Project report part 2

Ονόματα:

Νικολετόπουλος Παναγιώτης ΑΜ: 1115201200126

Βιρβίλης Αντώνης ΑΜ:1115201200126

Περειεχομενα :

Part 1: Linear hash

Οι συναρτήσεις του linear hash και οι δηλώσεις τους μπήκαν στο functions.c και στο functions.h.

Πλέον η ρίζα του trie αντικαθιστάτε με ένα hash layer από το οποίο αποκτάμε πρόσβαση με πολυπλοκότητα ο(1) στο πρώτο παΐδι .

Επομένως το hash_layer struct έχει ένα πίνακα από hash buckets όπου κάθε hash bucket έχει έναν πινάκα από nodes.

Η αύξηση των hash_buckets ακλουθεί τις αρχές του linear hashing όπου αυξάνονται κατά ένα . Η υλοποίηση μας χρησιμοποιεί controlled split δηλαδή αυξάνονται τα buckets μόλις γέμιση ένα ποσοστό του hash(το έχουμε αφήσει 90%) .

Για μείωση του χρόνου εκτέλεσης μένει να δούμε αν μας συμφέρει να κάνουμε realloc το διπλάσιο των αριθμό των buckets αντί κατά ένα .

Σε περίπτωση overflow ενός bucket διπλασιάζουμε τον πίνακα των παιδιών του bucket . Αυτό μας διευκολύνει στην γρήγορη αναζήτηση και ταξινόμηση των παιδιών χωρίς να αλλάζουμε πολλά linked συνδεδεμένα buckets. Ωστόσο σε περίπτωση που το bucket γυρίσει στο αρχικό μέγεθος τότε δεν το συρρικνώνουμε.

Σε περίπτωση collision κάνουμε binary search όπως στο πρώτο part.

Για την hash function χρησιμοποιούμε την djb2 και σε περίπτωση που βρούμε τιμή μικρότερη αυτής του bucket που είναι να γίνει split τότε ξανά υπολογίζουμε την τιμή με άλλο mod.

Στο μεγαλύτερο μέρος των συναρτήσεων απλά προσαρμόσαμε hash layer ώστε να βρίσκουμε το πρώτο στοιχείο εκεί. Επομένως οι συναρτήσεις insertTrienode ,deleteTriennode() συνεχίζουν να δουλεύουν αναδρομικά ενώ η συνάρτηση lookup trie node δουλεύει σειριακά.

Part 2: Bloom filter

Το bloomfilter αποτελείται απο M το πληθος bits το οποιο δεσμευεται σαν ένας πίνακας από ακεραίους Στα αρχεια bloomfilter.c/.h υλοποιηθήκανε συναρτήσεις για τον binary χειρισμό του bloomfilter. Οι συναρτήσεις αυτές χρησιμοποιούνται από άλλες συναρτήσεις που χειρίζονται το bloomfilter.

Εχουμε επιλεξει σαν μεγεθος το 8192 για το bloomfilter, το οποιο με υπολογισμους και δοκιμες, ειδαμε οτι μας βολεύει.

Για το hashing μιας λεξης πριν την τοποθετηση της στο bloomfilter χρησιμοποιείται η murmur3 hash και συγκεκριμένα η 64bit έκδοση αυτής.

