ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Κ23α – Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2018-2019

Ζήσης Μπεληγιάννης 1115201400118

Παναγιώτης Στασινός 1115201400191

Χρονομετρήσεις

Οι χρονομετρήσεις έγιναν στα workstation της σχολής (2 cores με 2 threads ο καθένας).

Παράμετροι Παραδοτέων

Παράμετροι με τις οποίες παραδόθηκε η σειριακή υλοποίηση: h1=12, h2=8, buffer Size=1MB, χρήση greedy αλγορίθμου για την επιλογή εκτέλεσης των predicate.

Παράμετροι με τις οποίες παραδόθηκε η παράλληλη υλοποίηση:h1=8, h2=6, buffer Size=128000 KB, αριθμός threads 1, χρήση greedy αλγορίθμου για την επιλογή εκτέλεσης των predicate.

*προτιμήθηκε ο greedy αλγόριθμος από τον αλγόριθμο BestTree για τους λόγους που αναφέρονται παρακάτω.

Καλύτερες Παράμετροι

Παρατηρήσαμε οτι οι καλύτερες χρονομετρήσεις επιτυγχάνονται με h1=8, h2=6 και buffer Size=64000 KB και χρήση 3 thread για την παράλληλη υλοποίηση.

UPDATE (9/3/19)

Χρονομετρήσεις με compiler optimization με χρήση της flag –O (τα καινούρια Makefiles βρίσκονται στον αρχικό φάκελο master).

Χρονομέτρηση Σειριακού

BUFFER SIZE	h1,h2 = 8,6	h1,h2 = 12,8	h1,h2 = 12,10
64000	0,6086	0,6297	0,6446
12800	0,6106	0,6284	0,6462
256000	0,6113	0,6357	0,6461
512000	0,6147	0,6369	0,6491
1024000	0,6211	0,6392	0,6536

Χρονομέτρηση Παραλληλοποιημένου

Σταθερό μέγεθος buffer 64000 byte

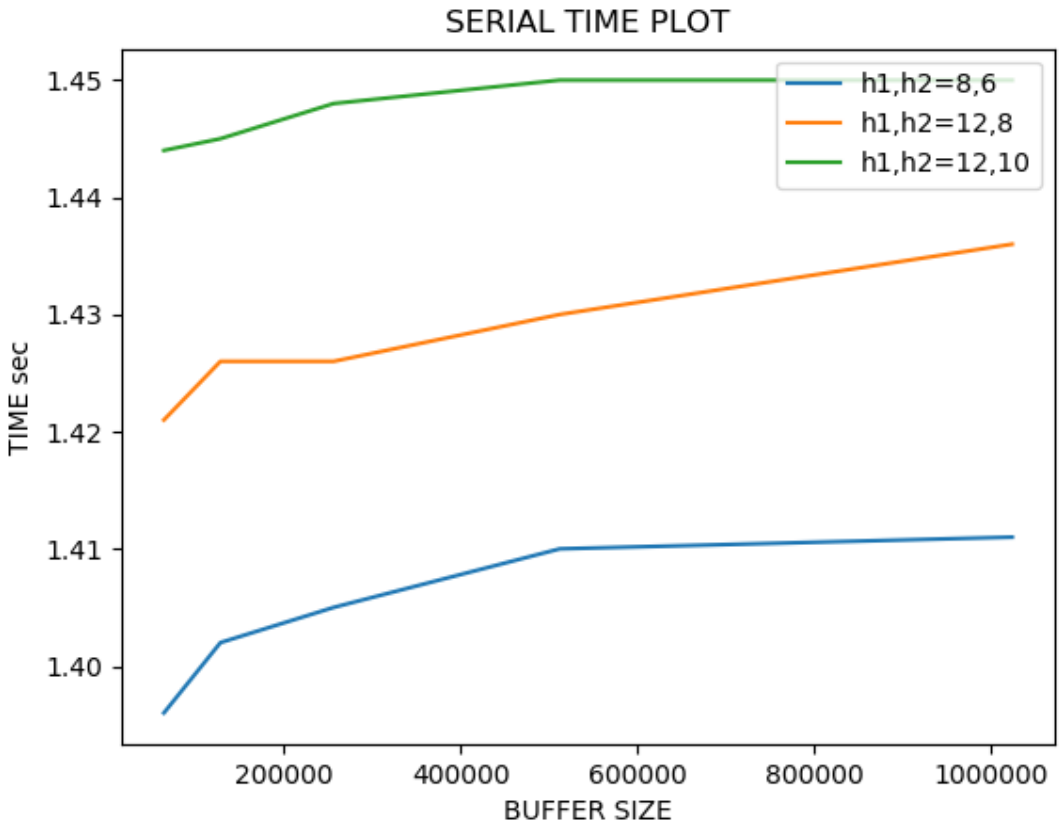
NUMBER OF THREADS	h1,h2 = 8,6	h1,h2 = 12,8	h1,h2 = 12,10
1	0,7595	1,2530	1,2568
2	0,7469	1,2462	1,2338
3	0,7285	1,2556	1,2679
4	0,8652	1,3640	1,4106
5	0,8987	1,4658	1,4729

