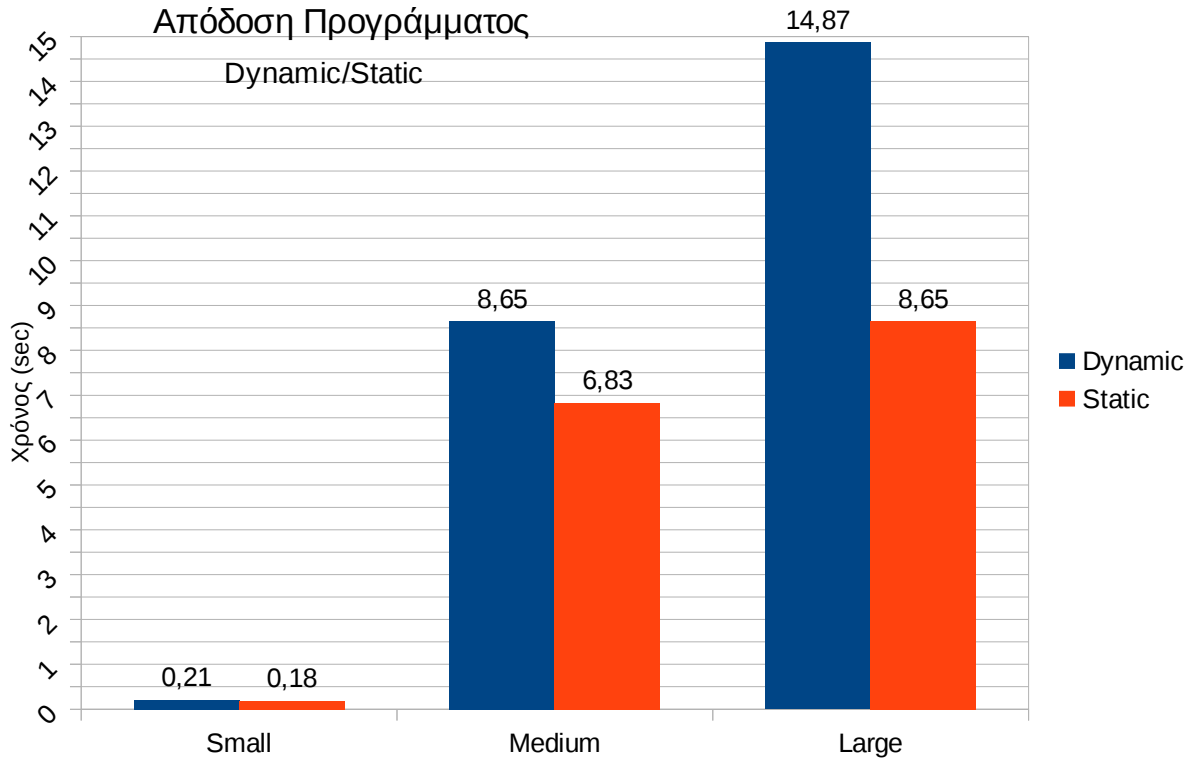
Εργασία του Μαθήματος :

# Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Υλοποιήθηκε από τους:
Γιώργος Μανδηλαράς 1115201200097
Θοδωρής Μανδηλαράς 1115201200098



# Table of Contents

# Πρωτεύουσες Δομές της Εργασίας

## trie_node

Είναι ο βασικός κόμβος της δομής, περιέχει την λέξη και διάφορες άλλες πληροφορίες που χρειάζονται. Την λέξη την αποθηκεύουμε στατικά σε ένα char array αν η λέξη είναι μικρότερη από έναν αριθμό SIZE όπου έχει γίνει defined στο trie_node.h, αλλιώς αποθηκεύεται δυναμικά. Υπάρχει το πεδίο isFinal το οποίο είναι μια bool μεταβλητή που κρατάει την πληροφορία αν ο κόμβος μας είναι τελικός η όχι. Έχει επίσης ένα trie_node * όπου σε αυτόν δημιουργούμε έναν δυναμικό πίνακα trie_node στον οποίο θα αποθηκευτούνε τα παιδιά του κόμβου. Αυτός ο πίνακας περιέχει trie_node κόμβους, και όχι δείκτες σε trie_node, έτσι ώστε να έχουμε καλύτερο locality με σκοπό να αποφεύγουμε αστοχίες στην cache. Η μεταβλητή size δείχνει τους πόσους κόμβους περιέχει το array και το max_size δείχνει ποιος είναι ο μέγιστος αριθμός που μπορεί ναι χωρέσει. Αν το size φτάσει το max_size τότε αλλάζουμε το μέγεθος του πίνακα με realloc.

Υπάρχουν επίσης διάφορες άλλες μεταβλητές στο trie_node , οι οποίες βοηθούν συγκεκριμένες λειτουργίες του προγράμματος, οι οποίες θα αναλυθούνε περεταίρω , όπως η isHyperNode που δηλώνει αν ο κόμβος είναι υπέρ-κόμβος στην περιπτώση που το δέντρο είναι στατικό, και τα A_version και D_version που κρατάνε την πληροφορία για το versioning.

## Trie

Το trie είναι η κύρια δομή που χρισιμοποιήται . Η δομή περιέχει μόνο έναν κόμβο trie_node ο οποίος αποτελεί την ρίζα.

### Λειτουργίες του trie

•Trie_Insert(trie_node *our_node, char* n_gram, int version)

Κατά την εισαγωγή ψάχνουμε την πρώτη λέξη του ngram στα παιδιά της ρίζας (our_node), εκτελώντας binary search. Αν την βρούμε τότε ψάχνουμε την επόμενη λέξη στα παιδιά αυτού του κόμβου κτλ. Αν δεν την βρούμε τότε δημιουργούμε ένα νέο trie_node και το τοποθετούμε στην κατάλληλη θέση στο array , έτσι ώστε να είναι ταξινομημένο. Αν το array είναι άδειο (δηλαδή size = 0) τότε μπαίνει στην πρώτη θέση του πίνακα. Επειδή χρησιμοποιούμε πολυνηματισμό και πρέπει να ξέρουμε πότε εισάχθηκε ο κάθε κόμβος, στο trie_node που θα δημιουργηθεί θα το αρχικοποιήσουμε με το version στην μεταβλητή A_version του trie_node.

•void Trie_Find(trie_node *our_node, int index, Str_Cntr *str, Str_Cntr *return_val, BloomFilter *bf, String_Array* my_array, int thread_id, int version)

Στην αναζήτηση για δυναμικό δέντρο χρησιμοποιούμε την Trie_Find. Παίρνουμε το ngram σε ένα array από λέξεις(String_Array* my_array), αυτό το κάνουμε γιατί κάθε φορά ελέγχουμε την κάθε λέξη με τις επόμενες της αν αποτελούν ngram. Δηλαδή αρχικά θα ελέγξουμε την πρώτη λέξη αν υπάρχει, αν ναι τότε θα δούμε στα παιδιά του κόμβου αν υπάρχει η δεύτερη, αν υπάρχει και

αυτή θα κοιτάξουμε για την τρίτη, αλλά αν όμως η τρίτη δεν βρεθεί θα πρέπει να γυρίσουμε πάλι πίσω στην δεύτερη, για αυτό είναι σημαντικό να έχουμε αποθηκευμένες τις λέξεις. Το index μας δείχνει από ποια λέξη αρχίζουμε και κοιτάμε.

Στην περίπτωση που βρεθεί ίδια λέξη τότε την αποθηκεύουμε στο Str_Cntr *str, έτσι ώστε όταν βρεθεί τελική λέξη να έχουμε ολόκληρο το ngram αποθηκευμένο και να το αντιγράψουμε στο Str_Cntr *return_val.

Κάθε φορά που βρίσκουμε κάποια λέξη ελέγχουμε αν είναι τελική (isFinal=true) , και στην περίπτωση του πολυνηματισμού αν το A_version και D_version είναι συμβατά με το version που είναι το νήμα. Αν ικανοποιεί όλες τις συνθήκες τότε το περνάμε μέσα από το bloom Filter το οποίο ελέγχει αν έχουμε ξαναβρεί αυτό το ngram μέσα στην ίδια αναζήτηση, αν δεν το έχουμε ξαναβρεί τότε το αποθηκεύουμε στην τελική δομή που θα επιστραφεί (Str_Cntr *return_val), η οποία θα περιέχει όλα τα ngram που βρέθηκαν.

Κάθε φορά που βρίσκουμε ένα ngram καλούμε την insert_ArrayCntr όπου το εισάγει στις δομές που έχουμε για τον υπολογισμό του top-k (εδώ χρειάζεται το thread_id).

•void StaticTrie_Find(trie_node *our_node, int i, Str_Cntr *str, Str_Cntr *return_val, BloomFilter *bf, String_Array* array, int thread_id, int version)

Για την αναζήτηση σε Στατικό δέντρο χρησιμοποιούμε την StaticTrie_Find. Σε αυτήν κάθε φορά πριν εξετάσουμε τον κάθε κόμβο ελέγχουμε αν είναι hyperNode κοιτώντας την bool μεταβλητή isHyperNode, ωστόσο περιέχει την πρώτη του λέξη στο word_1 ή word_2 (αναλόγως αν είναι στατικά ή δυναμικά αποθηκευμένη) έτσι ώστε να χρησιμοποιηθεί αυτή σε περίπτωση που ελεγχθεί από τον πατρικό της κόμβο. Αν είναι τότε δημιουργούμε την λέξη με βάση το hyperNode_array που δείχνει πόσους χαρακτήρες έχει η κάθε λέξη, και με βάση το hyperNode_word όπου περιέχει τους χαρακτήρες όλων των λέξεων. Στην συνέχεια συγκρίνουμε την λέξη με την λέξη του ngram και αν είναι ίδια εκτελούμε τις ίδιες ενέργειες που κάναμε και στην Trie_Find, με την διαφορά ότι για τον έλεγχο για το αν η λέξη είναι final κοιτάμε τον αριθμό που περιέχει τον αριθμό των χαρακτήρων της λέξης μας στο hyperNode_array. Αν είναι αρνητικός σημαίνει πως η λέξη μας είναι τελική.

•bool Trie_Delete_Versioning(trie_node *ournode, char* n_gram, int version)

Η Trie_Delete_Versioning εκτελείται κάθε φορά που έρχεται ένα D ngram στην ριπή και απλά βρίσκει το ngram στο trie και του αλλάζει το D_version .Επιστρέφει true αν πρέπει να αλλάξει το D_version της ρίζας.

•bool Trie_Delete(trie_node *ournode, char* n_gram)

Η Trie_Delete εκτελείται στο τέλος κάθε ριπής και διαγράφει τους κόμβους, αν βρεθούν και αν πρέπει. Αυτό γίνεται με την χρήση μιας στοίβας στην οποία αποθηκεύουμε τους κόμβους που βρίσκουμε στην πορεία αναζήτησης του ngram. Αν βρεθεί το ngram τότε αδειάζουμε την στοίβα και διαγράφουμε τους κόμβους αν ικανοποιούν κάποια κριτήρια, για παράδειγμα αν δεν έχουν παιδιά και αν δεν είναι ο τελικός κόμβος που ζητήθηκε να διαγραφτεί. Αν βρεθεί κόμβος που δεν πρέπει να διαγραφτεί τότε σταματάμε τις διαγραφές και καταστρέφουμε την στοίβα. Επιστρέφει true αν πρέπει να διαγραφτεί η ρίζα.

•void Trie_Destroy(trie_node* node)

Καταστρέφει αναδρομικά το trie που έχει για ρίζα το trie_node.

# Linear Hashtable

Το Linear Hashtable αποτελεί την ρίζα της δομής μας. Η δομή του hashtable αποτελείται από ένα array από buckets (και όχι από δείκτες σε buckets, έτσι ώστε να επιτύχουμε καλύτερο locality) μεγέθους M, το οποίο είναι defined και έχει επιλεχθεί ως μια δύναμη του 2 έτσι ώστε να γίνεται με λογικές πράξεις ο υπολογισμός του mod και του pow. Το κάθε bucket περιέχει έναν πίνακα από trie_node όπου εκεί αποθηκεύονται trie_node, όπου αυτά τα μπορεί να αποτελέσουν ρίζα σε trie. Το array του bucket αρχικοποιείται με μέγεθος C το οποίο είναι defined στο αρχείο trie_node.h.

## Μεταβλητές του hashtable

• int standart_size: το οποίο απεικονίζει το αρχικό μέγεθος του hashtable (M) και αυτό που θα καθορίζει και σε όλη διάρκεια τις ζωής του τον έλεγχο του round και τον υπολογισμό των hash func.

• int Hcurrent_size: το οποίο απεικονίζει το μέγεθος που έχει το hashtable εκείνη την στιγμή.

• Int standart_bucket_size: Περιεχέι το σταθερό μέγεθος (C) που είχε κάθε bucket από την δημιουργία του hashtable και είναι αναγκαίο για την δημιούργημα overflow bucket.

• int p: ο δείκτης που δείχνει ποιο bucket θα γίνει split.

