατά πολύ τον αριθμό των n-gram που θα εξάγουμε από την πρόταση που αναζητούμε μέσω της query. Για το σκοπό αυτό διατηρούμε και ένα bloomfilter στο οποίο αποθηκεύουμε όλες τις final λέξεις. Πιο συγκεκριμενα: Έστω πως πραγματοποιούμε μια query με πρόταση που περιέχει 20 λέξεις από τις οποίες final γνωρίζουμε πως είναι η 10η λέξη και έστω το βάθος του trie 3. Δεν έχει νόημα να σχηματίσουμε οποιοδήποτε

διαφορετικό n-gram και να το αναζητήσουμε στο trie πέρα από αυτά που έχουν ώς final την 10 λέξη και βάθος το πολύ τρία, περιορίζουμε κατα πολύ λοιπόν τα n-gram που θα αναζητήσουμε αποκόπτοντας εξ' αρχής n-gram που ξέρουμε εκ των προτέρων πωε δεν μπορούμε να βρούμε. Έτσι στο παραπάνω παράδειγμα θα αναζητήσουμε μόνα τα n-gram που αποτελουνταια απο τις: 18η, 19η, 20η λέξεις, 19η, 20η λέξεις, 20η λέξη, τρία ουσιαστικά n-gram. Ο λόγος πιθανώς που το σκεπτικό αυτό δεν απέδωσε τα αναμενόμενα είναι πως οι final λέξεις είναι πολλές άρα δεν μπορούμε να πετύχουμε σημαντική βελτίωση με την απόρριψη και αποκοπή πολλών n-gram εκ των προτέρων. Σε περίπτωση που τα datasets είχαν διαφορετική μορφή, δηλαδή οι Q προτάσεις δεν περιείχαν πολλές final λέξεις, ο χρόνος εκτέλεσης με αυτή την version συγκριτικά με αυτή που παραδώσαμε θα ήταν σημαντικά μικρότερος.

- Βλέποντας πως με την επιπλέον βελτίωση της query δεν καταφέραμε να βελτιώσουμε τον χρόνο στραφήκαμε στην επιδιόρθωση και βελτίωση μικρότερων ατελειών που εν τέλει αθροιστικά μας έδωσαν χρονικά καλύτερο αποτέλεσμα της τάξης των 8-10 δευτερολέπτων στο medium dataset. Για παράδειγμα περιορίσαμε κατά πολύ συνεχόμενες κλήσεις συναρτήσεων όπως η strlen(), και δεσμεύσεις πινάκων μέσω new και malloc() οι οποίες απαιτούνταν μια φορά όμως εμείς τις πραγματοποιούσαμε συνεχόμενα. Οι συγκεκριμένες, λοιπόν, ενέργειες πραγματοποιούνται εξωτερικά της κύριας λειτουργίας και το αποτέλεσμα τους αποθηκεύεται, ώστε να επαναχρησιμοποιηθεί χωρίς να ξαναδημιουργηθέι ότι χρειάζεται. Διαγράψαμε επιπλέον και παρακάμψαμε ορισμένους προσωρινούς buffer ώστε να περιορίσουμε τις δεσμεύσεις μνήμης που πραγματοποιούμε.
  Επίσης, το γεγονός ότι πειραματιστήκαμε με τις παραμέτρους του προγράμματός μας όπως το αρχικό μέγεθος των πινάκων που αναπαριστούν τα παιδιά κάθε κόμβου του trie, βελτίωσε σημαντικά το χρόνο και

περιόρισε τη σπατάλη μνήμης. Αυτό συνέβη, διότι μετά από μελέτη των datasets, παρατηρήσαμε πως εκτός από το 1ο και το 2ο επίπεδο του trie, στα υπόλοιπα επίπεδα υπάρχουν κατά μέσο όρο 1-2 παιδιά. Έτσι, λοιπόν, επιλέξαμε να ορίσουμε το αρχικό μέγεθος του πίνακα ίσο με 2 ώστε να μην δεσμεύουμε παραπάνω χώρο από όσο χρειάζεται. Ακόμη, παρόμοια λογική ακολουθήθηκε και για το μέγεθος του bit vector του Bloomfilter.

Μετρήσεις και απόδοση:

Χρόνοι - Μνήμη:

Οι παρακάτω χρόνοι έχουν προκύψει από την εκτέλεση του προγράμματός μας σε ένα Desktop υπολογιστή με i5 επεξεργαστή και 8gb RAM.

Part1:

Small	1.485 sec	13.23 mb

Part2:

Small static	0.976 sec	21.26mb
Medium static	33.596 sec	244.3mb
Small dynamic	1.141 sec	18.63mb
Medium dynamic	40.578 sec	284.4mb

Part3:

Τα παρακάτω αποτελέσματα είναι με τα static input για τις static εκτελέσεις και με τα dynamic input για τις dynamic.

