Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Εργασία 2019-2020

Μέλη ομάδας:
Δημάκης Βασιλειος : 1115201400324
Δεωνάς Παύλος : 1115201500033

- **Λειτουργικότητα**

Σε γενικές γραμμές η εφαρμογή μας αποτελεί εναν εκτελεστή sql ερωτημάτων με λιγο πιο περιορισμένα εργαλεία απο οτι ενας πραγματικος εκτελεστής.
Για την ακρίβεια εκτελουνται ερωτήματα των οποίων τα κατηγορήματα(predicates) ειναι κυρίως φιλτρα της μορφής (R.A (>,<,=) X ,οπου X ένας ακέραιος αριθμός και Join queries.
Εδικότερα διαβάζουμε απο καποια δυαδικά αρχεία ενα σύνολο σχέσεων οι οποίες αποτελούνται απο κολώνες με εγγραφές 64-bit απρόσημων ακεραίων και περνάμε τις σχέσεις αυτές στη μνήμη ενω παράλληλα κρατάμε τα δεδομένα αυτα σε εναν πίνακα απο struct relations, struct το οποίο κρατάει των αριθμό των κολωνών,τον αριθμό των εγγραφών καθως και τις εγγραφές.Αφου κρατήσουμε αυτα τα δεδομένα υπολογίζουμε και κάποια στατιστικά για την καθε κολώνα της καθε σχέσης
όπως την μικρότερη τιμή,την μεγαλύτερη τιμή,των πλήθος των τιμών και το πλήθος των μοναδικών τιμών της σχέσης.
Στην συνέχεια διαβάζουμε απο ενα άλλο αρχείο καποια queries πάνω στις σχέσεις που εχουμε ηδη κρατήσει και τα εκτελόυμε.

<u>Ανάλυση και Εκτέλεση των queries:</u>

Το αρχείο με τα queries ειναι χωρισμένο σε πακέτα απο queries και εμεις υπολογίζουμε το ενα πακέτο με βαση τα κατηγορήμτα του και κρατάμε καποια δεδομένα τα οποία θα εξηγήσω αργότερα αναλυτικότερα για να τα εκτυπώσουμε στο τέλος καθε πακέτου.
Καθε query είναι της μορφής: 3 0 1|0.2=1.0&0.1=2.0&0.2>3499|1.2 0.1
με το ‘|’ να χωρίζει τις σχέσεις που θα πάρουνε μέρος στο ερώτημα αυτό τα κατηγορήματα,και τις κολώνες απο τις σχέσεις που θα προβληθούν στο τέλος αφου εκτελεστύν και τα κατηγορήματα.
Για κάθε query κανουμε μια ανάλυση σχετικα με το ποιες σχεσεις εμπεριέχει,ποια είναι τα κατηγορήμτα,και ποιες κολώνες ποιανων σχεσεων θ εκτυπωθούνε στο τέλος.Στην συνέχεια υπολογίζουμε την προτεραιότητα με βαση την οποία θα εκτελεστούν τα κατηγορήμτα.

Για την ακρίβεια τα κατηγορήματα τα οποια εχουν μορφή φιλτρου οπως εξηγησα παραπάνω εκτελούνται πάντα πρωτα διοτι ειναι σιγουρο οτι θα μειώσουν τις εγγραφές που θα κρατήσουμε απο τη κολώνα στην οποία εκτελείται το φιλτρο, πράγμα το οποίο μας βολεύει μακροπρόθεσμα διοτι αν εχουμε και ενα κατηγόρημα JOIN στο οποίο εμπεριέχεται η ίδια κολωνα οι εγγραφές που θα παρουν μέρος στο JOIN θα είναι λιγοτερες και αυτο είναι θετικό διοτι η JOIN ειναι χρονοβόρα διαδικασία και οσο λιγοτερες εγγραφες παιρνουν μερος τοσο το καλύτερο.Τωρα οσο αναφορα τα JOIN κατηγορήματα προσπαθούμε να βρούμε το καλύτερο μονοπάτι εκτέλεσης τους με τη βοήθεια των στατιστικών που κρατήσαμε.Καλύτερο μονοπάτι εννοώ το πιο γρήγορο σε χρόνο και με την μικρότερη δέσμευση μνήμης.Ουσιαστικα προσπαθούμε να υπολογίσουμε ποια JOIN κατηγορήματα παράγουν τα λιγότερα αποτελέσματα και με αυτη τη σειρά τα εκτελούμε.