• int round: Η τιμή του round.

• int finals: ποσά final υπάρχουν μέσα στο δένδρο.

• Bucket* table: δείκτης σε πινάκα με τα bucket.

### Μεταβλητές του Bucket

• int Bcurrent_size: το μέγεθος που έχει το αντίστοιχο bucket.

• int max_size: το μέγιστο μέγεθος που μπορεί να φτάσει χωρίς να χρειαστεί να γίνει overflow

• trie_node* children: δείχτης σε πινάκα που περιεχέι όλα τα trie_nodes στο αντίστοιχο bucket.

## Λειτουργίες του hashtable

•Hashtable* createHashTable();

Δημιουργεί ένα hashtable , κάνει τις απαραίτητες αρχικοποιήσεις και επιστρέφει τον δείκτη του.

• bool destroyHashTable(Hashtable* ht)

Σειριακά βρίσκει κάθε κόμβο στα bucket, διαγράφει το κάθε υποδένδρο trie που μπορεί να έχει, έπειτα ελευθερώνει τον χώρο για κάθε bucket και τέλος τον χώρο του hashtable.

•void insertHashTable( Hashtable* ht, char* ngram, int version)

Περνώντας σαν όρισμα ένα ngram, το σπάει και πάει να εισάγει την πρώτη λέξη του στο hashtable. Η λέξη περνάει μέσα από την συνάρτηση hashCode() η οποία επιστρέφει έναν ακέραιο θετικό αριθμό, μέσο αυτού υπολογίζεται σε ποιο bucket ανήκει η λέξη. Έπειτα μεσώ της binary_search βρίσκουμε σε ποια θέση η λέξη υπάρχει. Αν δεν υπάρχει σε εκείνο το σημείο δημιουργεί ένα καινούργιο node και την εισάγει στο σωστό σημείο. Έπειτα αν υπάρχουν και άλλες λέξεις στο ngram καλείτε η Trie_Insert() για την εισαγωγή του υπολοίπου ngram, αλλιώς ο κόμβος ορίζεται ως final και του δίνεται το version στο A_version.

Αν το bucket που είναι να εισαχθεί είναι γεμάτο τότε γίνεται overflow επεκτείνοντας το μέγεθος του κατά C με realloc() και έπειτα γίνεται η εισαγωγή του. Μέτα από αυτό πραγματοποιείται η διαδικασία του SPLIT. Για να εκτελεστεί το split, αντιγράφουμε όλο το bucket που είναι να γίνει split (αυτό που δείχνει το p) σε ένα προσωρινό bucket, αδειάζουμε αυτό με το p και γίνεται ξανά η διαδικασία εισαγωγής των λέξεων αφού έχουμε επεκτείνει το πλίθος των buckets.

• bool deleteHashTable(Hashtable* ht, char* ngram)

Παίρνει την πρώτη λέξη από το ngram και βρίσκει την θέση της μέσα στο hashtable. Αν η λέξη δεν είναι ίδια με αυτήν στην αντίστοιχη θέση τότε η λέξη δεν υπάρχει και επιστρέφει false. Αλλιώς καλεί την Trie_Delete() για την διαγράφει του υπολοίπου ngram. Αν αυτή επιστρέψει true τότε σημαίνει ότι και η λέξη στο bucket πρέπει να διαγραφεί και να διορθωθεί το bucket. Τέλος επιστρέφει true.

•bool deleteHashTable_Versioning(Hashtable* ht, char* ngram, int version)

Κάνει το ίδιο πράγμα με την deleteHashTable αλλά καλεί την Trie_Delete_Versioning και αν επιστρέψει true τότε δεν σβήνει τον κόμβο αλλά αλλάζει το D_version του κόμβου.

• trie_node* lookupTrieNode(Hashtable* ht, char *word)

Παίρνει μια λέξη και βρίσκει σε ποια θέση θα είναι μέσα στο hashtable. Άμα η λέξη υπάρχει επιστέφει έναν δείχτη στο node με αυτή την λέξη αλλιώς επιστρέφει NULL. Για τον υπολογισμό της δύναμης του 2 χρισιμοποιήται shift και για το mod λογικές πράξεις (μιας και έχουμε πάρει τους αριθμούς τέτοιους έτσι ώστε πάντα να είναι δύναμη του 2).

•char* FindHashTable(void* arguments, int thread_id)

Παίρνει όρισμα σε void* γιατί καλείται μέσα από thread και θέλαμε ο Job scheduler που από τον οποίο καλείται η συνάρτηση να είναι ανεξάρτητος και προσαρμόσιμος με κάθε συνάρτηση. Το argument περιέχει πληροφορίες όπως το ngram που θα αναζητηθεί, ένα bool για αν το δέντρο είναι στατικό, και το version που βρισκόμαστε.

Στην αρχή σπάει το ngram σε λέξεις με την str_split και ξεκινάει την αναζήτηση. Καλεί την lookupTrieNode όπου επιστρέφει δείκτη σε trie_node, αν αυτό είναι final και είναι συμβατό με το version μας τότε το περνάμε μέσω του bloom filter και αν επιστρέψει πως δεν έχει ξαναϊδωθεί τότε

το εισάγουμε στο string που θα επιστραφεί. Στην συνέχεια αναλόγως αν το δέντρο είναι στατικό η δυναμικό καλούμε την StaticTrie_Find ή την Trie_Find αντίστοιχα. Η παραπάνω διαδικασία εκτελείται για την κάθε λέξη του ngram.

Κάθε φορά που βρίσκεται final ngram τότε το περνάμε από την insert_ArrayCntr όπου εκτελεί το κομμάτι του top-k (εδώ χρειάζεται το thread_id και επειδή το argument δημιουργήθηκε στη φάση του submit_job δεν γινόταν να του το εντάξουμε).

# JobScheduler

Ο jobscheduler είναι η δομή που ρυθμίζει και ελέγχει τα threads. Χρησιμοποιεί μια ουρά την οποία την γεμίζει με jobs , τα οποία είναι μια δομή που περιέχουν μια generic ορισμένη συνάρτηση και την προτεραιότητα της.

## Περιγραφή της διαδικασίας

Αρχικά δημιουργείται ο jobscheduler με την χρήση της initialize_scheduler. Στην συνέχεια κάθε φορά που έρχεται ένα Q η main δημιουργεί ένα Argument με τα απαραίτητα δεδομένα που πρέπει να σταλθούν στην FindHashtable και έπειτα δημιουργεί ένα job με την create_Job όπου του δίνει την συνάρτηση FindHashtable, το argument και την ανάλογη προτεραιότητα που θα έχει, και στο τέλος εκτελεί submit_job όπου εκεί το job εντάσσεται στην ουρά του jobscheduler.

Στην συνέχεια όταν τελειώσει η ριπή και του έρθει F ,η main εκτελεί execute_all_jobs όπου ξυπνάει όλα τα threads και αυτά ξεκινούν να παίρνουν δουλείες και να τις εκτελούν, επίσης εκτελεί wait_all_tasks_finish όπου εδώ το master thread παγώνει μέχρι να τελειώσουν την δουλεία όλα τα thread. Εντωμεταξύ πριν εκτελεστεί η execute_all_jobs όλα τα threads είχαν παγώσει μέσα στην jobRoutine, έπειτα αφού εκτελεστεί ξυπνάει ένα thread παίρνει μια δουλεία , ξυπνάει το επόμενο και εκτελεί την δουλεία του. Αυτό θα συμβαίνει μέχρι να αδειάσει η ουρά. Όταν κάποιο thread τελειώσει την δουλεία του, τότε πάει και γράφει στο JS->results->array στην θέση της προτεραιότητας της δουλειάς που είχε αυτό που του επέστρεψε η δουλεία του.

Όταν η ουρά αδειάσει και ο αριθμός των δουλειών που εκτελέστηκαν ισούται με τον αριθμό των δουλειών που του ζητηθήκαν να εκτελεστούν, τότε το thread που το διαπίστωσε αυτό ξυπνάει το master thread που περίμενε στην wait_all_tasks_finish , και όλα τα thread ξανά αδρανοποιούνται μέχρι να τα αφυπνίσει πάλι το master thread με την execute_all_jobs. Στην συνέχεια το master thread εκτυπώνει μια μια τις θέσεις του JS->results->array όπου περιέχουν τα αποτελέσματα των Q.

Όταν εκτελεστεί η destroy_JobScheduler τότε ξυπνάνε τα threads και βλέπουν πως JS->final_task=true , οπότε κάνουν pthread_exit και έτσι καταστρέφονται.

### Versioning

Επειδή τα Q εκτελούνται στο τέλος της ριπής , κάθε φορά στις find το νήμα θα πρέπει να ξέρει ποιοι κόμβοι προστέθηκαν μετά ή διαγράφτηκαν πριν την στιγμή που έπρεπε να εκτελεστεί το Q . Οπότε για αυτό σε κάθε trie_node διατηρούμε το A_version και το D_version τα οποία

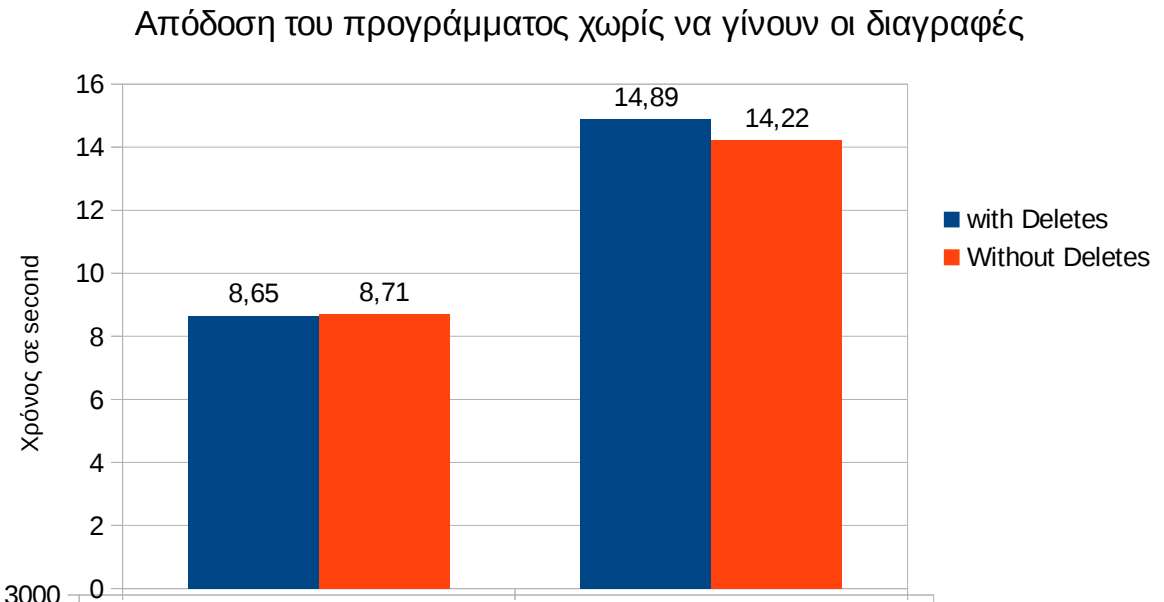
δηλώνουν πότε προστέθηκε και πότε διαγράφτηκε ο κόμβος αντίστοιχα. Το A_version αρχικοποιείται με 0 και το D_version αρχικοποιείται με -1, και κάθε φορά που προσθέτεται νέος κόμβος το A_version του δίνεται η τιμή του current_version στην οποία υπάρχει αυτή τη στιγμή η δομή, και κάθε φορά που εκτελείται η deleteHashTable_Versioning δίνει στον κόμβο που πρέπει να διαγραφτεί την τιμή του current_version στον D_version.