Παρατηρούμε ότι με το optimization του compiler οι χρόνοι μειώνονται ακόμα περισσότερο.

END OF UPDATE

Χρονομέτρηση Σειριακού

BUFFER SIZE	h1,h2 = 8,6	h1,h2 = 12,8	h1,h2 = 12,10
64000	1,396	1,421	1,444
12800	1,402	1,426	1,445
256000	1,405	1,426	1,448
512000	1,41	1,43	1,45
1024000	1,411	1,436	1,45

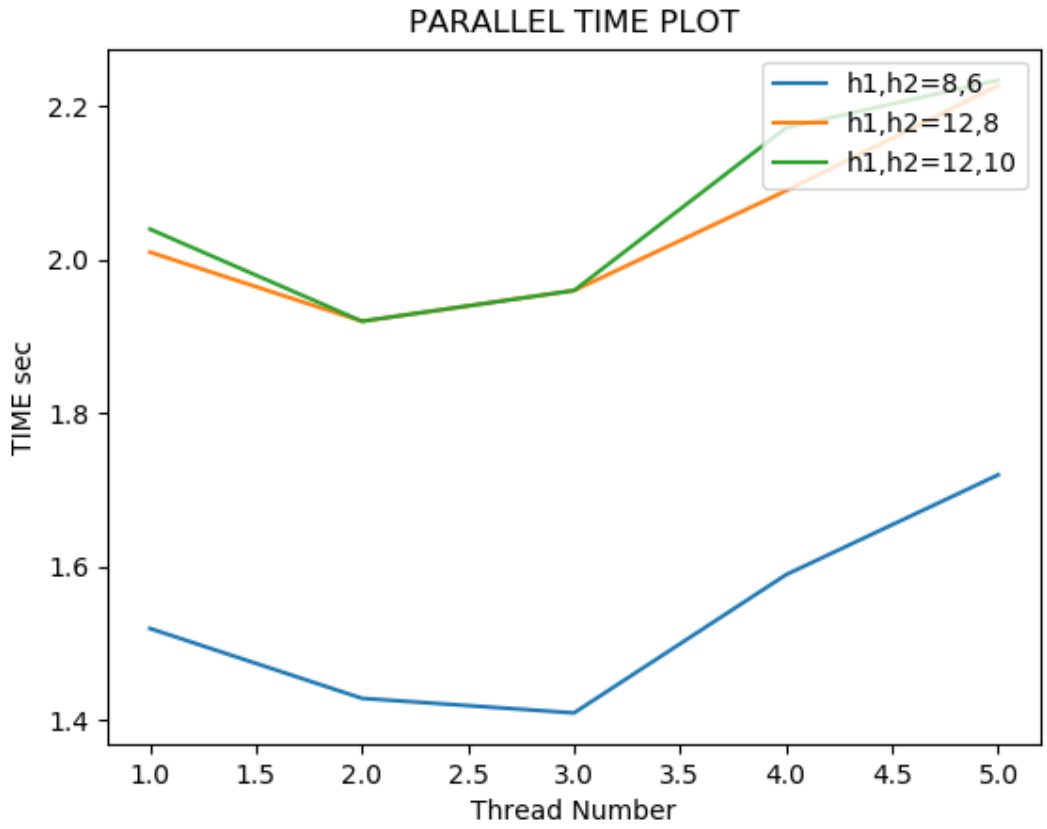


(Παρατήρηση : για μικρότερα μεγέθη του buffer ο χρόνος αυξανόταν, καλύτεροι χρόνοι με buffer size 64000 Byte)

Χρονομέτρηση Παραλληλοποιημένου

Σταθερό μέγεθος buffer 64000 byte

NUMBER OF THREADS	h1,h2 = 8,6	h1,h2 = 12,8	h1,h2 = 12,10
1	1,52	2,01	2,04
2	1,429	1,92	1,92
3	1,41	1,96	1,966
4	1,59	2,09	2,172
5	1,72	2,227	2,234



(Παρατήρηση : επειδή το πρόγραμμα τεσταρίστηκε στους υπολογιστές της σχολής οι οποίοι διαθέτουν 4 thread είναι λογικό ο χρόνος να βελτιστοποιείται με thread pool 3 thead, καθώς η δουλεια μοιράζεται ομοιόμορφα μεταξεί των 3 thread και του main thread (4 στο σύνολο))

Χρονομέτρηση με χρήση BestTree

Ο χρόνος του σειριακού με χρήση BestTree, buffer 64000 bytes, h1=8 και h2=6 είναι 2,7681secs και ο αντίστοιχος του παραλληλοποιημένου για 3 threads είναι 2,7901secs.