Εκτέλεση των κατηγορημάτων:

Με βάση την προτεραιότητα των κατηγορημάτων που ανέφερα παραπάνω ε κτελούμε τα κατηγορήματα τα οποια οπως προανέφερα μπορουν να είναι ειτε φίλτρα είτε JOIN.
Για καθε κατηγόρημα που εκτελείται τα αποτελέσματα που προκύπτουν απο αυτο τα κρατάμε σε μια Ενδιάμεση Δομή (Intermediate Result), γιατι μπορει ας πουμε να υπαρξει ενα join στο οποιο οι οι σχέσεις που συμμετέχουν σε αυτο να εχουν συμμετάσχει σε προηγούμενα κατηγορήματα(φιλτρα ή join) ,αρα οι εγγραφές που θα πρέπει να φορτώσουμε για να εκτελεουμε το τρέχον κατηγόρημα θα είναι αυτές που θα βρίσκονται μεσα στην ενδιάμεση δομή, αυτες δηλαδη που έμειναν απο προηγούμενα κατηγορήματα.Θα υπάρξει αναλυτικότερη εξήγηση σχετικά με την λειτουργία της ενδιάμεσης δομής παρακάτω.

Φίλτρα : Εδω χρησιμοποιούμε ενα struct column_data το οποιο αποτελείται απο ενα πινακα απο struct tuple(εγγραφές) και το πλήθος αυτων για να φορτώσουμε την κολώνα στην οποία θα εφαρμόσουμε το φίλτρο και αναλόγως αν το φίλτρο ειναι < , > ,= κάνουμε την αντίστοιχη αναζήτηση μεσα στην κολώνα με βάση τον αριθμό που μας δίνεται (π.χ. 1.0=32432 , θα κτατήσουμε μονο τις τιμές που ειναι ισες με το 32432 της κολώνας 0 απο τη σχέση 1) και καταγράφουμε τα αποτελέσματα στην Ενδιάμεση Δομή

JOIN : π.χ. 0.2=1.0. Χρησιμοποιώντας παλι το struct column_data (για 2 κολώνες αυτη τη φορα) φορτώνουμε απο την μνήμη σε ενα column_data τη κολώνα 2 της σχέσης 0 και σε ενα αλλο colunm_data τη κολώνα 0 της σχέσης 1. Στην συνέχεια για να μειώσουμε οσο περισσότερο τον χρόνο που διαρκεί ενα JOIN διοτι μιλαμε για κολώνες με αρκετές εγγραφές ταξινομούμε τις εγγραφές και απο τις 2 κολώνες με την μέθοδο Sort και μετα κάνουμε το JOIN για τις 2 κολώνες με την μέθοδο Join.
Σχετικά με τις μεθόδους Sort και Join θα υπάρξει αναλυτική επεξήγηση παρακατω.

Προβολές:

π.χ. 1.2 0.1

Με βαση το παράδειγμα πρέπει να προβάλουμε τις εγγραφες που εχουν απομείνει μετα την εκτέλεση των κατηγορημάτων στις κολώνες 2 και 1 των σχέσεων 1 και 0 αντίστοιχα.Αυτο όμως που εκτυπώνεται ακριβώς είναι το άθροισμα των τιμών των εγγραφών που εχουν απομείνει στις κολώνες.
Για την επίτευξη αυτου δημιουργούμε εναν πίνακα all_sums στον οποίο κρατάμε τα αθροίσματα.Ουσιαστικά επειδη οπως αναφέρθηκε παραπάνω τα queries τα δεχόμαστε σε μορφή πακέτων, κρατάμε στον πινακα τα αθροίσματα για καθε query και στο τέλος του πακέτου μολις δούμε στο αρχείο την εντολή F(Flush) εκτυπώνουμε τα αθροίσματα που εχουμε κρατησει στον πινακα all_sums για καθε query με την σειρά που δόθηκαν τα queries.

Αυτή η διαδικασία που αναφέρθηκε (ανάλυση κατηγορμάτων,εκτέλεση κατηγορημάτων,προβολές) σε πρώτη φαση δηλαδή μεχρι το 2ο part έτρεχε μονονηματικά.Στην πορεία προσθέσαμε λειτουργίες πολυνηματισμού με σκοπό την μειωση χρονου και την παραλληλοποίηση του προγράμματος.
Σχετικα με την παραλληλοποίηση:

Η παραλληλοποιηση μας γινεται σε επιπεδο query. Δηλαδη τα threads μας δημιουργουνται για την παραλληλη εκτελεση πολλαπλων query μεσα σε batch. Η υλοποιηση αυτου εγινε με την δημιουργια αρχικα του threadpool ,την δημιουργια των threads ,οπως και την δημιουργια μιας ουρας που ειχε τον ρολο το job scheduler. Τα threads απο την αρχικοποιηση τους περιμενουν να μπει καποια εργασια στην ουρα για να την εκτελεσουν.Η εργασια αυτη ειναι καποιο query .Στην πραξη, καθε φορα που διαβαζουμε μια γραμμη απο το αρχειο .work που μας εχει δοθει προσθετουμε με την AddJob στην ουρα μας ,την συναρτηση που θελουμε ,μαζι με τα arguments που χρειαζονται ωστε να μπορει να την εκτελεσει καποιο thread. Τα threads απο την δημιουργια τους περιμενουν τα καταλληλα signals, ετσι ωστε να “ξεκλειδωθουν” και να μπορεσουν να παρουν “δουλεια ” μεσα απο την ουρα μας. (Παραδειγματα signals ειναι : αν η ουρα μας ειναι αδεια,αν ειναι καποιο αλλο thread μεσα στην ουρα για να παρει δουλεια κ.ο.κ). Αφου τα threads παιρνουν δουλεια απο την ουρα μας και εκτελουν τα queries παραλληλα εχει δημιουργηθει και μια συναρτηση με ονομα barrier που περιμενει μεχρι , ειτε ολα τα threads να τελειωσουν με τις δουλειες τους ειτε να αδειασει η ουρα (δηλαδη το τελος καποιου batch).Οταν αδειασει η ουρα με τις δουλειες των threads γινονται οι εκτυπωσεις των αποτελεσματων και το προγραμμα προχωραει στο επομενο batch με την ιδια ακριβως διαδικασια.
Τελος, υπαρχει συναρτηση destroy η οποια καταστρεφει τα threads μας καθως και απελευθερωνει ολη την μνημη που δεσμευσαμε για το threadpool και για την ουρα του job scheduler.

Sort:

Για την ταξινόμηση των σχέσεων ακολουθήσαμε τον τρόπο που περιγράφηκε στο μάθημα.Δηλαδή παίρνουμε την καθε σχέση της οποιας τις τιμες κραταμε σε εναν πινακα ,ταξινομούμε για το πρώτο byte με βαση το key καθε σχεσης και τα γράφουμε σε έναν αλλο πίνακα κρατώνταςδείκτες στον πινακα για το που ξεκινάνε και για το που τελειώνουνε οι εγγραφές που σαν 1ο byte έχουν 0,1,2,3 κ.ο.κ. Στην συνέχεια ταξινομούμε με βαση το 2ο byte και περναμε τις τιμές στον 1ο πίνακα κρατώντας παλι δείκτες οπως παραπάνω.Αυτο γινεται μεχρι η καθε ομάδα εγγραφών (ομαδα εννοω σχετικα με τους δεικτες που κραταμε πανω στον πινακα) να περιέχει λιγότερες εγγραφές απο 64KB .Όταν φτασουμε σε αυτο το σημείο καλουμε την συναρτηση quicksort(περιγράφεται παρακάτω) για καθε ομαδα και τοτε εχουν ταξινομηθει ολες οι εγγραφες. Ολη αυτη η διαδικασία υλοποιείται με την συναρτηση :
column_data Sort(column_data)

Για την ταξινομηση με quicksort εχει δημιουργηθει μια συναρτηση quicksort μια συναρτηση partition και μια συναρτηση swap. Στις συναρτησεις quicksort και partition στελνουμε τον πινακα μας μαζι με δυο αριθμους low , high, που χρησιμοποιουμε για να ξερουμε ποια στοιχεια του πινακα πρεπει να ταξινομησουμε.

Join:

Για την εύρεση των τελικών αποτελεσμάτων έχει φτιαχτεί μια συνάρτηση Join η οποία παιρνει σαν ορισματα τους δυο μας ταξινομημένους πίνακες. Η δ ιαδικασια που ακολουθεί ειναι αρχικα να διατρεχει τον εναν απο τους δυο πινακες (τον πιο μικρο σε μεγεθος - απο επιλογη-) και να
προχωραει στον αλλο μεχρι να βρει καποιο match. Αν βρεθει, γινεται εισαγωγη στην λισταυξανεται ενας μετρητης matches και ενας ακομα num_of_matches και προχωραει ο δεικτης μαςστο επομενο στοιχειο του δευτερου πινακα. Χρησιμοποιειται μια μεταβλητη η οποια αυξανεται καθε φορα που γινεται καποιο join για να μπορουμε στην περιπτωση που εχουμε διπλοτυπα στον πινακα τον οποιο εχουμε επιλεξει να διατρεξουμε, να ξερουμε ακριβως που θα βρουμε τα αποτελεσματα μας και να μην χρειαζεται καθε φορα να διατρεχουμε και τον δευτερο πινακα για να βρισκουμε αποτελεσματα. Επομενως το matches μηδενιζεται καθε φορα που αλλαζει το στοιχειο απο τον πρωτο πινακα. Το num_of_matches ειναι ενας μετρητης ο οποιος μας κραταει τα συνολικα joins που γινονται.Η λιστα τωρα ειναι μια απλη λιστα οπου κραταμε δυο δεικτες στην αρχη της και στο τελος της. Σε καθε κομβο της λιστας κραταμε τα payloads που μας ενδιαφερουν ανα δυαδες, ακριβως οπως γινονται τα matches

και εναν δεικτη για τον επομενο κομβο. Η εισαγωγη στη λιστα γινεται μεχρι να γεμισει καθε κομβος, δηλαδη να φτασει το 1MB και σε περιπτωση που πρεπει να φτιαχτει καποιος καινουργιος κομβος για να κερδισουμε χρονο αξιοποιησαμε τον δεικτη της λιστας που δειχνει καθε φορα στο τελος της. Ετσι δεν χρειαζοταν να διατρεχουμε ολη την λιστα και ηταν πιο ευκολο και γρηγορο το να δημιουργουμε κομβους και να κανουμε την εισαγωγη των αποτελεσματων μας.
Υπαρχουν και συναρτησεις εκτυπωσεις της λιστας αλλα και απελευθερωσης της απο τη μνημη.
(PrintResults(), freelist()).

Ενδιαμεση Δομη:

Η Ενδιαμεση Δομη μας αναπαρισταται απο ενα struct (Intermidiate_Result) το οποιο αποτελειται απο εναν μονοδιαστατο πινακα και απο δυο δυσδιαστατους πινακες. Η χρησμιοτητα του μονοδιαστατου πινακα ειναι για να κραταμε τον αριθμο των στοιχειων που εχουμε μεσα στην ενδιαμεση δομη για καθε μια σχεση. Αρχικοποιειται με -1 που σημαινει οτι ειναι αδεια η ενδιαμεση για αυτη τη σχεση. Αυτο μας βοηθαει πολυ καθως καθε φορα που κανουμε ενα join/φιλτρο ελεγχουμε αν υπαρχουν στοιχεια μεσα στην ενδιαμεση μας ,και σε περιπτωση που η τιμη αυτη ειναι διαφορη απο -1 ή 0 παιρνουμε τα στοιχεια απο την ενδιαμεση,αλλιως διαβαζουμε ολα τα στοιχεια της κολωνας. Τα στοιχεια μεσα στην ενδιαμεση γραφονται στον ενα δυσδιαστατο πινακα. Ο πινακας αυτος ειναι της μορφης ResArray[relation][index]. Το index ειναι ο δεικτης μου καθε φορα σε στοιχειο του πινακα οπου μεσα υπαρχουν τα payloads τα οπιοια εχουν κανει καποιο match. Στον αλλο δυσδιαστατο πινακα κραταμε ποιες σχεσεις συσχετιζονται μεταξυ τους. Η χρηση αυτου του πινακα ειναι για να ξερουμε καθε φορα που γινεται καποιο join αν υπαρχουν σχεσεις που επηρεαζονται απο αυτο χωρις να συμμετεχουν. Αυτο μας βοηθαει για να κανουμε update στην ενδιαμεση δομη μας καθε φορα μετα απο καποιο join για ολες τις σχεσεις,γι αυτο το λογο πριν προχωρησουμε στο επομενο join αλλαζουμε αυτον τον πινακα και κραταμε οτι καποιες σχεσεις μεταξυ τους συχεστιζονται. Το update της ενδιαμεσης δομης μετα απο καποιο join ,γινεται στην συναρτηση JoinUpdate. Σε αυτη τη συναρτηση στελνουμε την ηδη υπαρχουσα ενδιαμεση δομη, τη λιστα με τα καινουργια αποτελεσματα , το συνολικο αριθμος αυτων των αποτελεσματων, τις δυο μας σχεσεις καθως και το συνολικο αριθμο των σχεσων που συμμετεχουν σε αυτο το query. Αρχικα να σημειωθει οτι μετα απο καποιο join η JoinUpdate συναρτηση καλειται δυο φορες, μια φορα φορα για καθε σχεση που υπηρχε στο join. Στην αρχη αυτης της συναρτησης κανουμε εναν ελεγχο για να δουμε αν η σχεση μας εχει ηδη καποια στοιχεια μεσα . Σε περιπτωση που δεν υπαρχουν στοιχεια αυτης της σχεσης μεσα στην ενδιαμεση δομη μου αυτο που κανω, ειναι απλα να προσθεσουμε τα στοιχεια του join για αυτη τη σχεση μεσα στην ενδιαμεση δομης μας στην καταλληλη θεση. Σε περιπτωση ομως που ηδη υπαρχουν