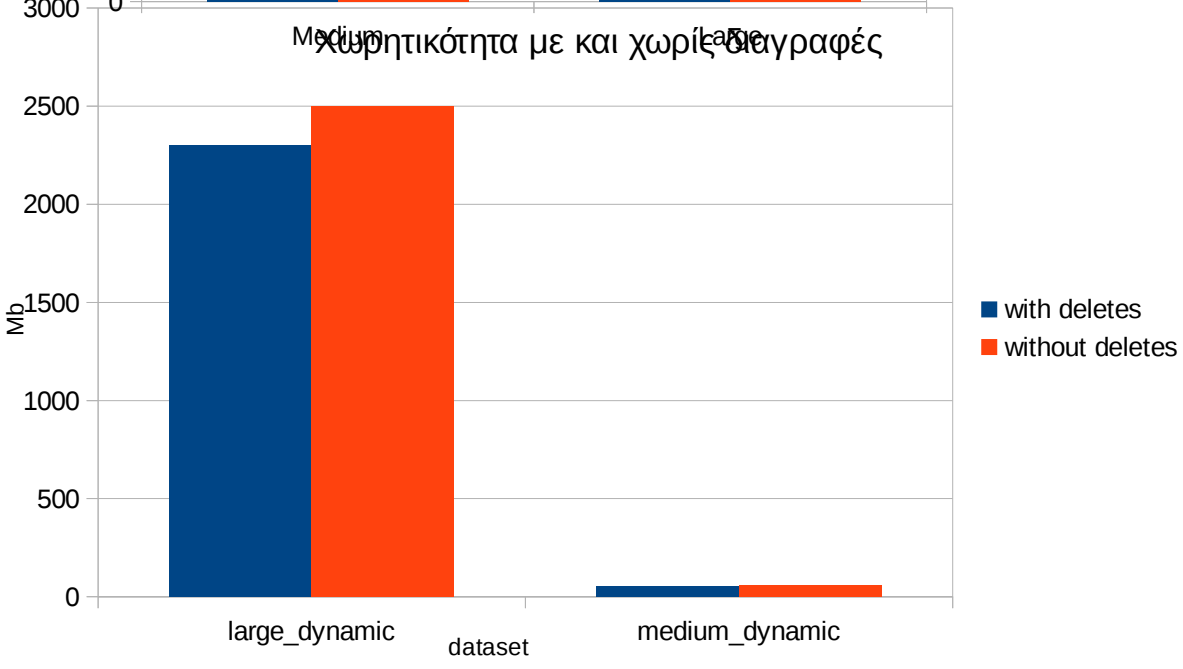
Οπότε κάθε φορά που ένα νήμα εκτελεί ένα Q ξέρει για ποιο version της δομής αναζητάει ngrams , έτσι κάθε φορά που βρίσκει ngram για να το συμπεριλάβει θα πρέπει να ικανοποιεί την εξής συνθήκη

“our_node->isFinal && (our_node->D_version > version || our_node->D_version==-1 ) && our_node->A_version <= version”

όπου το our_node είναι ένα trie_node

Παρόλα αυτά η deleteHashTable_Versioning δεν διεγράφη τους κόμβους. Αυτό επηρεάζει την λειτουργία των νημάτων μιάς και το εύρος αναζήτησης των νημάτων είναι μεγαλύτερο από τι θα έπρεπε να είναι, και ειδικά στο large dataset όπου γίνονται πολλές διαγραφές. Για αυτό κάθε φορά που έρχεται ένα D βάζουμε το ngram σε μια ουρά και στο τέλος κάθε ριπής παίρνουμε έναν έναν τους κόμβους της ουράς και καλούμε την deleteHashTable η οποία τους διαγράφει. Ωστόσο η διαγραφή των κόμβων κοστίζουν σε χρόνο, μιας και θα πρέπει να εξερευνήσει πάλι ολόκληρη τη δομή μέχρι να βρεθεί ο κόμβος που θα διαγραφτεί, άλλα συμφέρει σε χώρο μιας και μειώνεται το μέγεθος τις δομής



(medium_dynamic με διαγραφές : 53.1MB, χωρίς :57.1MB , αναλυτικότερα στο τέλος)

### Μεταβλητές του Job

•void* (*job)(void* , int): Η δουλεία που θα εκτελέσει, το int είναι το thread_id κατά την δημιουργία του job για αυτό δεν τοχω συμπεριλάβει μέσ'το void*.

•void* argument: Αυτό που θα του δοθεί στο σαν πρώτο όρισμα, περιέχει διάφορα δεδομένα για την FindHashtable .

•int priority : Ο αριθμός προτεραιότητας που έχει η δουλειά , επίσης υποδηλώνει την θέση του πίνακα όπου θα γραφτεί το αποτέλεσμα της συνάρτησης.

•Job* next : Το επόμενο στοιχείο της ουράς.

### Λειτουργίες του Job

•Job* create_Job(void* (*f)(void* , int), void* arg, int priority)
Δημιουργεί ένα job.

•Job* create_AnyJob(void* arg)
Επειδή η ουρά χρισιμοποιήται και για την διαγραφή και σε άλλα σημεία του προγράμματος , δημιουργεί job με σκοπό του οποίου το arg μπορεί να είναι οτιδήποτε.

### Μεταβλητές του Queue

•Job* first και Job* last : Ο πρώτος και ο τελευταίος κόμβος της ουράς.

•int size : Το μέγεθος τις ουράς

•int no_jobs: ο αριθμός τον δουλειών που του έχει ζητηθεί να εκτελεστούν.

### Λειτουργίες του Queue:

•Queue* createQueue(): Δημιουργεί και επιστρέφει έναν δείκτη σε Queue.

•bool emptyQueue(Queue* q) : Επιστρέφει true ή false αναλόγως αν η ουρά είναι άδεια

•void insertQueue(Queue* q, Job *node): Εισάγει ένα job στην ουρά.

•Job* pop(Queue* q): Αφαιρεί ένα job από την ουρά

•void destroyQueue(Queue* q) : Καταστρέφει την ουρά.

## Μεταβλητές του Jobscheduler

•int execution_threads : Το πόσα threads θα δημιουργηθούν.

•Queue* q: Η ουρά.

•pthread_t* tids: Ο πίνακας με τα id των threads.

•String_Array results : Η δομή στης οποίας τον πίνακα θα γράφουν τα αποτελέσματα των FindHashtable.

•int self : Το κάθε thread έχει διαφορετική τιμή σε αυτό το πεδίο , περιέχει έναν αριθμό από το 0 μέχρι το execution_threads.

•bool occupied: Είναι true αν κάποιος διαβάζει η γράφει στην ουρά.

•int executed: Πόσες δουλείες έχουν εκτελεστεί.

•bool final_task : Είναι true αν πρέπει να κάνουν exit.

•int waitting: Πόσα thread είναι παγωμένα.

•pthread_cond_t c_read και pthread_mutex_t mtx_read :Παγώνει τα νήματα όταν πάνε να διαβάσουν/γράψουν από/στην την ουρά η στην occupied.

•pthread_cond_t c_write καιpthread_mutex_t mtx_write : Παγώνει τα νήματα όταν πάνε να διαβάσουν/γράψουν την μεταβλητή executed.

•pthread_cond_t c_master καιpthread_mutex_t mtx_master: Παγώνει και ξυπνάει το master thread όταν τα threads έχουν ολοκληρώσει την δουλεία τους.

## Λειτουργίες του Jobscheduler

•JobScheduler* initialize_scheduler(int execution_threads, String_Array *result):
Δημιουργεί τον JobScheduler, αρχικοποιεί τα mutexes και τα conditional varaibles ,δημιουργεί και εκκινεί τα threads και επιστρέφει τον δείκτη.

•void submit_job(JobScheduler* js , Job* job):
Τοποθετεί το job μέσα στην ουρά του JobScheduler.

•void execute_all_jobs(JobScheduler* JS):
Εκκινεί όλα τα threads για να ξεκινήσουν την δουλεία τους.

•void wait_all_tasks_finish(JobScheduler* JS):

Παγώνει και περιμένει σήμα  απ'όταν τα thread τελειώσουν την δουλειά τους. Επίσης κάνει το occupied=true έτσι ώστε τα threads να παγώσουν σίγουρα.

•void* JobRutine(void* js):

Όταν τα νήματα εκκεινούνται μπαίνουν σε αυτήν την διαδικασία . Όταν μπαίνουν παγώνουν γιατί η ουρά είναι άδεια, αλλά όταν ξυπνήσουν τότε ένα ένα παίρνει την μια δουλεία από την ουρά και την εκτελεί , όταν την τελειώσει τότε γράφει στο JS->results->array το αποτέλεσμα της δουλείας του και αυξάνει το executed. Άμα η ουρά αδειάσει και το executed είναι ίσο με το JS->q->no_jobs τότε αυτό σημαίνει πως τα νήματα έχουν τελειώσει την δουλεία τους και ξυπνάνε το master thread και με την σειρά τους αυτά πάνε για ύπνο.

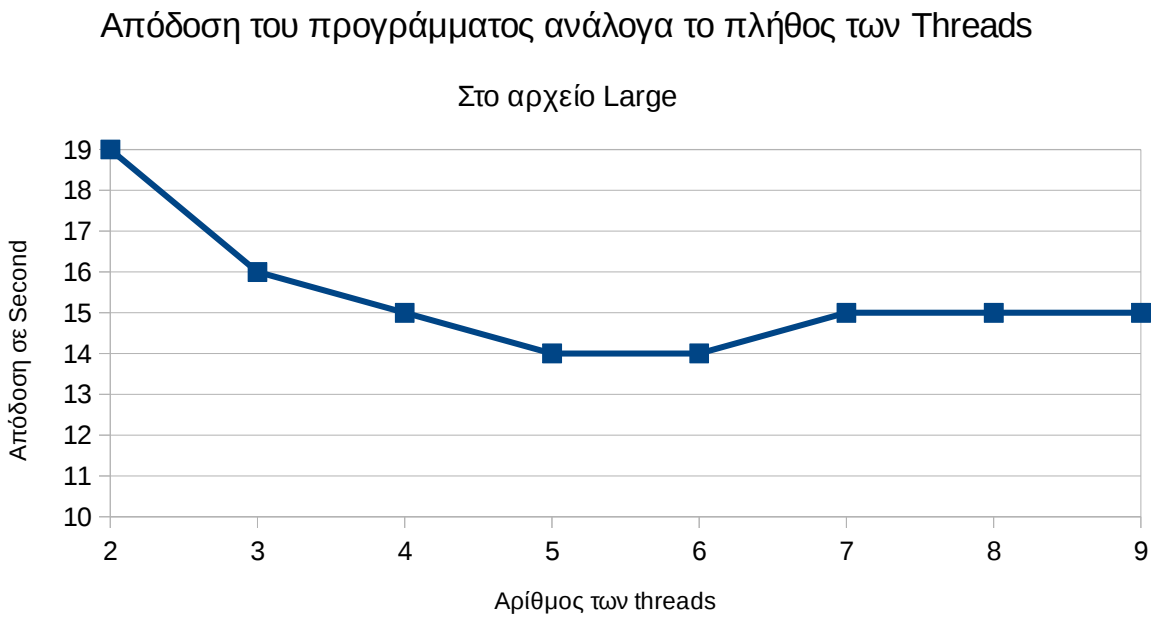
Στην περίπτωση που ξυπνήσουν και το JS->final_task είναι true τότε σημαίνει πως ο jobscheduler καταστρέφεται οπότε  κάνουν exit.

•void  destroy_JobScheduler(JobScheduler* JS):

Θέτει το JS->final_task σε true και ξυπνάει τα threads όπου με την σειρά τους αυτά τερματίζουν και στην συνέχεια αποδεσμεύει ότι χρειάζεται.

## Βελτίωση απόδοσης με τη χρήση threads

Με την χρήση της παραλληλίας ο χρόνος εκτέλεσης των νημάτων μειώθηκε δραματικά , ωστόσο παρατηρήσαμε πως δεν σημαίνει πώς όσα περισσότερα νήματα έχουμε τόσο μεγαλύτερη θα είναι και η μείωση στον χρόνο. Μετά από ένα σημείο τα threads δημιουργούσαν συμφόρηση στους πυρήνες και ο χρόνο αυξήθηκε αντί να μειωθεί. Βέβαια αυτό έχει να κάνει και με το πόσους πυρήνες διαθέτει ο επεξεργαστείς και με τα πόσα νήματα μπορεί να εξυπηρετήσει.
Η σχέση χρόνου και threads φαίνεται αναλυτικότερα στο παρακάτω διάγραμμα:



# Δευτερεύουσες Δομές της Εργασίας

## Bloom Filter

Το Bloom Filter χρισιμοποιήται από την FindHashtable ,Trie_Find και την StaticTrie_Find για την αποφυγή διπλότυπων. Το κάθε νήμα δημιουργεί δικό του Bloom Filter .

Αρχικά για την δημιουργία του bloom filter καλείται η BF_Init η οποία δέχεται σαν όρισμα έναν αριθμό n, το οποίο δηλώνει πόσα ngram υπάρχουν στην δομή μας. Με βάση αυτό τον αριθμό και μιας ενδεικτικής πιθανότητας P ,για το επιθυμητό positive false η οποία είναι δηλωμένη ως defined στο trie_node.h , υπολογίζουμε τις τιμές του m (μέγεθος του bit vector ) και k (αριθμός hashfunctions) χρησιμοποιώντας τους παρακάτω τύπους :

$$m = -\frac{n \ln p}{(\ln 2)^2} \qquad\qquad k = \frac{m}{n} \ln 2.$$

Ωστόσο για n των αριθμό των finals που υπάρχουν στην δομή μας στο mdeium και large dataset οι τύποι μας δίναν υπερβολικά μεγάλο m, για αυτό με διάφορα test βρήκαμε κάποιες τιμές όπου για τέτοιο n το medium και το large dataset έβγαζαν σωστά αποτελέσματα χωρίς να παράγουν εξωφρενικά μεγάλο m και μεγάλο k, όπου θα αύξανε τον χώρο και τον χρόνο εκτέλεσης. Επίσης απορρίψαμε το ενδεχόμενο να το κάνουμε με σταθερό m , παρόλο που βρήκαμε τιμή η οποία ικανοποιούσε όλα τα datasets, γιατί στο small δέσμευε περισσότερο χώρο απ'όσο πραγματικά χρειαζόταν , και σε περίπτωση μελλοντικών dataset να μην χρειάζεται να αλλάξουμε την τιμή του, αντ'αυτού το πρόγραμμα να μπορέσει να υπολογίσει μόνο του την κατάλληλη τιμή.