Η απόκλιση των χρονομετρήσεων μεταξύ του BestTree και της greedy μεθόδου (calculate_priority()) οφείλεται στην επιλογή σειράς εκτέλεσης των predicates καθώς με τη χρήση του BestTree παρουσιάζεται η περίπτωση του nested loop join η οποία είναι πιο αργή σε σύγκριση με το RadixHashJoin.

Thread Pool

Το πληθος των thread που δημιουργεί η συνάρτηση thread_pool_create() καθορίζεται με #define μέσα στο αρχείο thread_pool.h .

Το thread pool αρχικοποιείται μέσα στην main πριν το διάβασμα των query και απευθείας γίνεται sleep. Στην RadixHashJoin ενεργοποιείται για να γίνουν οι παράλληλοι υπολογισμοί και στη συνέχεια πάλι αδρανοποείται. Επίσης ενεργοποιείται για τον υπολογισμό του αθροίσματος του αποτελέσματος του query. Τέλος αφου εχουν γίνει όλα τα queries στην main καταστρέφεται.

BONUS ΕΡΩΤΗΜΑ

Έχει υλοποιηθεί και το bonus ερώτημα για την παραλληλοποίηση του υπολογισμού των αθροισμάτων. Βρίσκεται στο αρχείο query.c στην συνάρτηση print_sum() (ο κώδικας βρίσκεται σε σχόλιο καθώς έδινε ίδιους χρόνους με την σειριακή υλοποίηση) στην οποία για κάθε στήλη δημιουργείται ένα job το οποίο υπολογίζει το αντίστοιχο άθροισμα.

