

# Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

**Εργασία 2019-2020**

Μέλη ομάδας:

Δημάκης Βασίλειος : 1115201400324

Δεωνάς Παύλος : 1115201500033

## ● Λειτουργικότητα

Σε γενικές γραμμές η εφαρμογή μας αποτελεί έναν εκτελεστή sql ερωτημάτων με λίγο πιο περιορισμένα εργαλεία από ότι ένας πραγματικός εκτελεστής.

Για την ακρίβεια εκτελούνται ερωτήματα των οποίων τα κατηγορήματα(predicates) είναι κυρίως φίλτρα της μορφής (R.A (>,<=) X , όπου X ένας ακέραιος αριθμός και Join queries.

Εδிகότερα διαβάζουμε από κάποια δυαδικά αρχεία ένα σύνολο σχέσεων οι οποίες αποτελούνται από κολώνες με εγγραφές 64-bit απρόσημων ακεραίων και περνάμε τις σχέσεις αυτές στη μνήμη ενώ παράλληλα κρατάμε τα δεδομένα αυτά σε έναν πίνακα από struct relations, struct το οποίο κρατάει τον αριθμό των κολωνών, τον αριθμό των εγγραφών καθώς και τις εγγραφές. Αφού κρατήσουμε αυτά τα δεδομένα υπολογίζουμε και κάποια στατιστικά για την κάθε κολώνα της κάθε σχέσης όπως την μικρότερη τιμή, την μεγαλύτερη τιμή, των πλήθος των τιμών και το πλήθος των μοναδικών τιμών της σχέσης.

Στην συνέχεια διαβάζουμε από ένα άλλο αρχείο κάποια queries πάνω στις σχέσεις που έχουμε ήδη κρατήσει και τα εκτελούμε.

## Ανάλυση και Εκτέλεση των queries:

Το αρχείο με τα queries είναι χωρισμένο σε πακέτα από queries και εμείς υπολογίζουμε το ένα πακέτο με βάση τα κατηγορήματα του και κρατάμε κάποια δεδομένα τα οποία θα εξηγήσω αργότερα αναλυτικότερα για να τα εκτυπώσουμε στο τέλος κάθε πακέτου.

Κάθε 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 πάρουν μέρος στο ερώτημα αυτό τα κατηγορήματα, και τις κολώνες από τις σχέσεις που θα προβληθούν στο τέλος αφού εκτελεστούν και τα κατηγορήματα.

Για κάθε query κάνουμε μια ανάλυση σχετικά με το ποιες σχέσεις περιέχει, ποια είναι τα κατηγορήματα, και ποιες κολώνες ποιων σχέσεων θα εκτυπωθούν στο τέλος. Στην συνέχεια υπολογίζουμε την προτεραιότητα με βάση την οποία θα εκτελεστούν τα κατηγορήματα.

Για την ακρίβεια τα κατηγορήματα τα οποία έχουν μορφή φίλτρου όπως εξηγήσα παραπάνω εκτελούνται πάντα πρώτα διότι είναι σίγουρο ότι θα μειώσουν τις εγγραφές που θα κρατήσουμε από τη κολώνα στην οποία εκτελείται το φίλτρο, πράγμα το οποίο μας βολεύει μακροπρόθεσμα διότι αν έχουμε και ένα κατηγορήμα JOIN στο οποίο εμπεριέχεται η ίδια κολώνα οι εγγραφές που θα παρουν μέρος στο JOIN θα είναι λιγότερες και αυτό είναι θετικό διότι η JOIN είναι χρονοβόρα διαδικασία και όσο λιγότερες εγγραφές παίρνουν μέρος τόσο το καλύτερο. Τώρα όσο αναφορά 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. Ουσιαστικά προσπαθούμε να υπολογίσουμε ποια JOIN κατηγορήματα παράγουν τα λιγότερα αποτελέσματα και με αυτή τη σειρά τα εκτελούμε.

### Εκτέλεση των κατηγορημάτων:

Με βάση την προτεραιότητα των κατηγορημάτων που ανέφερα παραπάνω εκτελούμε τα κατηγορήματα τα οποία όπως προανέφερα μπορούν να είναι είτε φίλτρα είτε JOIN.