Ο πίνακας που χρησιμοποιούμε είναι ένα char array, μιας και θέλαμε να περιορίσουμε όσο το δυνατόν περισσότερο τον μέγεθος του πίνακα σε bytes. Επίσης για hushfunction χρησιμοποιούμε την MurmurHash3, μιας και αυτή μας φάνηκε η πιο ιδανική από την έρευνα που κάναμε στο διαδίκτυο. Μιας και χρειάζονται k hashfunction καλούμε την MurmurHash3 με seed, το οποίο κάνει κάποια αρχικοποίηση, από 0 έως k-2 και για τελευταία το άθροισμα των δύο τελευταίων (για να μειώσουμε τον χρόνο).

Για να ελεγχθεί αν ένα ngram υπάρχει στο bloom Filter καλείται η BF_Lookup η οποία καλεί τις hushfunctions για το ngram και ελέγχει τις τιμές του πίνακα. Άμα βρει έστω και ένα 0 τότε το κάνει 1 και επιστρέφει πως δεν βρέθηκε, αλλιώς επιστρέφει πως βρέθηκε. Έτσι η BF_Lookup κάνει ταυτόχρονα και αναζήτηση και εισαγωγή.

Το Bloom Filter δημιουργείται στην αρχή της FindHashTable και καταστρέφεται στο τέλος της.

Πριν την χρήση του bloom Filter για να ελέγχουμε τα διπλότυπα είχαμε κατασκευάσει έναν ταξινομημένο πίνακα , όπου κάθε φορά που ερχόταν ένα ngram έψαχνε με binary search στον πίνακα και άμα το έβρισκε το απόρριπτε, αλλιώς το έβαζε στον πίνακα στην κατάλληλη θέση .

Στο part 2 (δηλαδή με lienar hash και top-k) είχαμε υπολογίσει την διαφορά χρόνου με μεταξύ με χρήση bloom filter και sorted array στο medium_dynamic dataset. Με bloom Filter έκανε μέσω χρόνο 36,86 δευτερόλεπτα και με sorted array 40,03 δευτερόλεπτα. Ωστόσο μετά την

υποβολή του part 2 κάναμε διάφορες βελτιστοποιήσεις οι οποίες μείωσαν τον χρόνο του medium dataset σε λιγότερο από 30s.

# Top-k

Για τον υπολογισμό του top-k χρησιμοποιήσαμε διαφορετική παραδοχή από αυτήν που είχαμε υλοποίηση στο part2, επειδή δεν θα ήταν αποτελεσματική χρησιμοποιώντας πολυνηματισμό.

## Περιγραφή της διαδικασίας

Το top-k υπολογίζεται με την χρήση ταξινομημένων πινάκων και έναν σωρό ελαχίστου(min heap). Αρχικά δημιουργούνται τόσοι πίνακες όσα και τα threads , για αυτό το κάθε νήμα έχει ένα thread_id για το οποίο ισχύει 0 <= thread_id < execution_threads, και το κάθε νήμα έχει τον δικό του πίνακα στον οποίο γράφει. Κάθε φορά που βρίσκεται ένα τελικό ngram το οποίο έχει περάσει και από τον έλεγχο του bloom filter , εκτελείται η insert_ArrayCntr η οποία έχει ως arguments το ngram και το thread_id. Αυτή η συνάρτηση αρχικά κάνει binary serach στον πίνακα για να βρει αν υπάρχει ήδη αλλιώς να βρει την θέση που θα πρέπει να τοποθετηθεί , έτσι ώστε ο πίνακας να είναι ταξινομημένος. Άμα βρεθεί η λέξη του αυξάνει τον counter, αλλιώς την εισάγει στην σωστή θέση.

Όταν εκτελεστούν όλα τα Q θα υπάρχουν execution_threads ταξινομημένοι πίνακες όπου θα περιέχουν ngrams με το counter τους. Οπότε αρχικά πρέπει να γίνουν merge οι πίνακες. Αυτό γίνεται με την void Merge_Arrays , η οποία ξεκινάει και έχει δείκτες για τους πίνακες αρχικοποιημένους με 0. Αρχικά θεωρεί το ngram του πρώτου πίνακα το μικρότερο και το κρατάει, μαζί με τον δείκτη της θέσης του πίνακα. Στην συνέχεια ψάχνει για μικρότερα ή ίσα. Αν βρει ίσο, τότε του προσθέτει το counter σε μία μεταβλητή που χρησιμοποιούμε, και κρατάει και αυτού τον δείκτη. Αν βρει μικρότερο τότε κρατάμε αυτό το ngram , αποθηκεύουμε τον counter του και σβήνουμε τους υπόλοιπους δείκτες των πινάκων που είχαμε κρατήσει και κρατάμε αυτουνού, οπότε στην συνέχεια ψάχνουμε να βρούμε μικρότερα ή ίσα με αυτό το ngram. Στο τέλος της αναζήτησης θα έχουμε αποθηκευμένο το μικρότερο ngram από τις πρώτες θέσεις του πίνακα, τον counter του, και τις θέσεις των πινάκων όπου αυτό εμφανίστηκε. Οπότε το αποθηκεύουμε στο τελικό array στην θέση που πρέπει, και αυξάνουμε τους δείκτες των πινάκων στους οποίους βρέθηκε αυτό το ngram. Στην συνέχεια πάμε και ψάχνουμε το επόμενη μικρότερο ngram στους δείκτες του κάθε πίνακα. Η διαδικασία σταματάει όταν ελέγξουμε όλα τα ngrams των πινάκων . Στο τέλος θα έχουμε έναν πίνακα ο οποίος θα περιέχει τα όλα τα ngram και το πόσες φορές εμφανίστηκε το καθένα.

Οπότε τώρα θα πρέπει να βρούμε τα top-k σε αυτόν τον πίνακα. Αυτό επιτυγχάνεται με την χρήση ενός min heap τον οποίο τον δημιουργούμε από τις πρώτες k θέσεις του πίνακα. Ο min heap πάντα θα περιέχει το μικρότερο στοιχείο (με το μικρότερο counter ) στην ρίζα του, οπότε αυτό που κάνουμε είναι να συγκρίνουμε τα υπόλοιπα στοιχεία του πίνακα με την ρίζα του min heap , και κάθε φορά που θα βρίσκουμε στοιχείο με μεγαλύτερο counter (ή με ίσο και μικρότερο αλφαριθμητικό) θα το εναλλάσσουμε με την ρίζα και θα καλούμε την heapify ,η οποία συντηρεί τον σωρό. Έτσι όταν τελειώσουμε τις συγκρίσεις , ο σωρός μας θα περιέχει τα top-k με το μικρότερο στην ρίζα . Οπότε για να τα εκτυπώσουμε κάνουμε pop από την ρίζα και καλούμε την heapify , για να μεταφέρει το επόμενο μικρότερο στην θέση της ρίζας, και το ngram που κάναμε pop από τον σωρό το βάζουμε στο string. Όμως επειδή το string θέλουμε να περιέχει από το μεγαλύτερο προς το

μικρότερο και ο σωρός μας βγάζει από το μικρότερο στο μεγαλύτερο, κάθε φορά που κάνουμε pop το ngram που παίρνουμε το τοποθετούμε στην αρχή του string. Έτσι στο τέλος καταφέρνουμε να έχουμε τα top-k από το μεγαλύτερο στο μικρότερο.

Η πολυπλοκότητα ανεύρεσης των top-k με την χρήση ελάχιστου σωρού και πίνακα υπολογίζεται ότι είναι $O(k + (n-k)Logk + kLogk)$. Πήραμε την ιδέα από την παρακάτω ιστοσελίδα.

https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/

## Δομές για την υλοποίηση του top-k

•Word_Cntr : Περιέχει ένα Char στο οποίο αποθηκεύεται το ngram, και έναν counter ο οποίος υποδείχνει το πόσες φορές έχει εμφανιστεί το ngram.

•Array_Cntr : Είναι η δομή όπου το κάθε thread έχει . Περιέχει ένα πίνακα από Word_Cntr , το μέγεθος του και το μέγιστο μέγεθος που μπορεί να χωρέσει πριν χρειαστεί να γίνει realloc.

•MinHeap : Είναι ο σωρός που με βάσει αυτόν υπολογίζεται το top-k. Περιέχει έναν πίνακα από δείκτες σε Word_Cntr (Word_Cntr **heap) και το μέγιστο μέγεθος του.

## Λειτουργίες του top-k

•void init_TopK()
Κατασκευάζει και αρχικοποιεί τις δομές.

•void destroy_TopK()
Αποδευσμεύει τις δομές που χρησιμοποιήθηκαν.

•int binarySearch(Word_Cntr *array, int size, char *searchVal)
Εκτελεί binary Search σε έναν πίνακα τύπου Word_Cntr.

•bool insert_ArrayCntr(char *ngram, int index)
Εισάγει ένα ngram στον πίνακα Array_Cntr topk[index] , όπου το index είναι το thread_id του thread.

•void Merge_Arrays()
Εκχωρεί όλους τους πίνακες στον τελικό πίνακα final, εκτελώντας την διαδικασία που περιγράφτηκε παραπάνω.

•void calculate_TopK(int k)
Υπολογίζει και εκτυπώνει το top-k με την χρήση του πίνακα final και του min heap, όπως περιγράφτηκε παραπάνω.

•void clean_TopK()

Καθαρίζει τις δομές. Στο τέλος κάθε υπολογισμού του topk καλείται αυτή η συνάρτηση και καθαρίζει τις δομές έτσι ώστε να μην χρειάζεται να της επαναδημιουργούμε κάθε φορά.

•void heapify(Word_Cntr **minHeap, int i, int size)

Συντηρεί τον σωρό, δηλαδή τον διαμορφώνει έτσι ώστε να είναι το μικρότερο στην ρίζα.

## Σχεδιαστικές επιλογές

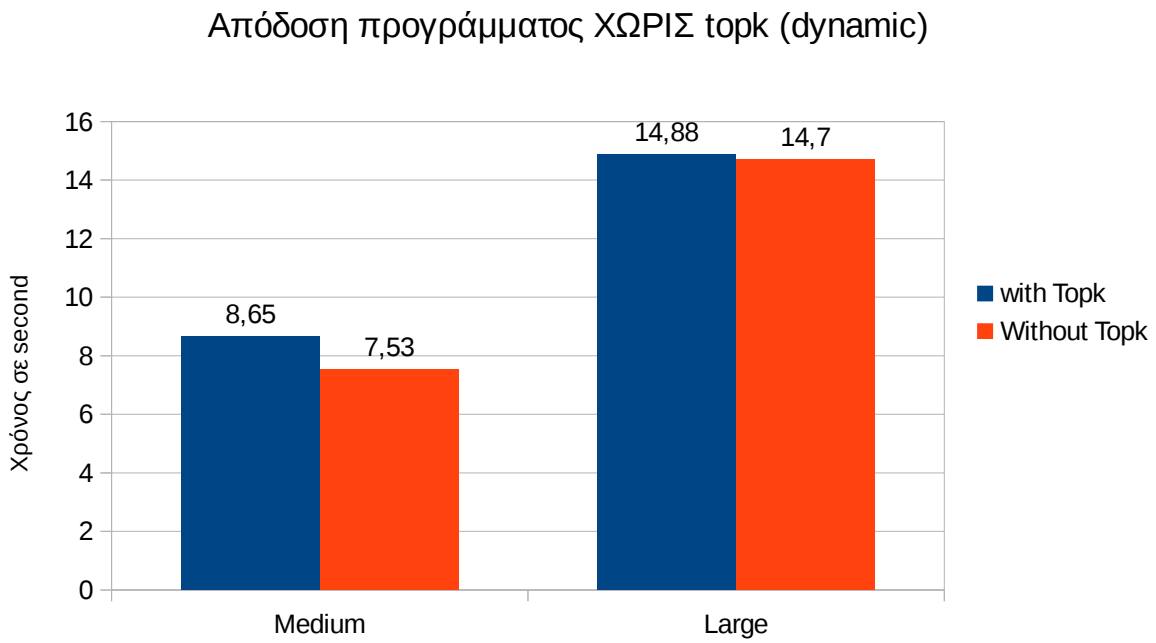
Αρχικά για την εισαγωγή του ngram σε πίνακα, δεν χρησιμοποιούσαμε binary search αλλά ένα hashtable του οποίου οι κόμβοι δείχνανε σε θέσεις στον πίνακα που θα έμπαινε το ngram. Η διαδικασία περιληπτικά ήταν η εξής : ερχόταν ένα ngram, το περνάγαμε από μια hashfunction η οποία μας έβγαζε μια θέση για το hashtable του οποίου η θέση ήταν ή κενή ή είχε μια λίστα θέσεων του πίνακα. Αν ήταν κενή έβαζε το ngram στην τελευταία θέση του πίνακα και έβαζε αυτή τη θέση στον πίνακα με θέσεις στον κόμβου του hashtable. Αλλιώς έλεγχε τις θέσεις και αν κάποια από αυτές περιείχε ίδιο ngram του αύξανε τον counter , αλλιώς το έβαζε στο τέλος του πίνακα και πρόσθετε και αυτή τη θέση στον πίνακα με θέσεις του κόμβου του hashtable.