Manual Συναρτήσεων

parse.c :

- **void information_storing(struct file_info* , uint64_t *, struct file_stats *);**
  συνάρτηση η οποία αποθηκέυει την απαραίτητη πληροφορία για κάθε σχέση και συγκεκριμένα τον αριθμό γραμμώ, αριθμό στηλών και τους δείκτες προς τις στήλες των σχέσεων.Επίσης υπολογίζει τα στατιστικά για κάθε σχέση, δηλαδή βρίσκει τον μικρότερο και το μεγαλύτεο στοιχείο για τις στήλες όλων των σχέσεων καθώς και τον πλήθος των distinct values κάθε στήλης σύμφωνα με τους τύπους που έχουν δοθεί στην εκφώνηση της

εργασίας.Για τον υπολογισμό του πλήθους των distinct στοιχείων χρησιμοποιείται bit vector για την εξοικονόμηση χώρου.

- **void print_query_info(struct query_info* query);**
  εκτυπώνει τις πληροφορίες του query που έχει δοθεί ως όρισμα.
- **void insert_pred(struct query_info* query, char* pred, int index);**
  αποθήκευση predicate.

query.c :

- **void calculate_query(struct query_info *, struct file_info*, struct thrd_pool *thread_pool);**
  η συνάρτηση καλείται για κάθε query του αρχείου και υπολογίζει τα ζητούμενα αθροίσματα.Στη συνάρτηση αυτή καλείται η Radix Hash Join παίρνοντας ως ορίσματα τις κατάλληλες σχέσεις που πρέπει να γίνουν join. Η κύρια λούπα της συνάρτησης γίνεται για κάθε predicate που υπάρχει στο query και ανανεώνει τα ενδιάμεσα αποτελέσματα μετά από κάθε επανάληψη.Στην συνάρτηση αυτή λαμβάνονται υπόψη και υποπεριπτώσεις που αφορούν τα predicate, όπως για παράδειγμα η περίπτωση να συναντήσουμε predicate που υπάρχει παραπάνω από μία φορά στο query. Η σειρά με την οποία εκτελούνται τα predicate καθορίζεται με βάση κάποιους αλγορίθμους που έχουν υλοποιηθεί, όπως είναι η calculate_priority ή h QueryOptimization(Best Tree).Στο τέλος της συναρτήσης υπολογίζονται και εκτυπώνονται τα αθροίσματα τα οποία "ζητούνται" από το query.
- **result* comparison_query(struct file_info *, int , uint64_t , int, char, result *);**
  η συνάρτηση καλείται σε περίπτωση που κάποιο predicate είναι φίλτρο.Ελέγχει τις τιμές της σχέσης και αν ικανοποιούν την συνθήκη του εκάστοτε φίλτρου τότε αποθηκέυει τα σωστά row Ids στα ενδιάμεσα αποτελέσματα.
- **int calculate_priority(struct priority *, qinfo *, finfo *, int);**
  συνάρτηση η οποία αναθέτει για κάθε predicate μια τιμή προτεραιότητας σύμφωνα με την οποία γίνεται η επιλογή με την οποία θα τρέξει κάθε predicate.Ο αλγόριθμος ακολουθεί greedy λογική καθώς σε κάθε επανάληψη επιέγει εκείνο το predicate που επιστρέφει τα λιγότερα αποτελέσματα και το οποίο δεν βέβαια δεν έχει εκτελεστεί ήδη.Επίσης τα predicate τύπου φίλτρο εκτελούνται πρώτα καθώς στην βοηθούν στην βελτίωση του χρόνου αλλά και του χώρου.Τέλος ελεγχέται η περίπτωση των cross products καθώς είναι κάτι που επιβαρύνει τον υπολογισμό των αποτελεσμάτων το οποίο θέλουμε να αποφύγουμε.
- **void calculate_sum(struct result*, struct query_info *, struct file_info *, int, uint64_t);**
  υπολογισμός των αθροισμάτων που πρέπει να εκτυπωθούν.Η συνάρτηση λαμβάνει τα ενδιάμεσα αποτελέσματα τα οποία περιέχουν τα row Ids των γραμμών και με την χρήση των δεικτών που έχουμε για τις στήλες των αρχείων (και χρησιμοποιόντας τα row Ids ως offset) υπολογίζει τα ζητούμενα αθροίσματα.
- **void print_sum(struct result*, struct query_info *, struct file_info *, struct thrd_pool *);**
  λαμβάνει τα Ids των column για τις οποίες πρέπει να προβληθούν τα αθροίσματα και καλεί την συνάρτηση calculate_sum για να πραγματοποιήσει τους υπολογισμούς.
- **int predicate_exists(qinfo *, int, int);**
  ελέγχει την περίπτωση που ένα predicate βρίσκεται στο query παραπάνω από μία φορά. Στην περίπτωση που η εμφάνιση του συγκεκριμένου predicate δεν είναι μοναδική ο αλγόριθμος φροντίζει η πραγματοποίηση του να γίνει μία φόρα.