στοιχεια μεσα στην ενδιαμεση γινονται τα εξης: Αρχικα Δημιουργουμε μια νεα ενδιαμεση δομη , μεσα στην οποια για την σχεση που εχουμε αυτη τη στιγμη στη συναρτηση γραφουμε τα στοιχεια της οπως μας ηρθαν απο τη λιστα μετα το join,στην συνεχεια αντιγραφουμε για ολες τις αλλες σχεσεις τα στοιχεια που υπαρχουν μεσα στην αρχικη μας ενδιαμεση δομη (αν υπαρχουν) και ανενωνουμε ολες μας της μεταβλητες-πινακες.
Πριν αποδεσμευσουμε ομως την αρχικη μας ενδιαμεση θα πρεπει να δουμε αν ειναι αναγκαιο να κανουμε update και σε αλλη σχεση μεσα στην ενδιαμεση η οποια συσχετιζεται με την σχεση που εχουμε αυτη τη στιγμη μεσα στην συναρτηση μας. Ελεγχουμε λοιπον αν η σχεση μας εχει καποι συσχετιση με αλλες σχεσεις. Σε περιπτωση που δεν εχει,αποδεσμευουμε την αρχικη μας ενδιαμεση και επιστρεφουμε την νεα μας. Σε περιπτωση ομως που υπαρχουν συσχετισεις με την σχεση μας , θα πρεπει στην νεα ενδιαμεση να αλλαξουμε τα στοιχεια της συσχετισμενης σχεσης και να τα τοποθετησουμε στην νεα ενδιαμεση. Για να γινει αυτο, κοιταζουμε παραλληλα δυο πινακες, ο ενας ειναι αυτος με τα καινουργια αποτελεσματα και ο αλλος με τα παλια αποτελεσματα της σχεσης μας. Η διαδικασια που ακολουθουμε μοιαζει με αυτη της Join καθως παλι ψαχνουμε ομοιους αριθμους μεσα σε δυο πινακες με τον ιδιο τροπο και εκμεταλλευομαστε το γεγονος οτι τα στοιχεια της συσχετισμενης σχεσης με την σχεση μας , ειναι παραλληλα. Καθε φορα λοιπον βλεπουμε ενα στοιχειο της σχεσης μας απο τον ενα πινακα(νεα ενδιαμεση) και αναζητουμε το ιδιο στον αλλο πινακα (παλια ενδιαμεση).Σε περιπτωση που το βρουμε στη νεα ενδιαμεση αντιγραφουμε το παραλληλο στοιχειο για την συσχετισμενη σχεση μας. Αυτο ολοκληρωνεται οταν ολα τα στοιχεια της συσχετισμενης σχεσης γραφτουν στην νεα ενδιαμεση. Για να γλυτωσουμε καποιες επιπλεον αναζητησεις στον πινακα της παλιας ενδιαμεσης πειραζουμε καποιες μεταβλτητες οι οποιες μας δινουν την δυνατοτητα να ελεγχουμε συγκεκριμενα στοιχεια του πινακα αντι για ολα.. Στο τελος αυτης της διαδικασιας εχουμε πλεον ολοκληρωσει την ανανεωση της ενδιαμεσης δομης μας και απελευθερωνουμε την παλια ενδιαμεση καθως και οτι αλλο ειναι αναγκαιο.
Στην ενδιαμεση δομη επιπλεον γραφουμε πραγματα και μετα απο καποιο φιλτρο. (>,<,=)
Αυτο γινεται στην συναρτηση FilterUpdate. Η συναρτηση αυτη λειτουργει ακριβως οπως λειτουργει και η JoinUpdate με την μονη διαφορα , οτι επειδη τα φιλτρα προηγουνται ολων, δεν χρειαζεται να κανουμε ελεγχο για συσχετισμενες σχεσεις καθως και ολη τη διαδικασια που κανουμε στην JoinUpdate. Αρκει λοιπον να γραψουμε στην ενδιαμεση μας τα στοιχεια που παιρνουμε απο την εφαρμογη του φιλτρου για την σχεση που το εφαρμοσαμε.