Αυτός ο τρόπος μεν έκανε πιο γρήγορη εισαγωγή στον πίνακα αλλά μετά το merge τον πινάκων θα ήταν πιο περίπλοκο και πιο χρονοβόρο , μιας και οι πίνακες δεν θα ήταν ταξινομημένοι. Επίσης μίας και τρέχουν πολλά νήματα το κάθε νήμα θα έπρεπε να έχει δικό του hashtable , οπότε αυτό θα σπατάλαγε πολύ μνήμη.

Επίσης μια άλλη ιδέα που είχαμε ήταν να υπάρχει ένας πίνακας και ένα hashtable και το κάθε νήμα που θα πήγαινε να γράψει στον πίνακα ή στο hashtable θα πέρναγε από mutexes, έτσι κάθε φορά μόνο ένα thread θα έγραφε. Ωστόσο μεν αυτή η λύση μπορεί να ήταν η πιο αποτελεσματική μιας και θα γλιτώναμε το merge και το κάθε νήμα δεν θα είχε τον δικό του πίνακα, την απορρίψαμε γιατί θεωρήσαμε ότι δεν είναι σωστό να ελέγχεται η ροή τον νημάτων από άλλο σημείο του προγράμματος , εκτός του jobscheduler.

## Απόδοση top-k

Χάριν τού πολυνηματισμού, που κάνει τις σπάταλες σε χρόνο insert_ArrayCntr ταυτόχρονα, και τις καλές υλοποιήσεις τις εκ χώνευσης των πινάκων και του υπολογισμού του top-k, η διαδικασία παίρνει παίρνει πού λίγο χρόνο. Στο παρακάτω διάγραμμα φαίνεται η διαφορά των μέσων χρόνων που έκανε το πρόγραμμα υπολογίζοντας και μη το top-k.




## Static Trie

Η υλοποίηση του static trie γίνεται με την χρήση μερικών βοηθητικών πεδίων στο trie_node και με την χρήση των συναρτήσεων StaticTrie_Find , Compress ,CompressTrie, και convert_toHyperNode. Επιπλέον έχουμε υλοποιήσει και έναν 2ο τρόπο που εκτελείται με τις συναρτήσεις convert_toHyperNode , Compress2 και CompressTrie2.

### Πεδία του trie_node για το static trie

•bool isHyperNode : Είναι true αν ο κόμβος είναι υπέρ-κόμβος

•char* hyperNode_word : Περιέχει τους χαρακτήρες των λέξεων.

•int hyperNode_wordSize : Περιέχει το μέγεθος του hyperNode_word.

•int hyperNode_wordMax : Περιέχει το μέγιστο μέγεθος που μπορεί να φτάσει το hyperNode_word πριν χρειαστεί να γίνει realloc.

•int* hyperNode_array : Είναι ο πίνακας όπου περιέχει πόσους χαρακτήρες έχει η κάθε λέξη

•int hyperNode_size : Το μέγεθος του hyperNode_array.

•int hyperNode_max : Το μέγιστο μέγεθος που μπορεί να φτάσει το hyperNode_array πριν χρειαστεί να γίνει realloc

## Λειτουργίες για το static trie

•void convert_toHyperNode(trie_node* father, trie_node* son)

Παίρνει δύο κόμβους (τον son και τον father) , τους ενώνει και τους μετατρέπει σε έναν υπέρ-κόμβο. Αρχικά ελέγχει αν ο father είναι υπέρ-κόμβος.

Αν δεν είναι ο father υπέρ-κόμβος, τότε αντιγράφει την λέξη του father στο hyperNode_word και στην συνέχεια προσκολλάει την λέξη του son. Έπειτα δημιουργεί τον πίνακα με τους χαρακτήρες αναλόγως με το μέγεθος της κάθε λέξης , και αν κάποια από αυτές είναι τελικιά τότε του δίνει αρνητική τιμή. Στην συνέχεια ο father αντιγράφει τον δείκτη με τα παιδιά του son και ο son διαγράφεται. Εν τέλει ο father σημειώνεται ως υπέρ-κόμβος.

Αν ο father είναι υπέρ-κόμβος τότε προσκολλάει στο hyperNode_word την λέξη του son και προσθέτει στον πίνακα τον αριθμό με τους χαρακτήρες της λέξης του son (και το κάνει αρνητικό αν ο son είναι τελικός κόμβος) . Στην συνέχεια του αντιγράφει τον δείκτη με τα παιδιά του son και μετά ο son διαγράφεται.

•void CompressTrie(trie_node* node)

Παίρνει έναν κόμβο και ελέγχει πόσα παιδιά έχει. Αν έχει ένα παιδί τότε καλεί την convert_toHyperNode με ορίσματα αυτόν και το παιδί του και έπειτα ξανά καλεί την CompressNode με όρισμα το node, που πλέον έχει μετατραπεί σε υπέρ-κόμβο και έχει τα παιδιά που είχε το παιδί του. Αν έχει περισσότερα από ένα παιδιά, τότε για κάθε παιδί καλεί την CompressNode με όρισμα το παιδί.

•void Compress(Hashtable* ht)

Για κάθε κόμβο του hashtable σε κάθε bucket που έχει trie_node ελέγχει τα παιδιά του. Αν αυτός ο κόμβος έχει μόνο ένα παιδί τότε καλεί την convert_toHyperNode για αυτόν και το παιδί και μετά καλεί αναδρομικά πάλι την Compress. Αν έχει περισσότερα από ένα παιδιά τότε για το καθένα από αυτά καλεί την CompressNode με αυτό το παιδί.

Δεύτερη υλοποίηση του static trie.

Σε αυτή την υλοποίηση η δημιουργία των Hypernode δεν γίνεται ανά 2 κόμβοι αλλά έχουμε μια ουρά με δείκτες στους κόμβους οι οποίοι πρέπει να συμπιεστούν και τους κάνει όλους μαζί.

•void Compress2(Hashtable* ht)

Αυτή η συνάρτηση παίρνει για όρισμα το hashtable και για κάθε bucket του το οποίο δεν είναι άδειο, παίρνει ένα ένα τα παιδιά του και καλεί την CompressTrie2 για να συμπιεστεί το αντίστοιχο trie που έχει το κάθε ένα.

•void CompressTrie2(trie_node* node)

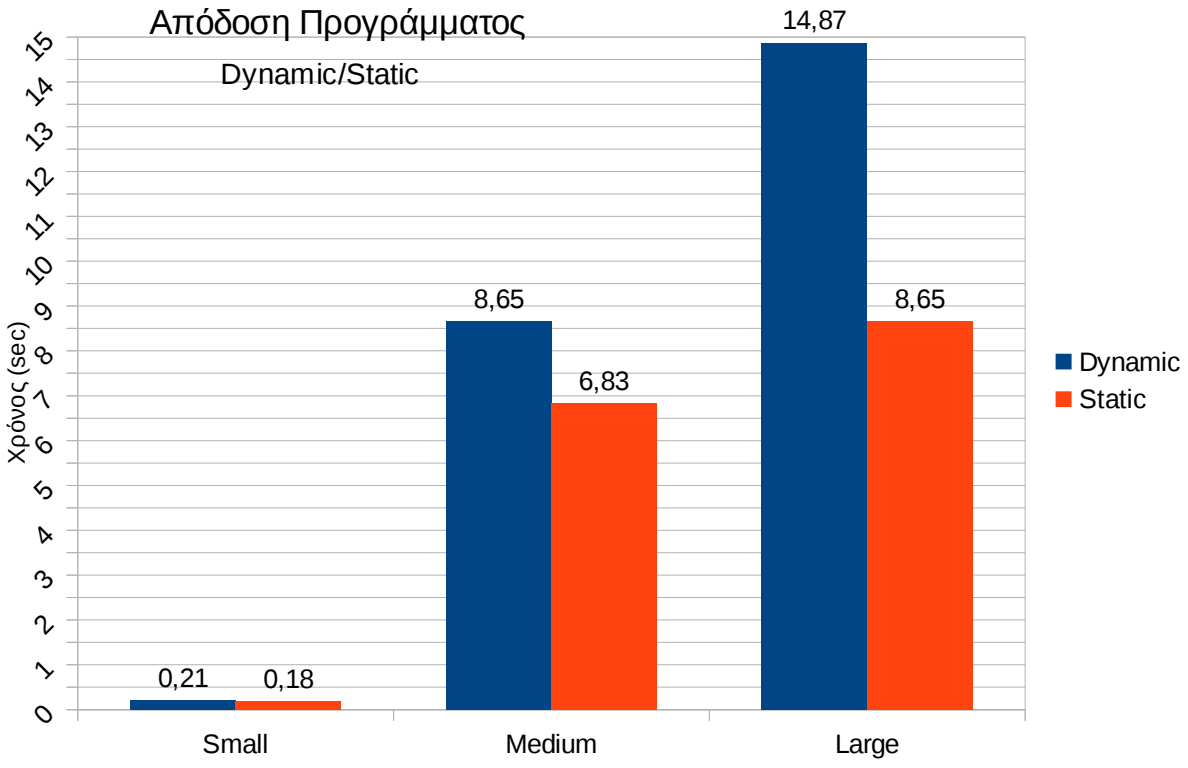
Αυτή η συνάρτηση λειτουργεί αναδρομικά. Αρχικά αν ο κόμβος node δεν έχει παιδιά τότε επιστρέφει αφού δεν έχει τίποτα να συμπιέσει. Στην συνέχεια δημιουργεί μια λίστα που θα εισάγει όλους τους δείκτες σε trie node που πρέπει να συμπιεστούν. Έτσι, όσο το node έχει ένα παιδί, εισάγει τον δείκτη στην λίστα και βάζουμε το node να δείχνει στο παιδί του. Όταν βρει node με περισσότερα από ένα παιδί τον εισάγει και αυτόν και έπειτα καλεί την convert_toHyperNode2 να συμπιέσει όλους τους κόμβους στην λίστα. Τέλος, πλέον ο δείκτης μας πρέπει να δείξει στον πρώτο δείκτη που μπήκε στην ουρά όποτε το καταφέρνουμε αυτό με την βοηθητική μεταβλητή Home. Έτσι, έπειτα καλούμε την CompressTrie2 για τα υπόλοιπα παιδιά του node μας.

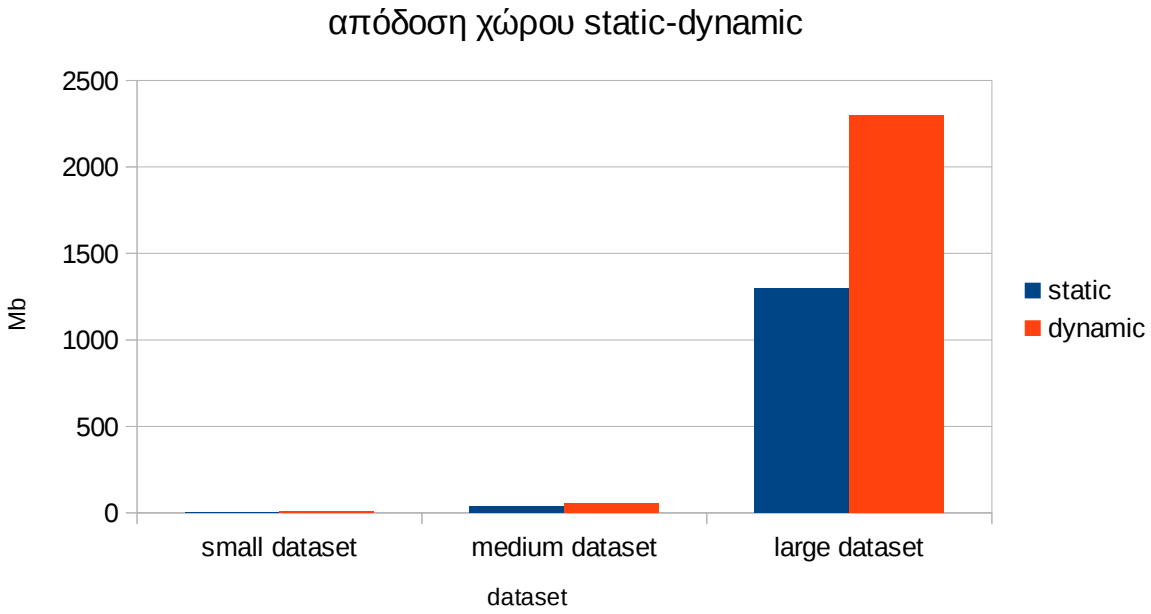
•void convert_toHyperNode2(Queue* q)

Έχει παρόμοια λειτουργεία με την convert_toHyperNode άπλα αντί να συμπιέζει τους δυο κόμβους μόνο συμπιέζει όλους τους κόμβους που έχει η λίστα μας με δείκτες. Ο κόμβος που διατιριται είναι αυτός που εισάχθηκε πρώτος στην ουρά.