utils.c :

- **void update_results(result *, result *, int,**
  result *, int, result*, struct file_info*, struct query_info *, int);
  συνάρτηση αρχικοποίησης των ενδιάμεσων αποτελεσμάτων.Σε περίπτωση που οι ενδιάμεσες δομές έχουν δημιουργηθεί ήδη καλεί την update_interlists για την ανανέωση των ενδιάμεσων αποτελεσμάτων.
- **void update_interlists(result *, result *, result *, int, result *, int, struct file_info*, struct query_info *, int);**
  συνάρτηση ανανέωσης των ενδιάμεσων αποτελεσμάτων η οποία χρησιμοποιεί τα indexes των προηγούμενων ενδιάμεσων αποτελεσμάτων για να αποθηκεύσει τα νέα αποτελέσματα.
- **void updateDifferCol(relation *, relation *, result *, finfo *, qinfo *, int);**
  ανανέωση ενδιάμεσων αποτελεσμάτων σε περίπτωση πραγματοποίησης join μεταξύ σχέσεων που έχουν γίνει ήδη join αλλά για διαφορετικές στήλες η μία τουλάχιστον από τις δύο.
- **void relation_similarity(relation *, relation *, result *, finfo *, qinfo *, int);**
  συνάρτηση στην περίπτωση που έχουμε join μεταξύ στηλών της ίδιας σχέσης.
- **void update_existing_interlists(relation *, relation *, result *, finfo *, qinfo *, int);**
  συνάρτηση για την περίπτωση που και οι δύο σχέσεις που συμμετέχουν στο predicate βρίσκονται ήδη στα ενδιάμεσα αποτελέσματα.

- **void update_results_filter(result *, result *, int , finfo *, qinfo *, int, struct thrd_pool *);**
  δημιουργία ενδιάμεσων δομών μετά την εκτέλεση φίλτρου ή κλήση της συνάρτησης update_interlists_filter για την ανανέωση των ενδιάμεσων αποτελεσμάτων σε περίπτωση που αυτά υπάρχουν.
- **void update_interlists_filter(result *, result *, result *, int, finfo *, qinfo *, int);**
  ανανέωση ενδιάμεσων αποτελεσμάτων μετά την εκτέλεση φίλτρου.

- **void create_relation(struct relation*, struct file_info *, int rel_id, uint64_t column);**
  δημιουργία relation από τα αρχεία που έχουμε αποθηκεύσει στην μνήμη.
- **void create_rel_from_list(struct relation*, struct result*, struct file_info *, int, uint64_t);**
  δημιουργία μιας relation από λίστα που περιέχει τα ενδιάμεσα αποτελέσματα.Τα keys της relation αντιστοιχούν στις γραμμές των αρχείων που έχουμε ως δεδομένα και που έχουμε αποθηκεύσει στη μνήμη. Τα payloads και αυτά αντιστοιχούν στα αρχεία.
- **void create_interlist(struct result *, struct result*, struct result*, struct file_info*);**
  η συνάρτηση αυτή χρησιμοποείται μετά την ολοκλήρωση της συνάρτησης Radix Hash Join η οποία μας έχει επιστρέψει τους συνδυασμούς τιμών των δύο σχέσεων που έγιναν join.Η create_interslist αποθηκέυει τις τα row Ids της κάθε σχέσης στην δική της ξεχωριστή λίστα.
- **void list_to_relation(struct relation*, struct result*, struct file_info *, int, uint64_t);**
  η συνάρτηση δημιουργεί relation από τα ενδιάμεσα αποτελέσματα με key τον index που αντιστοιχεί στην ενδιάμεση δομή και με payload το πραγματικό value της αντίστοιχης τιμής του αρχείου.

