Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Μέρος Β'

Μεντζέλος Ηλίας ***1115201400106***
Πλακιάς Φώτιος-Απόστολος ***1115201400161***
Πολύδωρος Αθανάσιος ***1115201400164***

Bloom Filter

Για να βρούμε πιο γρήγορα, αν ένα n-gram υπάρχει ήδη στο αποτέλεσμα της αναζήτησής μας χρησιμοποιούμε το bloom filter. Στην αρχή του προγράμματος μας το δημιουργούμε, έπειτα το χρησιμοποιούμε σε κάποια αναζήτηση, το καθαρίζουμε στο τέλος κάθε αναζήτησης (set-άρουμε όλα τα byte σε 0), ώστε να το χρησιμοποιούμε ξανά στην επόμενη αναζήτηση, μέχρι το τέλος του προγράμματος.

Έχουμε δημιουργήσει μια δομή bloom_filter η οποία περιέχει τον πίνακα με τα byte, το μέγεθος του και το πόσες hashfunction πρέπει να περάστει. Τα default values για το μέγεθος και τις hash functions είναι ορισμένα στο αρχείο bloom_filter.h . Μετά από αρκετές δοκιμές και στο μικρό αλλά και στο μεσαίο αρχείο βρήκαμε ότι το κατάλληλο μέγεθος πίνακα είναι 200.000, ώστε να τρέχει με σωστά αποτελέσματα το medium αρχείο, ενώ αρκούν 3 hash functions ώστε να πετύχουμε τον συνδυασμό απόδοσης και χρόνου.

Η συνάρτηση hash_results μας επιστρέφει έναν πίνακα με τα αποτελέσματα των hash function από τα οποίες "φιλτράραμε" τα αποτελέσματά μας. Έπειτα, μέσα στην bloom_insert_and_check, η οποία ελέγχει αν όλες οι θέσεις που μας έδωσαν οι hash function για τον πίνακα είναι 1 (γεμάτες) και μόνο αν είναι όλες οι θέσεις του γεμάτες υποθέτουμε ότι είναι διπλή εισαγωγή. Η συνάρτηση επιστρέφει 1 αν το n-gram υπάρχει και 0 αν δεν υπάρχει.

Σαν hash function χρησιμοποιήσαμε την συνάρτηση murmur (περισσότερες πληροφορίες στο https://en.wikipedia.org/wiki/MurmurHash). Για να έχουμε Ν διαφορετικές hash συναρτήσεις χρησιμοποιούμε την μέθοδο Cassandra Hashing, η οποία μιας παρέχει μια καλή διασπορά στα αποτελέσματα των διαφορετικών hash function.

Top K

Για την εύρεση των top-k ngrams σε μια ριπή χρησιμοποιούμε μια δομή heap μαζί με ένα δευτερεύων ευρετηρίου τύπου hash-table.
Ο σωρός αρχικοποιείται στην αρχή του προγράμματος και μετά από κάθε ερώτημα 'F'.
Ενημέρωση/Εισαγωγή κόμβου στο heap γίνεται κάθε φορά που βρίσκουμε ένα καινούριο αποτέλεσμα και αφού βεβαιωθούμε ότι δεν υπάρχει ήδη στο τρέχων αποτέλεσμα.
Η εισαγωγή κόμβου στο heap απαιτεί πρώτα τον έλεγχο μήπως υπάρχει ήδη αυτό το ngram στον σωρό, πράγμα το οποίο έχει γραμμική πολυπλοκότητα Για να το αποφύγουμε αυτό χρησιμοποιούμε linear hashing των κόμβων του heap έτσι ώστε να βρίσκουμε σε O(1) αν ένα ngram υπάρχει ήδη στον heap.

Αφού έχουμε έτοιμο το heap, η εύρεση των K μεγαλύτερων είναι εύκολη και έχει πολυπλοκότητα K*log(K). Το max βρίσκεται ήδη στην κορυφή και σίγουρα ένα από τα 2 παιδιά είναι το αμέσως επόμενο(τα προσθέτουμε σε μια ταξινομημένη λίστα). Διαλέγουμε το πρώτο παιδί από την λίστα, προσθέτουμε τα παιδιά του σε αυτήν και συνεχίζουμε μέχρι να φτάσουμε το K.

(Οι κόμβοι του heap δημιουργούνται άρα και καταστρέφονται από το hash table, και το heap δουλεύει με pointer σε αυτούς)

Hash

Το linear hashing χρησιμοποιήται για την γρήγορη εύρεση της πρώτης λέξης ενός ngram.
Για hash function χρησιμοποιήσαμε την sdbm (http://www.cse.yorku.ca/~oz/hash.html).
Αρχικά το root μας πλέον έχει για children ένα hash table αντί για trie node . Το hash table αποτελείται από bucket που περιέχουν τα παιδιά σε trie_node. Το linear hashing υλοποιήθηκε σύμφωνα με το σύνδεσμό που δόθηκε για αυτό. Δηλαδή μόλις γεμίσει ένα bucket τότε ξεκινάει η expand όπου κάνει realloc το hash table (Μια θέση παραπάνω). Στο split των δεδομένων αν τύχει να πάνε όλα μας τα δεδομένα στο καινούργιο bucket και το καινούργιο παιδί πρέπει να μπει και αυτό εκεί , τότε κάνουμε realloc το νέο bucket. Αξίζει να σημειωθεί ότι κάναμε μια μικρό αλλαγή στη binary search για να χρησιμοποιείται και στην αναζήτηση μέσα σε bucket αλλά και στα trie nodes . Ο καθαρισμός του χώρου γίνεται απλά περνώντας σειριακά από κάθε bucket και κάθε παιδί του bucket όπου και ελευθερώνεται

Static Files

Αρχικά χρησιμοποιήσαμε μια νέα δομή που λέγεται ultra_node που περιέχει τον πίνακα με τα μεγέθη των λέξεων και αν είναι τερματικές καθώς και το μέγεθος του πίνακα. Αυτή η δόμη περιέχεται από την trie node όπου και αρχικοποιήται με NULL . Για την συμπίεση του trie δημιουργήσαμε μια συνάρτηση που ουσιαστικά είναι μια υλοποίηση του DFS αλγορίθμου για να περιπλανηθούμε σε όλο το trie. Μέσα στην DFS καλείται η compress όπου παίρνει ένα trie_node και πρώτα ελέγχει αν μπορεί να συμπιεστεί με το παιδί του και αν με πόσα επίπεδα θα συμπιεστεί . Συγχρόνως εκτός από το μέτρημα των συμπιέσεων που πρέπει να

γίνουν κρατάμε και το μέγεθος της νέας λέξεις . Έπειτα ακολουθεί η επανάληψη που θα συμπιέσει τις λέξεις και θα χρησιμοποιήσει το struct που αναφερθήκαμε παραπάνω. Επίσης υλοποιήθηκε νέα binary search για να εφαρμόζεται σε στατικό trie .

Διαφορά Στατικού και Δυναμικού trie

Για να καταλάβουμε ποιο είναι πιο γρήγορο αλλάξαμε τη πρώτη λέξη του medium.static σε dynamic και παρατηρήσαμε ότι οι χρόνοι είναι σχεδόν ίδιοι.
Το πλεονέκτημα του στατικού είναι ότι απλά μόλις γίνει το compress η δεσμευμένη μνήμη είναι λιγότερη. Τέλος πιστεύουμε ότι σε μεγαλύτερα αρχεία init με περισσότερη πληροφορία σίγουρα το static βοηθάει τη δεσμευμένη μνήμη αλλά επιπλέον πιστεύουμε ότι θα βοηθάει και στο χρόνο.