## Παρατήρηση

Και οι δυο τρόποι υλοποίησης δεν παρουσιάζουν σημαντική αλλαγή στο χρόνο ή στην κατανάλωσης μνήμης αφού έχουν παρόμοια πολυπλοκότητα.






(small_dynamic :6.1MB, small_static: 5.0MB ,medium_dynamic: 53.1MB, medium_static: 38.1MB)

Συγκριτικά static με dynamic παρατηρούμε μείωση του χρόνου και του χώρου. Ωστόσο η μεγάλη διαφορά χρόνου που εντοπίζεται στο large dataset οφείλεται στις πολλές διαγραφές που συμβαίνουν μίας και για την διαγραφή καλούνται δύο συναρτήσεις που εξερευνούν το δέντρο.

# Βοηθητικές Δομές

Για την υλοποίηση της εργασίας χρησιμοποιηθήκαν εκτός τον παραπάνω πολλές άλλες βοηθητικές δομές, οι οποίες βοήθησαν να επιλυθούν κάποια προβλήματα και να βελτιστοποιηθεί η κατανομή των δεδομένων.

## Βοηθητικές δομές που βρίσκονται στο structures.h

•Char : Περιέχει ένα char* και ένα char[] . Την χρησιμοποιούμε για να αποθηκεύσουμε αλφαριθμητικά σε διάφορες δομές και αναλόγως αν η λέξη είναι μικρότερη από έναν αριθμό SIZE τότε αποθηκεύεται στο char[] αλλιώς στο char*. Με αυτό επιτυγχάνουμε καλύτερο locality. Η Char_Init το δημιουργεί ένα Char και το επιστρέφει και η Char_binary_search κάνει binary search σε array από Char.

•String_Array: Περιέχει έναν πίνακα από char* (char**) , ένα size που δείχνει πόσα περιέχει και ένα max που δηλώνει πόσος είναι ο μέγιστος αριθμός που χωράει , πριν χρειαστεί realloc. Αυτή η δομή χρησιμοποιείται στις find και περιέχει τον πίνακα με τις λέξεις. Τον πίνακα με

τις λέξεις τον παράγουμε με την χρήση της split_str η οποία παίρνει ένα κείμενο (char*) , το σπάει σε λέξεις και τις αποθηκεύει στον πίνακα του String_Array.

•Str_Cntr : Επρόκειτο για μια δομή η οποία περιέχει ένα string και διάφορες πληροφορίες σχετικά με αυτό, όπως το πόσα string του έχουν ενταχθεί, το μέγεθος του και το πόσα χωράει. Χρησιμοποιείται κυρίως στις Find και περιέχει την τιμή η οποία επιστρέφεται.

•Argument: Περιέχει όλα τα δεδομένα που στέλνονται ως ορίσματα στην FindHashtable. Επειδή η FindHashtable στον πολυνιματισμό καλείται από τα νήματα του jobScheduler, θέλαμε τα νήματα να μην χρειάζεται να παίρνουν πολλά δεδομένα για να τα στείλουν ως ορίσματα , επίσης θέλαμε να κατασκευάσουμε τον jobscheduler με τέτοιον τρόπο ώστε να είναι προσαρμόσιμος σε οποιαδήποτε άλλο πρόγραμμα για αυτό κάναμε την συνάρτηση που παίρνει το job να παίρνει ορίσματα σε void* και να επιστρέφει void*. Ωστόσο για την υλοποίηση του top-k , μιας και το κάθε νήμα έχει τον δικό του πίνακα, το κάθε νήμα θα πρέπει να έχει έναν μοναδικό αριθμό ο οποίος θα υποδείχνει στο νήμα ποιος πίνακας είναι ο δικός του. Ωστόσο κάθε νήμα αυτόν τον αριθμό τον έπαιρνε μέσα στο jobRoutine και δεν γινόταν να τον εκχωρήσουμε μέσα στο Argument, για αυτό το δίνουμε ως δεύτερο όρισμα στη συνάρτηση. Το Argument περιέχει το version στο οποίο είναι το νήμα, ένα bool που δηλώνει αν το trie είναι static ή όχι, το hashtable και το string στο οποίο θα ψάξουμε για τα ngrams.

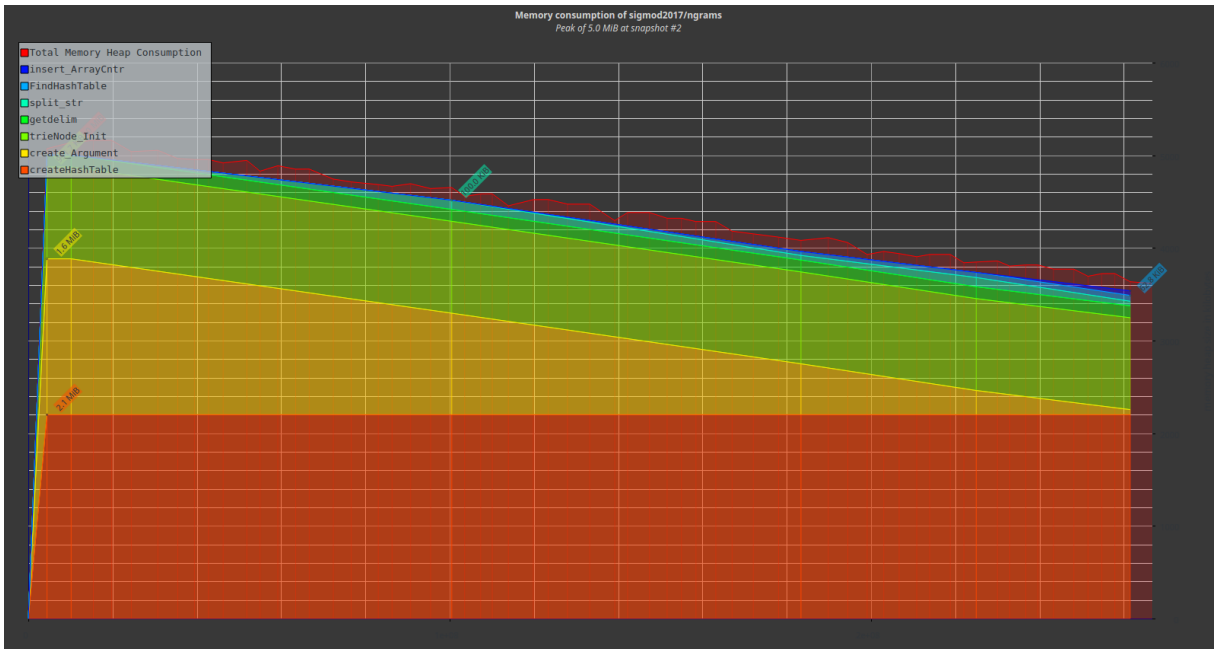
•stack_node: Περιέχει ένα trie_node* tnode το οποίο δείχνει σε ένα node στο trie , και ένα int kid_pos το οποίο δηλώνει σε ποια θέση στον πίνακα του πατέρα βρέθηκε. Χρισιμοποιήται για την Delete για να ξέρουμε ποιον κόμβο να διαγράψουμε και για να ξέρουμε σε ποια θέση ήταν στον πίνακα του πάτερα ο οποίος θα πρέπει να φτιαχτεί.

•Stack:Χρησιμοποιήθηκε στην Trie_Delete. Περιέχει ένα stack_node* snodes το οποίο γίνεται με malloc ένας πίνακας τον οποίο διαχειριζόμαστε ως stack. Το top δείχνει τη θέση του πάνω κόμβου και το size το μέγεθος.

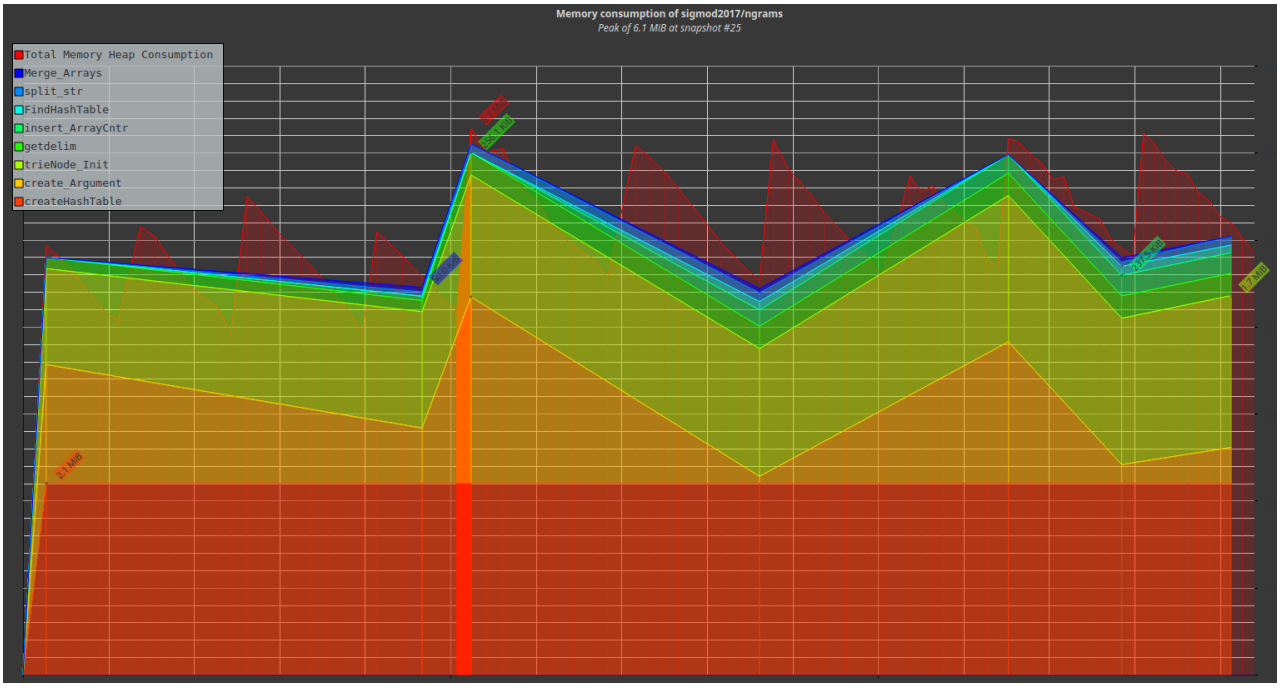
## Αναλυτική κατανάλωση χώρου του προγράμματος

Η κατανάλωση χώρου του προγράμματος υπολογίστηκε με την χρήση του προγράμματος valgrind και του εργαλείου massif.