- **void list_to_rel_with_indexes (struct relation*, struct result*);**
  δημιουργία relation από λίστα result χρησιμοποιώντας ως payload τον value της λίστας.
- **void copy_result(result* dest, result* source);**
  συνάρτηση αντιγραφής της λίστας με μεταβλητή source στη λίστα dest.

thread_pool.c :

- **int thrd_pool_wait(struct thrd_pool *temp);**
  μπολκαρει το main thread μεχρι να μην εχει μείνει καμία δουλειά στην job_queue και κανένα thread να μην εκτελεί ακόμα καποιο job.
- **Int thrd_pool_create(struct thrd_pool *temp);**
  αρχικοποιεί τα mutexes της job queue και του μετρητή των active thread, το flag end σε μηδεν, δημιουγεί την job queue και στο τελος δημιουργει τα thread τα οποία εκτελούν την συναρτηση ready().
- **Int thrd_pool_destroy(struct thrd_pool *temp);**
  δίνει στην flag μεταβλητή end τιμή 1 ωστε να τερματίσουν ολα τα threads και στην συνέχεια περιμένει ολα τα threads να τερματίσουν. Τέλος κάνει destroy τα mutexes και τα semaphores.
- **int thrd_pool_sleep(struct thrd_pool *temp);**
  μπλοκαρει τα thread μέσω του semaphore που αντιστοιχεί στο καθένα
- **int thrd_pool_wake(struct thrd_pool *temp);**
  ξεμπλοκάρει τα thread ώστε να είναι έτοιμα να πάρουν jobs

job_scheduler.c :

- **void *ready(void *args);**
  λειτουργία : *εκτελείται από τα thread κατα την δημιουργία τους. Κάθε thread εκτελεί την επανάληψη ζητώντας για job να εκτελέσει. Αν του επιστραφεί NULL δεν υπάρχει κάποια εργασία να κάνει και επιστρεφει στην αρχη της επανάληψης. Αν έχει κληθεί η thrd_poool_sleep() τότε μπλοκάρεται στο semaphore του και περιμενει για κληση της thrd_pool_wake() για να συνεχίσει, αλλιώς συνεχιζει να ζητάει για job μεχρι να πάρει. Αν λαβει job, αυξάνει την μεταβλητή των ενεργών thread κατά ένα, εκτελεί το job και στην συνέχεια μειώνει κατά ένα την μεαβλητή.*
- **int job_queue_create(struct job_queue *queue);**
  λειτουργία : *αρχικοποιεί την job queue με μήκος μηδέν*
- **int job_queue_push(struct thrd_pool *pool, int (*func)(void**), void **args);**
  λειτουργία : *καλείται απο το main thread για να εισάγει μια νεα δουλειά στην job_queue*
- **int job_queue_pull(struct job_queue *queue, struct job *temp_job);**
  λειτουργία : *καλείται από τα thread του pool για να λάβουν μια εγασια να κανουν, επιστρέφει στο temp_job->func NULL αν δεν υπαρχει job για εκτέλεση.*
- **int HistogramJob(void **args);**
  (int)args[0] , start_index για την αρχική relation
  (int)args[1] , end_index για την αρχική relation