<u>Εφαρμογη των φιλτρων σε μια σχεση(>,<,=) :</u>
Ολα τα φιλτρα εφαρμοζονται στην συναρτηση Equalizer η οποια δεχεται την κολωνα της σχεσης που μας ενδιαφερει να φιλτραρουμε καθως και το mode (>,<,=) . Η συναρτηση διατρεχει ολο τον πινακα και βρισκει τα στοιχεια που ικανοποιουν τη συνθηκη και τα τοποθετει σε εναν πινακα.

Ο πινακας αυτος στην συνεχεια στελνεται στην ενδιαμεση δομη και γινεται η παραπανω διαδικασια.

Στο τέλος του προγράμματος αλλα και κατα τη διάρκεια της εκτέλεσης απελευθερωνουμε την μνήμη που δεσμεύουμε. Οταν τελείωσει το πρόγραμμα δεν υπάρχει καποιο leak στη μνήμη.Έχει ελεγθεί με valgrind

● Συναρτήσεις

Ακολουθεί μια εξήγηση όλων των συναρτήσεων που εχουν υλοποιηθεί.

- *Intermediate_Result* create_Intermediate_Result(int)* : Δημιουργία Ενδιαμεσης Δομης
- *Intermediate_Result* FilterUpdate (Intermediate_Result*, int ,uint64_t *, int ,int):* Ενημέρωση της ενδιαμεσης δομης μετα απο την εκτελεση ενος φιλτρου

- *Intermediate_Result* JoinUpdate (Intermediate_Result* , int , Result* ,int , int , int ,int):* Ενημέρωση της ενδιαμεσης δομης μετα απο την εκτελεση ενος join

  *uint64_t *Intermediate_Sum(Intermediate_Result* ,relation* ,int *, char* ,int):* Υπολογισμός αθροισμάτων για ενα query

- *void queries_analysis(char * ,relation*,int,struct statistics*,thread_pool *):* Ανάλυση των queries

- *struct Predicates* predicates_analysis(int,char *,relation*,int *):*Ανάλυση των κατηγορημάτων

- *int * predicates_priority(int,struct Predicates *):* Υπολογισμος προτεραιότητας για τα κατηγορήμτα ενος query

- *int * Join_Enumeration(relation*,int ,struct Predicates *,int *):* Υπολογισμος προτεραιότητας για τα κατηγορήμτα ενος query με βαση τα στατιστικά και τον αλγοριθμο join enumeration.

- *void count_statistics(relation * ,int * ,struct Predicates *, int,int):* Υπολογισμος στατιστικών για μια σχέση με βαση καποιο κατηγόρημα

- *Intermediate_Result * exec_predicates(relation *,struct Predicates *,int * ,int, int,int *):* Εκτέλεση κατηγορημάτων

- *void reset_statistics(relation*,struct statistics * ,char *):* Επαναφορα των στατιστικών στην πρβτη τους μορφη

- *Result* ListInit():* Δημιουργια λιστας αποτελεσμάτων για ενα join μεταξυ 2 σχάσεων

- *void InsertResult(uint64_t,uint64_t,Result*):* Εισαγωγή των αποτελεσμάτων στη λιστα

- *void PrintResults(Result*):* Εκτυπωση των αποτελεσμάτων που υπάρχουν στη λίστα

- *void freelist(Result*):* Αποδέσμευση μνήμης των κόμβων της λίστας και των δεδομένων της

- *relation * read_file(char*,int *,struct statistics**):* Διάβασμα αρχειου που περιέχει τις σχέσεις με τις εγγραφες.Φόρτωση των σχέσεων στη μνήμη.Αρχικοποίηση και Υπολογισμός στατιστικών για τις κολώνες πριν οποιοδηποτε κατηγόρημα τα οποια γενικα αλλάζουν τα στατιστικά

- *uint64_t * loadRelation(char*):* Φότρωση των σχέσεων στη μνημη

- *void swap(uint64_t * a, uint64_t * b):*
- *int partition (column_data, int , int)¨:*
- *void quickSort(column_data, int , int):* Υλοποίηση quicksort.Χρησιμοποιεί τις 2 παραπάνω συναρτήσεις

- *column_data Sort(column_data):* Ταξινόμηση μιας κολώνας μια σχέσης.Χρησιμοποιεί και την παρακάτω συνάρτηση