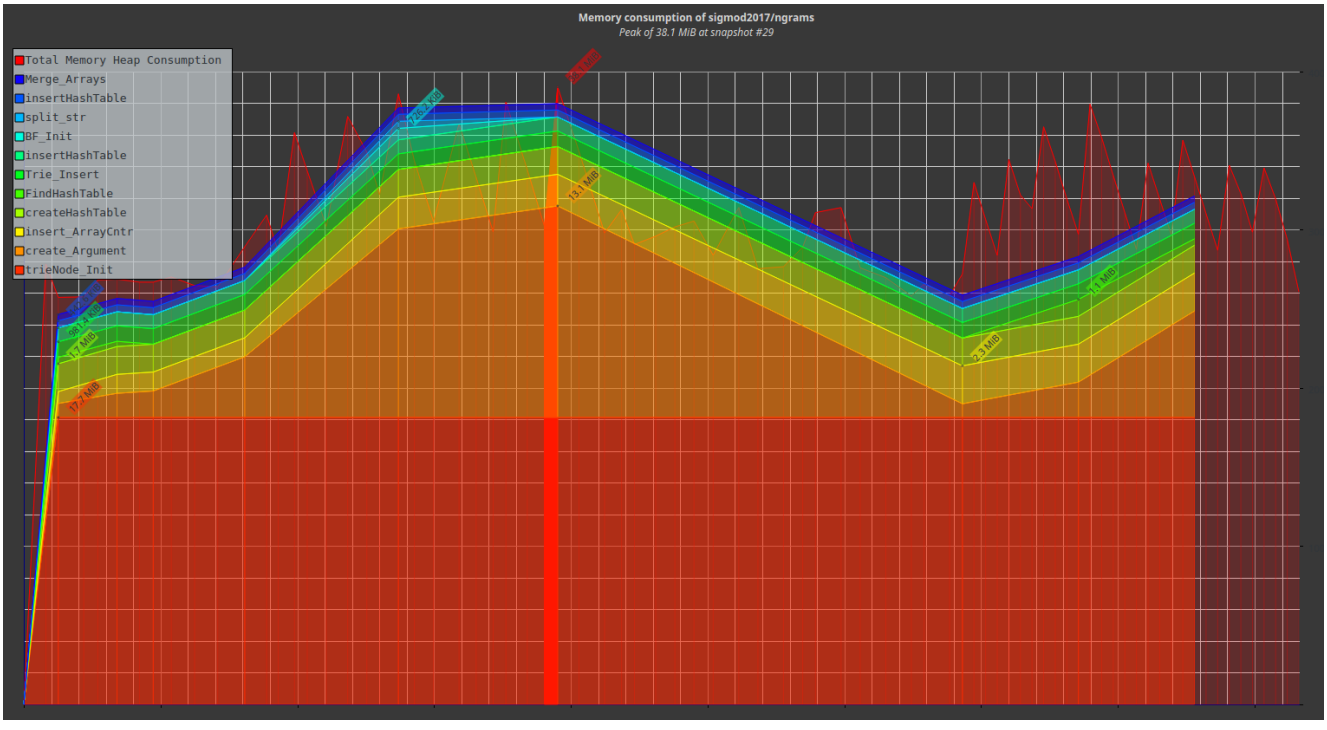
•small_static : peak 5.0MB



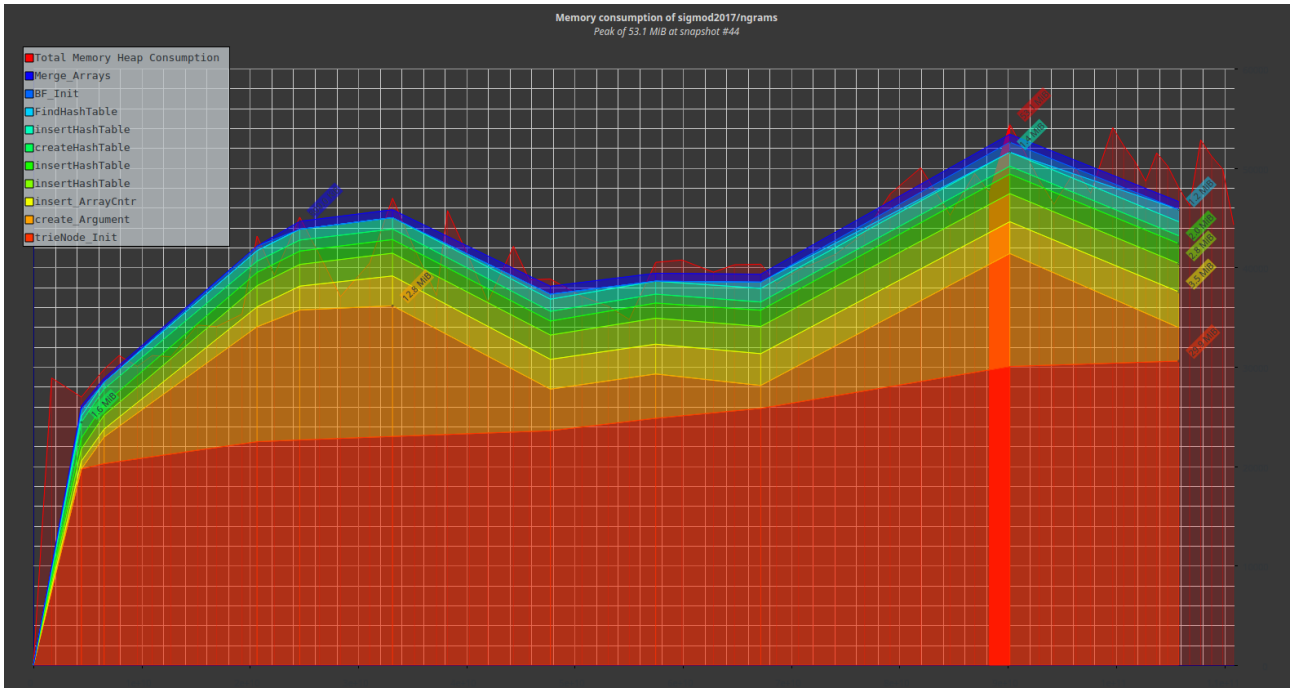
•small_dynamic : peak of  6.1MB



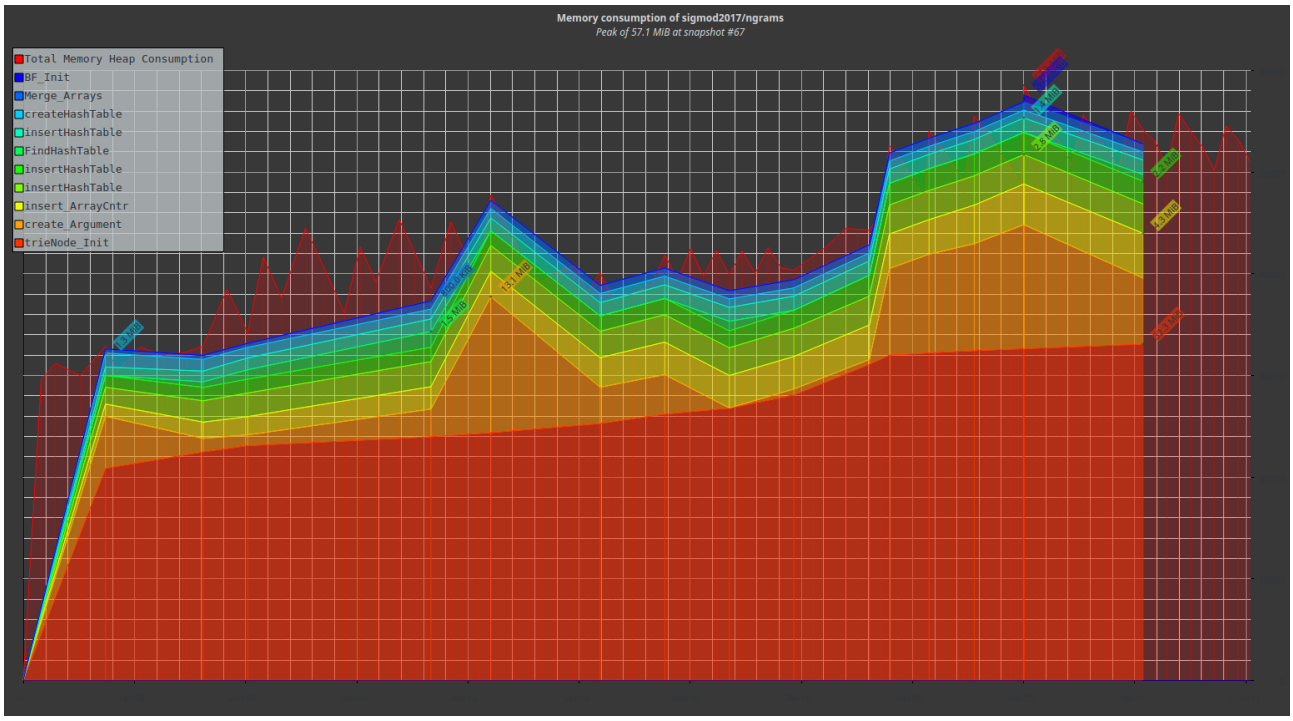

•medium _static : peak of 38.1 MB



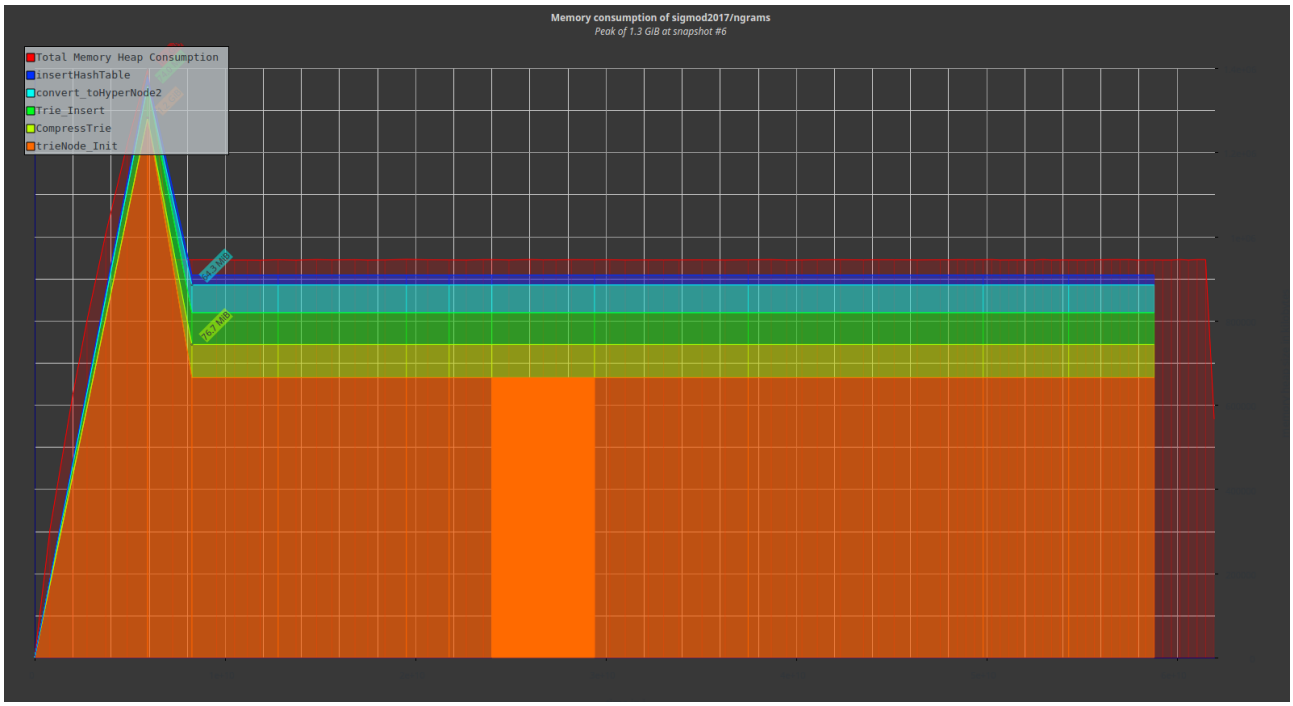
•medium_dynamic : peak of 53.1



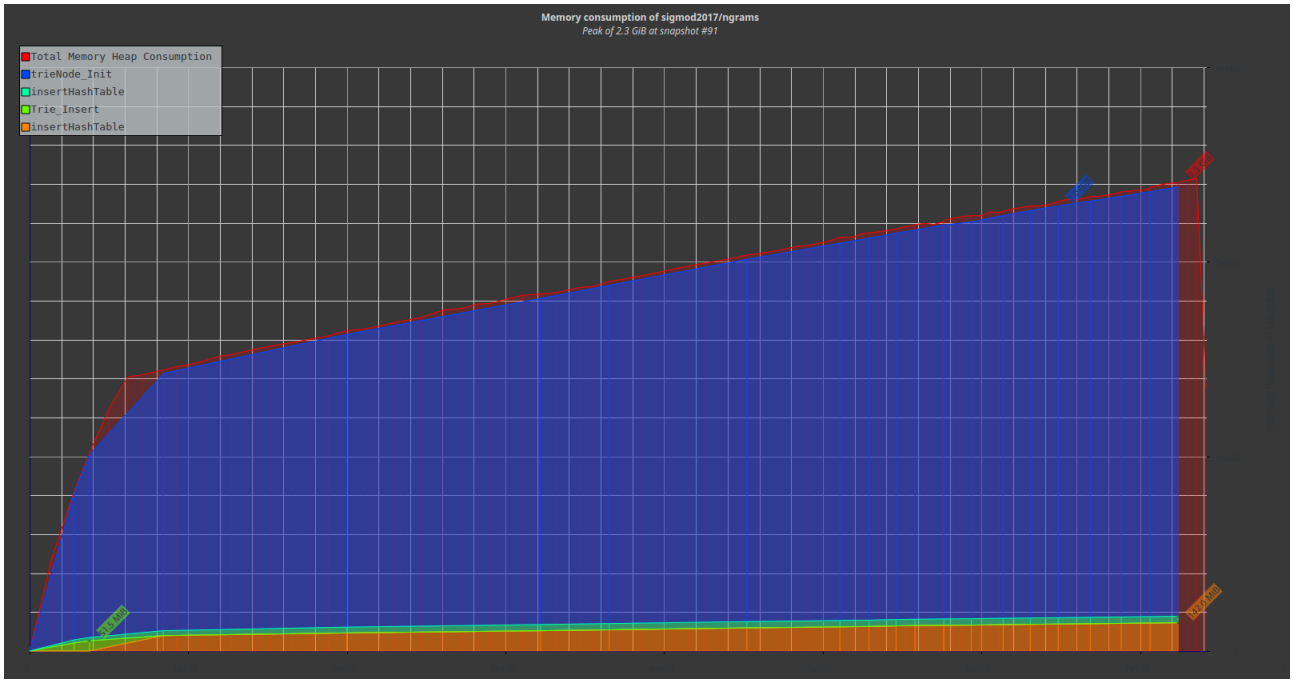

•medium_dynamic χωρίς διαγραφές : peak of 57.1MB



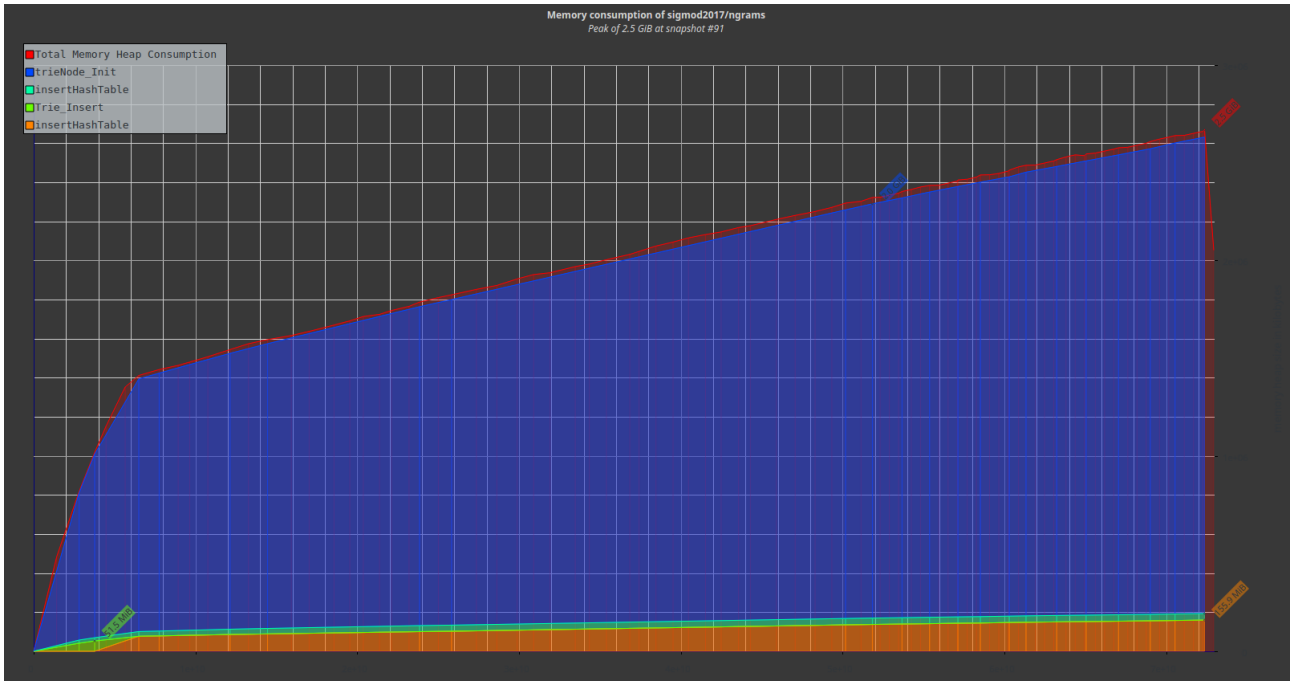
•large_static :peak of  1.3 GB




•large_dynamic : peak of 2.3GB



•large_dynamic χωρίς διαγραφές: peak of 2.5GB




## Παρατηρήσεις

Παρατηρούμε ότι τον περισσότερο χώρο τον πιάνουν trie_node για αυτό προσπαθήσαμε να μειώσουμε όσο το δυνατόν περισσότερο τις μεταβλητές του. Για παράδειγμα μειώσαμε το αρχικό μέγεθος που δίναμε στον πίνακα του trie_node (CHILDREN) . Αυτό είχε ως συνέπεια να γίνεται πιο συχνά realloc , ωστόσο οι περισσότεροι κόμβοι παρέμεναν με λίγα παιδιά οπότε έτσι γλιτώσαμε οι κόμβοι να μην δεσμεύουν άσκοπα μνήμη. Επίσης παρατηρήσαμε πως οι λέξεις έχουν συγκριτικά μικρό μέγεθος οπότε μειώσαμε και το SIZE , το οποίο είναι το μέγεθος των στατικών λέξεων. Αυτές οι αλλαγές ήταν πολύ σημαντικές γιατί όταν προτρέξαμε τα large datasets ο υπολογιστής σχεδόν ξέμενε από ram, μιας και αυτές οι μεταβλητές την δέσμευαν άσκοπα.

Επίσης παρατηρούμε ότι η συμπίεση τους κόμβους και η παραγωγή στατικού δέντρου έχει μεγάλη μείωση στον χώρο που καταναλώνει. Αυτό γίνεται εμφανές στο large dataset όπου η διαφορά αγγίζει μέχρι και το 1GB.

# Αναλυτική Χρονική απόδοση του προγράμματος

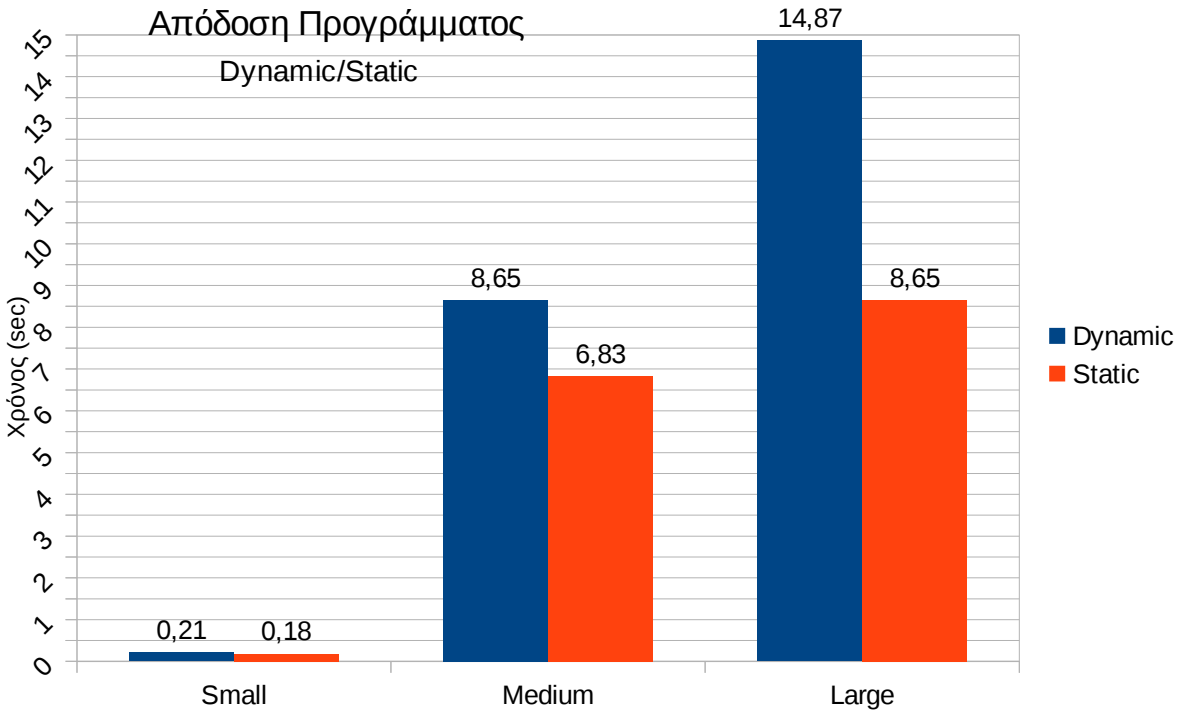