(struct histogram*)args[2] , *δείκτης του ιστογράμματος στο οποίο πρέπει να γράψει το συγκεκριμένο job*
(struct relation*)args[3] , *δείκτης στην αρχική σχέση*
(struct histogram*)args[4] , *δείκτης του αθροιστικού ιστογράμματος στο οποίο πρέπει να γράψει το συγκεκριμένο job*
λειτουργία : *αρχικοποιεί το ιστογραμμα και το αθροιστικό ιστογραμμα και στην συνεχεια υπολογίζει το ιστόγραμμα για το κομμάτι απο το start μέχρι και το end*

- **int PartitionJobs(void **args);**
(struct histogram*)args[0] , *δείκτης του ιστογράμματος από το οποίο πρέπει να διαβάσει το συγκεκριμένο job*
(struct histogram*)args[1] , *δείκτης του αθροιστικού ιστογράμματος από το οποίο πρέπει να διαβάσει το συγκεκριμένο job*
(struct relation*)args[2] , *δείκτης στην παλιά σχέση*
(struct relation*)args[3] , *δείκτης στην ordered relation*
(int)args[4] , *start_index για την παλιά relation*
(int)args[5] , *end_index για την παλιά relation*
λειτουργία : *αντιγράφει τα tuples της παλιάς relation που βρίσκονται μεταξύ start και end στο κατάλληλο bucket της ordered relation με βάση τα hist και τα psum του συγκεκριμένου job.*
- **int JoinJobs(void **args);**
(int)args[0] , *start του συγκεκριμένου bucket για την ordered relation R*
(int)args[1] , *end του συγκεκριμένου bucket για την ordered relation R*
(struct relation*)args[2] , *δείκτης στην ordered relation R*
(int)args[3] , *start του συγκεκριμένου bucket για την ordered relation S*
(int)args[4] , *end του συγκεκριμένου bucket για την ordered relation S*
(struct relation*)args[5] , *δείκτης στην ordered relation S*
(struct result*)args[6] , *δείκτης στο result στο οποίο γραφει το συγκεκριμένο job*
λειτουργία : *εκτελεί το radix hash join για ένα συγκεκριμένο bucket*
- **int Sum(void **args);**
(struct result*)args[0] , *δείκτης στο result που έχει τα indexes για συγκεκριμένο relation*
(struct file_info*)args[1] , *δείκτης στις πληροφορίες των relation*
(int)args[2] , *αριθμός relation*
(int)args[3] , *αριθμός column*
(uint64_t*)args[4] , *δείκτης στην μεταβλητή όπου θα αποθηκευτεί το άθροισμα*
λειτουργία : *υπολογίζει το άθροισμα για την συγκεκριμένη σχέση και το συγκεκριμένο column και το γυρνάει πίσω μέσω του (uint64_t*)args[4]*

str.c :

- **result* RadixHashJoin(relation *relR, relation* relS, struct thrd_pool *temp);**
λειτουργία: *επιστρέφει τον δείκτη των result της radix hash join. Κάνει wake up το thread pool και στην συνέχεια καλεί τις συναρτήσεις που δίνουν εργασίες για την στο thread pool για την δημιουργία των hist, psum και των τελικών αποτελεσμάτων.*
- **int Hist_and_Psum();**

λειτουργία : *μοιράζει την δουλειά της δημιουργίας των hist και της αρχικοποίησης των psum. Στην συνέχεια αφού όλα τα thread έχουν τερματίσει και είναι έτοιμα τα ιστογράμματα δημιουργεί σε σειριακό κομμάτι τα psum.*