- *void sorting (column_data *,column_data *,int, int , int , int):*

- *Result * Join(column_data , column_data,int *):* Υλοόποίηση merge-join

- *column_data load_column_data(relation *, int rel,int col):* Φόρτωση εγγραφών μιας κολώνας απο την μνήμη σε ενα struct column_data

- *column_data load_from_IR(relation * ,int ,int ,uint64_t ,uint64_t *):* Φόρτωση εγγραφών μιας κολώνας απο την ενδιάμεση δομή σε ενα struct column_data

- *Result * scan(column_data ,column_data ,int *):* Σάρωση μιας κολώνας στην περίπτωση που έχουμε κατηγόρημα με μορφή Join αλλα στην ίδια κολώνα της ιδιας σχέσης

- *uint64_t * Equalizer(column_data ,int ,int ,int *):* Υλοποίηση φίλτρων

- *void ThreadJob(thread_pool *):* Καλεί την get_job για να δώσει " δουλειά " στα threads να κάνουν

- *job * thread_pool_get_job(thread_pool *):* Δίνει " δουλειά " σε ενα διαθέσιμο thread απο το threadpool

- *thread_pool * thread_pool_init(int):* Δημιουργία threadpool

- *void thread_pool_add_job(thread_pool * ,void (*function)(void* arg),void *):* Εισαγωγή " δουλειάς " στην ουρά

- *queue * queue_init():* Δημιουργία ουράς για τις "δουλειές" των threads

- *void thread_pool_barrier(thread_pool *):* Περιμένει να τελειωσουν την δουλεια τους ολα τα threads τα οποια εχουνε παρει δουλειες απο την ουρα των jobs.Ουσιαστικα χρησιμοποιείται ετσι ωστε να εκτελείται πρώτα το ενα batch για να εκτυπωθούν τα αθροίσματα και να συνεχίσουμε με το αλλο batch

- *void thread_pool_destroy(thread_pool *):* Αποδέσμευση μνήμης του threadpool και των δεδομένων του και καταστροφή όλων των mutexes και των conditional variables

● Δομές Δεδομένων

```
struct statistics
{
   uint64_t * min;
   uint64_t * max;
   uint64_t * number;
   uint64_t * distinct;
   int ** dis_vals;
};
```

Στατιστικά για καθε κολώνα καθε σχέσης.Μικρότερη τιμη, μεγαλύτερη τιμή , πλήθος εγγραφών , διακριτές τιμές.

```
struct Predicates{
    int relation1;
    int colum1;
    int relation2;
    int colum2;
    int num;
    char op;
    int prio;
};
```

Πληροφορίες για καθε κατηγόρημα.Δηλαδη ποιες σχεσεις συμμετέχουν ποιες κολώνες των σχέσεων αυτών συμμετέχουν, ποια είναι η πράξη που γίνεται ,ποιος ειναι ο αριθμός σε περίπτωση που το κατηγόρημα είναι φίλτρο και ποια ειναι η προτεραιότητα του κατηγορήματος αυτου.

```
struct relation
{
    uint64_t num_tuples;
    uint64_t num_columns;
    uint64_t * data;
    struct statistics stats;

};
```

Πληροφορίες για καθε σχέση που διαβάζουμε απο τα αρχεία.Ποσες κολώνες έχουν,ποσες εγγραφές έχει η καθε κολώνα, όλες τις εγγραφές σειριακά στη μνήμη και τα στατιστικά για τις κολώνες τις σχέσεις αυτες

```
struct tuple
{
    uint64_t key;
    uint64_t payload;
};
```

Μια εγγραφή που έχει ενα αναγνωριστικο id και μια τιμή

```
struct hist
{
    unsigned binary;
    unsigned count;
};
```

Αυτο το struct χρησιμοποιείται απο την Sort μας για να κατηγοριοποιεί τις εγγραφέσμε βαση τα byte τους οπως μας ζητείται

```
struct column_data
```

```
{
   struct tuple * tuples;
   uint64_t num_tuples;
};
```

Αυτο το struct το χτησιμοποιούμε για να φορτώνουμε απο το data της δομής relation τις εγγραφές μιας κολώνας

```
struct ResultNode
{
   uint64_t buffer[SIZE_NODE][2];
   int counter;
   struct ResultNode *next;
};
```

Κόμβος λίστας με αποτελέσματα μετα απο ενα join κατηγόρημα

```
struct Result{
   ResultNode *first;
   ResultNode *current;
};
```