Όλοι οι χρόνοι μετρήθηκαν χρησιμοποιώντας το εργαλείο time τον linux. Ο υπολογιστής στον οποίο τρέξαμε το πρόγραμμα έχει επεξεργαστή intel i5 6500 ο οποίος είναι 4πύρινος με 3,2Gz απόδοση και μνήμη 6MB, και με 8GB ram. Η τελικές δοκιμές έγιναν με την χρήση 6 threads.

```
real    0m8.709s
user    0m32.345s
sys     0m0.372s
```

Ενδεικτική εικόνα της χρήσης του προγράμματος time.

Οι μέση χρόνοι του προγράμματος για τα dataset είναι οι εξής:

| Datasets | Dynamic | Static |
|---|---|---|
| Small | 0,212 | 0,184 |
| Medium | 8,654 | 6,831 |
| Large | 14,828 | 8,703 |




## Σχεδιαστικές επιλογές που μείωσαν τον χρόνο.

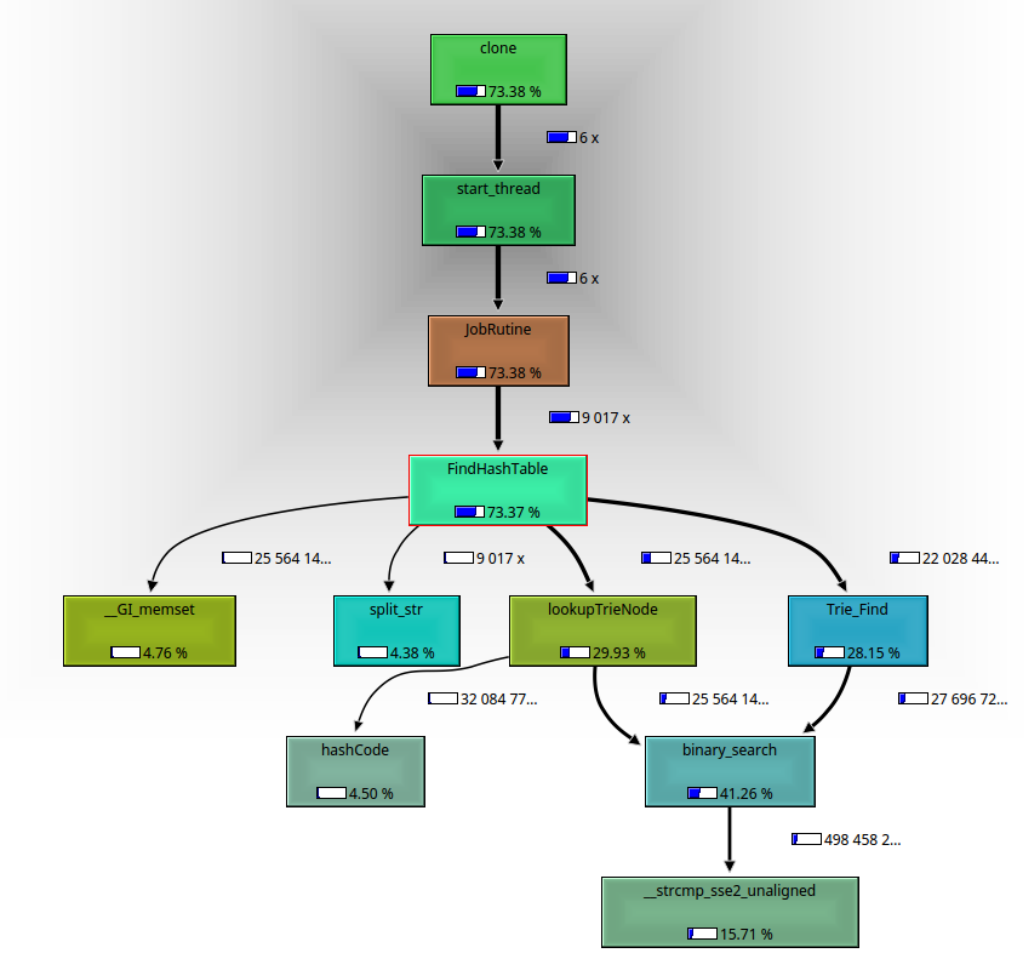
•Καλό locality : Μεγάλη έμφαση δώσαμε στον να υπάρχει καλό locality μεταξύ των δεδομένων. Αποφύγαμε την χρήση δεικτών ειδικότερα σε δεδομένα τα οποία τα χρισιμοποιούμε συχνα, όπως τους πίνακες των trie_node (είναι πίνακας με trie_nodes και όχι δείκτες), στα buckets όπου ο πίνακας περιέχει και αυτός trie_nodes. Επίσης οι λέξεις αποθήκευονται στατικά όποτε ειναι μικρότερες απο ένα συγκεκριμένο μέγεθος SIZE.

•Αντικατάσταση του pow: Μιας και στην ανεύρεση κόμβων στο hashtable χρειαζόταν να υπολογιστεί μια δύναμη του 2 την αντικαταστήσαμε με διάδικο shift. Αυτό μείωσε σημαντικά τον χρόνο μιας και ανεύρεση κόμβων από το hashtable είναι η πιο συχνή διαδικασία που γίνεται στο πρόγραμμα.

•Αντικατάσταση του '%' : Για τον ίδιο λόγο αντικαταστήσαμε την πράξη του mod με μια λογική πράξη (n%x ==> n & (x-1) ) , ωστόσο για να λειτουργεί αυτό σωστά θα πρέπει το x να είναι

πολλαπλάσιο του 2, για αυτό τον λόγο προσαρμόσαμε τα νούμερα ώστε πάντα να είναι πολλαπλάσια του 2.

Ο γράφος που παράγει το kcachegrind το οποίο χρησιμοποιήθηκε για να βελτιώσουμε τον χρόνο του προγράμματος.




Η πρόοδος μεταξύ των parts φαίνεται στο παρακάτω διάγραμμα.