Για κάθε κατηγορήμα που εκτελείται τα αποτελέσματα που προκύπτουν από αυτό τα κρατάμε σε μια Ενδιάμεση Δομή (Intermediate Result), γιατί μπορεί ας πούμε να υπάρξει ένα join στο οποίο οι οι σχέσεις που συμμετέχουν σε αυτό να έχουν συμμετάσχει σε προηγούμενα κατηγορήματα (φίλτρα ή join), άρα οι εγγραφές που θα πρέπει να φορτώσουμε για να εκτελέσουμε το τρέχον κατηγορήμα θα είναι αυτές που θα βρίσκονται μέσα στην ενδιάμεση δομή, αυτές δηλαδή που έμειναν από προηγούμενα κατηγορήματα. Θα υπάρξει αναλυτικότερη εξήγηση σχετικά με την λειτουργία της ενδιάμεσης δομής παρακάτω.

Φίλτρα : Εδώ χρησιμοποιούμε ένα struct column\_data το οποίο αποτελείται από ένα πίνακα από struct tuple (εγγραφές) και το πλήθος αυτών για να φορτώσουμε την κολώνα στην οποία θα εφαρμόσουμε το φίλτρο και αναλόγως αν το φίλτρο είναι  $<$ ,  $>$ ,  $=$  κάνουμε την αντίστοιχη αναζήτηση μέσα στην κολώνα με βάση τον αριθμό που μας δίνεται (π.χ.  $1.0=32432$ , θα κρατήσουμε μόνο τις τιμές που είναι ίσες με το 32432 της κολώνας 0 από τη σχέση 1) και καταγράφουμε τα αποτελέσματα στην Ενδιάμεση Δομή

JOIN : π.χ.  $0.2=1.0$ . Χρησιμοποιώντας πάλι το struct column\_data (για 2 κολώνες αυτή τη φορά) φορτώνουμε από την μνήμη σε ένα column\_data τη κολώνα 2 της σχέσης 0 και σε ένα άλλο column\_data τη κολώνα 0 της σχέσης 1. Στην συνέχεια για να μειώσουμε όσο περισσότερο τον χρόνο που διαρκεί ένα JOIN διότι μιλάμε για κολώνες με αρκετές εγγραφές ταξινομούμε τις εγγραφές και από τις 2 κολώνες με την μέθοδο Sort και μετά κάνουμε το JOIN για τις 2 κολώνες με την μέθοδο Join.

Σχετικά με τις μεθόδους Sort και Join θα υπάρξει αναλυτική επεξήγηση παρακάτω.

## Προβολές:

π.χ. 1.2 0.1

Με βάση το παράδειγμα πρέπει να προβάσουμε τις εγγραφές που έχουν απομείνει μετά την εκτέλεση των κατηγορημάτων στις κολώνες 2 και 1 των σχέσεων 1 και 0 αντίστοιχα. Αυτό όμως που εκτυπώνεται ακριβώς είναι το άθροισμα των τιμών των εγγράφων που έχουν απομείνει στις κολώνες. Για την επίτευξη αυτού δημιουργούμε έναν πίνακα `all_sums` στον οποίο κρατάμε τα αθροίσματα. Ουσιαστικά επειδή όπως αναφέρθηκε παραπάνω τα queries τα δεχόμαστε σε μορφή πακέτων, κρατάμε στον πίνακα τα αθροίσματα για κάθε query και στο τέλος του πακέτου μόλις δούμε στο αρχείο την εντολή `F(Flush)` εκτυπώνουμε τα αθροίσματα που έχουμε κρατήσει στον πίνακα `all_sums` για κάθε query με την σειρά που δόθηκαν τα queries.

Αυτή η διαδικασία που αναφέρθηκε (ανάλυση κατηγορημάτων, εκτέλεση κατηγορημάτων, προβολές) σε πρώτη φάση δηλαδή μέχρι το 2ο part έτρεχε μονονηματικά. Στην πορεία προσθέσαμε λειτουργίες πολυνηματισμού με σκοπό την μείωση χρόνου και την παραλληλοποίηση του προγράμματος.