Δείκτες στον πρώτο κόμβο και στο τελευταίο της λίστας που αναφέρθηκε απο πανω

```
struct Intermediate_Result
{
   uint64_t * relResults; ///posa apotelesmata exei to tade relation
   uint64_t ** resArray;  ///ta apotelesmata
   uint64_t ** Related_Rels;
};
```

Ενδιάμεση δομή με αποτελέσματα εγγραφών μετα απο την εκτέλεση των κατηγορημάτων

```
struct job
{
   void   (*function)(void* arg);
   void * arg;
   struct job * next;
} job;
```

Η “ δουλειά ” των threads

```
typedef struct queue
{
```

```
    job *first;
    job *last;
    int n_jobs;
};
```

Ουρά με δουλειές για τα threads

```
struct thread_pool
{
    pthread_t ** threads;
    int * threads_ids;
    int jobs_to_done;
    int jobs_done;
    int threads_alive;
    int threads_working;
    queue * job_queue;
    pthread_mutex_t empty_queue_mtx;
    pthread_cond_t empty_queue_cond;
    pthread_mutex_t barrier_mtx;
    pthread_cond_t barrier_cond;
    pthread_mutex_t alive_mtx ;
    pthread_cond_t alive_cond;
    pthread_mutex_t job_queue_mtx;

};
```

Αναπαράσταση του threadpool με πληροφορίες οπως τα ιδια τα threads τα id τους, με ακεραίους αριθμους που μας πληροφορούν καθε στιγμη για το ποσα threads δουλεύουν ,ποσα ειναι ζωντανά, πόσα δουλειές έχουν τελείωσει ,ποσες απομένουν. Επίσης η ουρά με τις “δουλειές” και φυσικα καποια mutexes και καποιες conditional variables που χρησιμοποιούνται για τον συγχρονισμό των threads.

```
struct args
{
    char * line;
    int rels;
    relation * relations;
    struct statistics * original;
    int Sums_count;
    uint64_t **all_sums;
    uint64_t *shows;
    pthread_mutex_t mutex;
};
```

Ορίσματα που χρειάζεται το καθε thread για να κάνει την δουλειά που του έχουμε αναθέσει

- **Χρόνοι εκτέλεσης και μνήμη που δεσμέυτηκε για την εκτέλεση του προγράμματος:**

π.χ. για το small dataset

Threads	Χρόνος Εκτέλεσης (sec)	Μνήμη
1	5,632	1,299,775,117 bytes ≈ 1,2 GB
2	2,230	1,299,775,433 bytes ≈ 1,2 GB
3	2,207	1,299,775,749 bytes ≈ 1,2 GB
4	2,200	1,299,776,065 bytes ≈ 1,2 GB
5	2,228	1,299,776,381 bytes ≈ 1,2 GB
6	2,240	1,299,776,697 bytes ≈ 1,2 GB
7	2,245	1,299,777,013 bytes ≈ 1,2 GB
8	2,233	1,299,777,329 bytes ≈ 1,2 GB

Όπως βλέπουμε τον καλύτερο χρόνο εκτέλεσης τον πατυχαίνουμε με 4 threads και εφοσον οι διαφορές με τα υπόλοιπα ειναι πολυ μικρές , επιλέξαμε το threadpool που δημιουργούμε να αποτελείται απο 4 threads.

Ακολουθεί πίνακας με χρόνους εκτέλεσης του προγράμματος πριν και μετα την προσθήκη της λειτουργίας του πολυνηματισμού

	Single-threaded	Multi-threaded
Small Dataset	5,768 sec	2,200 sec
Medium Dataset	9 min 23,345 sec	6 min 23,256 sec

Οι χρόνοι αυτοί μετρήθηκαν με την εντολη time και οι χρονοι με valgrind σε μηχάνημα με λειτουργικό Linux, επεξεργαστή Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz , RAM 8 GB και το πρόγραμμα μεταγλωττίστηκε gcc 7.4.0

- **Μεταγλώττιση και Εκτέλεση του προγράμματος**

Για την μεταγλώττιση του προγράμματος εχει υλοποιηθεί κατάλληλο Makefile οποτε αρκείη εντολή : make
Στο Makefile μετταγλωτίζεται καθε sousce αρχείο μόνο του ετσι ώστε να δημιουργηθούν τα κατάλληλα object files και τελος δημιουργείται το executable αρχείο απο ολα τα εκτελέσιμα.
Επίσης με make clean διαγράφονται τα object αρχεία και το executabe
Για την εκτέλεση του προγράμματος αρκει η εντολή : ./main <file1> <file2>
,οπου

file1 : small/small.init ή medium.medium.init
file2 : small/small.work ή medium/medium.work