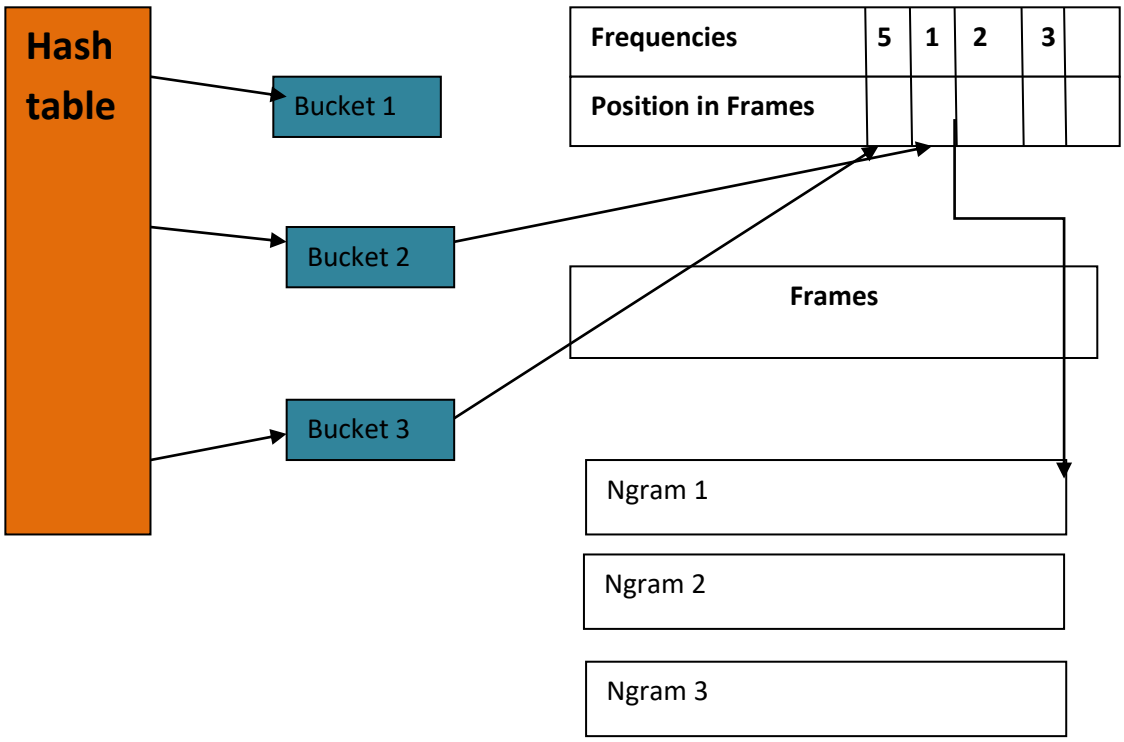
Καθε κληση της murmur3 μας επιστρέφει εναν πινακα με 2 τιμες. Ετσι κααλώντας 3 φορες την murmur3 παιρνουμε 6 διαφορετικες τιμες για να εισάγουμε στο bloomfilter.

Part 3: Top k frames and printing in F

Το κομμάτι των K frames και εκτύπωση ανά ριπή υλοποιήθηκε σε δυο φάσεις .

Στην πρώτη φάση θέλαμε απλά να εμφανίζουμε ανά ριπές όποτε αρχίσαμε να περνάμε σε ένα δυναμικό πίνακα κάθε ngram που βρίσκαμε και πέρναγε από το bloom filter . Στην συνέχεια εμφανίζαμε τα περιεχόμενα αυτού του πίνακα και αρχικοποιούσαμε τα στοιχεία της δομής του . Με αυτό τον τρόπο δεν χρειάζεται να κάνουμε free και malloc κάθε φορα πριν ξανά χρησιμοποιήσουμε τον πίνακα . Παρά μόνο realloc αν χρειαστεί να μπουν περισσότερα η μεγαλύτερα ngrams. Ωστόσο κρατάγαμε πόσα έχουν γίνει malloc συνολικά ώστε να σβηστούν .

Στην δεύτερη φάση που θέλαμε να εμφανίζονται και τα top k εισαγάγαμε δυο δομές ακόμα συνολικά . Ένα hash table που αυξάνετε γραμμικά και μια δομή που κρατάει πόσες φορές έχει εμφανιστεί κάθε ngram . Ένα σχήμα είναι πιο κατανοητό:



Όποτε την φορά που βάζαμε ένα ngram από ένα Q για εμφάνιση δημιουργούσαμε μια καινούργια εισαγωγή στο hash table με την οποία οδηγούμασταν στο πόσες εμφανίσεις έχει και ποια είναι η θέση του στην δομή Frames ώστε να μην χρειαστεί να κρατάμε το ngram και άλλου. Επίσης με αυτό τον τρόπο κάνουμε μια γρήγορη αναζήτηση για να βρούμε αν υπάρχει αυτό το ngram στα Q μέχρι τώρα και αν υπάρχει αυξάνουμε την τιμή εμφάνισης του.

Για να εμφανίσουμε τα top k κάνουμε μια ταξινόμηση στον πίνακα με τα frequencies με την quick sort και τον αντίστοιχο πίνακα με το που υπάρχει το ngram. Επομένως όταν έχει τελείωση η ταξινόμηση ξέρουμε που είναι τα ngrams με τις μεγαλύτερες τιμής εμφανίσεις.

Κατά την ταξινόμηση αν η τιμή εμφανίσεις είναι ιδία σε δυο ngram τότε βάζαμε μπροστά αυτό που είναι μικρότερο σε αλφαβητική σειρά το οποίο κατέληξε να έχει λάθοι. Για αυτό τον λόγο και για να γλυτώσουμε πολλά swaps όταν έχουν το ίδιο πλήθος (κυρίως στους άσσους που δεν χρειαζόμαστε συνήθως) φτιάξαμε μια συνάρτηση η οποία ταξινομεί μόνο τα κομμάτια από τα K πρώτα που χρειαζόμαστε .
Χρησιμοποιήσαμε την bubble sort σε αυτή την περίπτωση γιατί είναι πολύ μικρά τα νούμερα και με άλλους sorting αλγορίθμους μάλλον θα είχαμε μεγάλο overhead

Εμφανίζονται όλα στην σωστή σειρά.