- **int ReOrdered();**
  λειτουργία **:** *μοιράζει το αρχικό relation σε πλήθος τμημάτων όσο και το πλήθος των thread και αναθέτει σε κάθε thread την συμπλήρωση διαφορετικού τμήματος της reordered. Τέλος περιμένει ολα τα thread να τερματίσουν ώστε να είναι έτοιμο το reodered relation και τερματίζει.*
- **int Join();**
  λειτουργία : *δημιουργεί πλήθος job όσο και το πλήθος των bucket των relation και περιμένει μέχρι να τερματίσουν. Αφού έχει τα αποτελέσματα από όλα τα job τα ενώνει σε ένα και τερματιζει. Αυτό το αποτέλεσμα είναι και το τελικό που επιστρέφει η RadixHashJoin().*

result.c :

- **void result_init(result* result);**
  λειτουργία : *αρχικοποιεί το result με εναν κενό buffer*
- **void insert_result(int rowR, int rowS ,result* result);**
  λειτουργία : *ελέγχει αν υπαρχει χώρος στον τρέχον κόμβο και σε περίπτωση που μπορεί εισάγει την δυάδα αποτελεσμάτων στον buffer, αλλιώς δεσμεύει ενα νέο buffer και τον προσθέτει στην λιίστα και στην συνέχεια εισάγει τα αποτελέσματα*
- **void print_result(result* result);**
  λειτουργία : *βοηθητική συνάρτηση που εκτυπώνει το περιεχόμενο ενός result*
- **void free_result(result* res);**
  λειτουργία : *ελευθερώνει την μνήμη του result που δίνεται*
- **void insert_inter(int row, result* result);**
  λειτουργία : *παρόμοια με την insert_result() με την διαφορά οτι εισάγει μόνο ένα αποτέλεσμα στον buffer, χρησιμοποιείται κυρίως για την δομή των ενδιάμεσων αποτελεσμάτων*

query_selection.c :

- **void QueryOptimization (qinfo*, finfo*, fstats*, int *qselect);**
  λειτουργία : *κατασκευάζει το hashTable BestTree και εκτελεί τον δυναμικό αλγόριθμο ο οποίος επιστρέφει το καλύτερο συνδυασμό που χρησιμοποιεί όλες τις σχέσεις του query.*
- **void JoinEstimation(BestTree*, tree_node*, tree_node*, nodedata*);**
  λειτουργία : *βάση τα στατιστικά των δύο δέντρων που γίνονται Join κάνει την εκτίμηση για το νέο συνδυασμό σχέσεων που θα δημιουργηθεί από τα δύο προηγούμενα*.
- **void GreaterFilterEstimation(tree_node*);**
  λειτουργία: *υπολογίζει τα νέα στατιστικά της σχέσης που συμμετέχει σε predicate τύπου φίλτρου μεγαλύτερο (>) και ανανεώνει τις αντίστοιχες τιμές.*
- **void LessFilterEstimation(tree_node*);**
  λειτουργία: *ίδια λειτουργικότητα με την GreaterFilterEstimation αλλά για φίλτρα τύπου μικρότερο(<).*

- **void EqualFilterEstimation(treenode*);**
  λειτουργία: και *πάλι ίδια λειτουργικότητα με τις δύο προηγούμενες συναρτήσεις αλλά για φίλτρα τύπου ισότητας(=).*
- **void CreateJoinTreee(BestTree*, tree_node* , tree_node*);**
  λειτουργία: *Δημιουργία νέου δεντρού από την ένωση του παλιού δέντρου και του νέου. Στην υλοποίηση μας το νέο δέντρο αποτελείται από μία σχέση.Αρχικοποιεί δεδομένα του νέου δέντρου.*
- **void TreeInsert(BestTree*, char* key, nodedata*,);**
  λειτουργία: *εκχωρεί, με βάση το κλειδί που επιστρέφει η hash function, στο hash Table τον νέο συνδυασμό σχέσεων αφού πρώτα δεσμεύσει τον απαραίτητο χώρο.*
- **Tree_node* TreeSearch(BestTree*, char* key);**
  λειτουργία: *κάνει αναζήτηση στο hast Table το στοιχείο με το συγκεκριμένο key.*