	4 threads	**2 threads**	**6 threads**	**Memory**
Small static	0.430 sec	0.765 sec	0.444 sec	8.263mb
Medium static	10.050 sec	21.500 sec	13.278 sec	31.92mb
Large static	31.655 sec	39.793 sec	32.504 sec	568.7mb
Small dynamic	0.598 sec	0.683 sec	0.549 sec	15.67mb
Medium dynamic	16.426 sec	22.874 sec	16.019 sec	46.92mb
Large dynamic	1 min 8 sec	1 min 8 sec	58.607 sec	879.3mb

Τα παρακάτω αποτελέσματα είναι με τα static input για τις dynamic εκτελέσεις ώστε να γίνει η σύγκριση της dynamic με τη static version.

Small dynamic	0.501 sec
Medium dynamic	13.109 sec
Large dynamic	28.280 sec

Συμπεράσματα:

Σχολιασμός για τον αριθμό των thread:

Παρατηρούμε πως ο χρόνος εκτέλεσης όταν χρησιμοποιούμε 2 thread σε σχέση με τα 4 thread είναι καλύτερος, όπως είναι και φυσιολογικό καθώς πετυχαίνουμε έτσι καλύτερη παραλληλοποίηση του προγράμματος. Η εκτέλεση με 6 thread, δεν μας αποδίδει καλύτερα αποτελέσματα καθώς ο επεξεργαστής στον οποίο έγιναν οι δοκιμές είχε τέσσερις πυρήνες, οπότε και από τα μεγέθη τα οποία δοκιμάσαμε τα 4 thread απέδωσαν το καλύτερα χρονικά αποτέλεσμα.

Σύγκριση Dynamic και Static εκδόσεων ως προς το χρόνο:

Η σύγκριση προέκυψε εκτελώντας τη dynamic έκδοση του προγράμματος με τα static αρχεία, ώστε να έχουμε ένα κοινό μέτρο. Σε γενικές γραμμές η Static έκδοση αποδίδει καλύτερα χρονικά αποτελέσματα σε σχέση με την Dynamic. Το γεγονός αυτό οφείλεται στο ότι η αναζήτηση ngram μέσω της query γίνεται πιο γρήγορα στο static λόγω των συμπιεσμένων κόμβων. Εφόσον ο κόμβος έχει συμπιεστεί μπορούμε να αναζητήσουμε εκεί παραπάνω από μια λέξεις του ngram που αναζητούμε, αποφεύγοντας την περαιτέρω διάσχιση του trie ώστε να αναζητήσουμε τις λέξεις που απομένουν.

Σύγκριση Dynamic και Static εκδόσεων ως προς το χώρο:

Σε γενικές γραμμές και για αντίστοιχα μεγέθη εισόδων η static έκδοση καταλαμβάνει περισσότερο χώρο στη μνήμη σε σχέση με την dynamic. Αυτό συμβαίνει καθώς σε κάθε compressed node του static trie έχει προστεθεί ένα char* για τη γρήγορη αναζήτηση της πρώτης λέξης του συμπιεσμένου string του κόμβου χωρίς να απαιτείται να το διασπάσουμε για να πάρουμε την πρώτη του λέξη, πρακτική που μας βοηθάει στη βελτίωση

του χρόνου όπως αναλυτικά έχει εξηγηθεί παραπάνω στο report μας.

Συγκρίσεις part1-part2:

Στο part2 πετύχαμε καλύτερους χρόνους σε σχέση με το part1. Το bloomfilter που προστέθηκε επιτάχυνε τον έλεγχο ύπαρξης μιας λέξης στο μερικό αποτέλεσμα μιας query σε σχέση με το strstr() που χρησιμοποιούσαμε για όλο το string που είχαμε σχηματίσει. Επίσης το linear-hashtable που αντικατέστησε το root του trie, ελαχιστοποίησε τον χρόνο προσπέλασης των λέξεων του πρώτου επιπέδου και δεδομένου πως το input αποτελείται από πολλά ngram μιας λέξης η βελτίωση χρονικά ηταν μεγάλη. Βελτιστοποιήθηκε επίσης η query όπως αναλυτικά έχει ειπωθεί παραπάνω.

Ως προς το χώρο είχαμε αύξηση όπως είναι και το λογικό αφού προσθέσαμε επιπλέον δομές δεδομένων.

Συγκρίσεις part2-part3:

Η παραλληλοποίηση κατά κύριο λόγω καθώς και ορισμένες αλλαγές που αναφέρθηκαν προηγουμένως οδήγησαν στην επιπλέον μείωση του χρόνου. Παρατηρήσαμε επίσης πως το Part2 καταλάμβανε πολύ χώρο στη μνήμη. Πραγματοποιώντας μικροαλλαγές καθώς και παρατηρώντας πως ο αριθμός των παιδιών κάθε κόμβου είναι μικρότερος από ότι υπολογίζαμε κατα μέσο όρο, ορίσαμε μικρότερο αρχικό μεγέθος του πίνακα των παιδιών (απο 100 σε 2) και μειώσαμε κατα πολύ τον χώρο μη πραγματοποιώντας δεσμεύσεις μνήμης που μας ήταν άχρηστες.