Part 4: Static trie

Για το static trie φτιάξαμε δυο ξεχωριστά αρχεία με ξεχωριστές δομές από το δυναμικό hash: Το static_functions.c , static_functions.h.

Ίσως το αποτέλεσμα να ήταν το ίδιο αν πειράζαμε το functions.c με τις δομές του βάζοντας ακόμα μερικά πεδία και μερικά ifs στον κώδικα αλλά μάλλον θα κόστιζαν χρονικά και θέλαμε να κρατήσουμε ένα επίπεδο αφαίρεσης.

Οι αλλαγές από το δυναμικό trie είναι πλέον ότι το is_final είναι πίνακας με short ακέραιους και υπάρχει ένα extra πεδίο με όνομα number_of_words για να ξέρουμε πόσες λέξεις είναι αποθηκευμένες στο αρχείο.

Πειράξαμε λίγο την init_input για να γυρνάει 1 αν βρήκε την λέξη STATIC στην πρώτη σειρά. Σε αυτή την περίπτωση συνεχίζει να τα βάζει στο δυναμικό trie και στην συνεχεία τα κάνει compress .

Για το compress ξεκινάμε από τις ρίζες στο hash bucket και φτιάχνουμε ένα trie αναδρομικά μέχρι να βρούμε φύλο . Γυρνώντας κοιτάμε ποσά παιδιά έχει ο κόμβος και αν έχει μόνο ένα τότε το κάνουμε merge με το παιδί του.

Σε αυτό το σημείο να αναφερθούμε ότι το Part 4 έχει όλα τα προηγούμενα parts που αναφέρθηκαν (hashing, bloom filter ,top k).

Πέρα από την compress έχει υλοποιηθεί έχει και η εισαγωγή η οποία δεν χρησιμοποιείτε.

Για το lookup σε static trie πλέον κοιτάμε τις λέξεις στον κόμβο και αν τελειώσουν οι λέξεις τότε κοιτάμε στα παιδιά.

Πάλι σε αυτό το κομμάτι no leaks are possible

Μετρήσεις

Small dataset: 1.14 (χωρίς ανακατεύθυνση) , 0.92(με ανακατεύθυνση) όλα σωστά.

Medium dataset dynamic: 38.58(χωρίς) , 34.59 (με) δευτερόλεπτα

Όλα σωστά , no leaks are possible , no errors

Medium dataset static: 44 (χωρίς) , 41 (με) δευτερόλεπτα

Όλα σωστά, no leaks

Testing: no errors

Εμφανίσεις : Εμφανίζει ακριβώς τα ίδια αποτελέσματα με το small.result , medium_dynamic.result , medium_static.result.

Οδηγίες :

Make : compile project

Make test_project: compile unit testing

Make run : run small dataset

Make run2 : run test dataset

Make run3: run medium dataset

Make run_static: run medium static dataset

Unit testing

Για το κομμάτι του unit testing τροποποιήσαμε αρχικά τις συναρτήσεις ώστε πλέον να εφαρμόζονται στο γραμμικό hashing .

Με test_add και test_delete τσεκάρουμε αν όντως σβηστήκαν από το trie με την βοήθεια της test_if_exists.

Για να τσεκάρουμε την hash_function και γενικά το hash τροποποιήσαμε την συνάρτηση που εισάγει τα αρχικά ngrams ώστε με ανάλογο όρισμα να τα σβήνει κιόλας .

Επομένως με την test_hash_function περνάμε πρώτα τα ngrams στο trie (βλέποντας παράλληλα αν μπήκαν σωστά) και αμέσως τα σβήνουμε ένα ένα . Στο τέλος περιμένουμε όλα να έχουν σβηστεί . Αν δεν έχουν σβηστεί τότε υπάρχει πρόβλημα με την hash function και expansion του hash.

Για το unit test ενός στατικού hash χρησιμοποιήθηκαν οι εξής συναρτήσεις:

-Test_if_exists_static όπου τεστάρει αν υπάρχει ένα ngram στο static trie

-check_number _of_childs όπου τεστάρει αν υπάρχει μόνο ένα παιδί σε ένα στατικό trie(όπου θα ήταν λάθος)

-test_compress όπου τεστάρει εάν το compress έγινε σωστά βλέποντας αν υπάρχει μόνο ένα παιδί κάπου στο static trie για κάθε bucket

-test_everything_exists όπου με την βοήθεια της test_if_exists_static κοιτάει να δει αν όλα τα ngrams που χρησιμοποιήθηκαν στο να δημιουργηθεί στο δυναμικό trie υπάρχουν και στο static αφού γίνει compress.