Σχετικά με την παραλληλοποίηση:

Η παραλληλοποίηση μας γίνεται σε επίπεδο query. Δηλαδή τα threads μας δημιουργούνται για την παραλληλή εκτέλεση πολλών query μέσα σε batch. Η υλοποίηση αυτού έγινε με την δημιουργία αρχικά του `threadpool`, την δημιουργία των threads, όπως και την δημιουργία μιας ουράς που είχε τον ρολό το `job scheduler`. Τα threads από την αρχικοποίηση τους περιμένουν να μπει κάποια εργασία στην ουρά για να την εκτελέσουν. Η εργασία αυτή είναι κάποιο query. Στην πράξη, κάθε φορά που διαβάζουμε μια γραμμή από το αρχείο `.work` που μας έχει δοθεί προσθέτουμε με την `AddJob` στην ουρά μας, την συνάρτηση που θέλουμε, μαζί με τα arguments που χρειάζονται ώστε να μπορεί να την εκτελέσει κάποιο thread. Τα threads από την δημιουργία τους περιμένουν τα κατάλληλα signals, έτσι ώστε να “ξεκλειδωθούν” και να μπορέσουν να πάρουν “δουλειά” μέσα από την ουρά μας. (Παραδειγματα signals είναι: αν η ουρά μας είναι άδεια, αν είναι κάποιο άλλο thread μέσα στην ουρά για να πάρει δουλειά κ.ο.κ.). Αφού τα threads παίρνουν δουλειά από την ουρά μας και εκτελούν τα queries παραλληλά έχει δημιουργηθεί και μια συνάρτηση με όνομα `barrier` που περιμένει μέχρι, είτε όλα τα threads να τελειώσουν με τις δουλειές τους είτε να αδειάσει η ουρά (δηλαδή το τέλος κάποιου batch). Όταν αδειάσει η ουρά με τις δουλειές των threads γίνονται οι εκτυπώσεις των αποτελεσμάτων και το πρόγραμμα προχωράει στο επόμενο batch με την ίδια ακριβώς διαδικασία.

Τέλος, υπάρχει συνάρτηση `destroy` η οποία καταστρέφει τα threads μας καθώς και απελευθερώνει όλη την μνήμη που δεσμεύσαμε για το `threadpool` και για την ουρά του `job scheduler`.

## Sort:

Για την ταξινόμηση των σχέσεων ακολουθήσαμε τον τρόπο που περιγράφηκε στο μάθημα. Δηλαδή παίρνουμε την κάθε σχέση της οποίας τις τιμές κρατάμε σε έναν πίνακα, ταξινομούμε για το πρώτο byte με βάση το key κάθε σχέσης και τα γράφουμε σε έναν άλλο πίνακα κρατώντας δείκτες στον πίνακα για το που ξεκινάει και για το που τελειώνει οι εγγραφές που σαν 1ο byte έχουν 0,1,2,3 κ.ο.κ. Στην συνέχεια ταξινομούμε με βάση το 2ο byte και περνάμε τις τιμές στον 1ο πίνακα κρατώντας πάλι δείκτες όπως παραπάνω. Αυτό γίνεται μέχρι η κάθε ομάδα εγγραφών (ομάδα εννοώ σχετικά με τους δείκτες που κρατάμε πάνω στον πίνακα) να περιέχει λιγότερες εγγραφές από 64KB. Όταν φτάσουμε σε αυτό το σημείο καλούμε την συνάρτηση quicksort (περιγράφεται παρακάτω) για κάθε ομάδα και τότε έχουν ταξινομηθεί όλες οι εγγραφές. Όλη αυτή η διαδικασία υλοποιείται με την συνάρτηση :

```
column_data Sort(column_data )
```

Για την ταξινόμηση με quicksort έχει δημιουργηθεί μια συνάρτηση quicksort μια συνάρτηση partition και μια συνάρτηση swap. Στις συναρτήσεις quicksort και partition στέλνουμε τον πίνακα μας μαζί με δύο αριθμούς low, high, που χρησιμοποιούμε για να ξέρουμε ποια στοιχεία του πίνακα πρέπει να ταξινομήσουμε.

## Join:

Για την εύρεση των τελικών αποτελεσμάτων έχει φτιαχτεί μια συνάρτηση Join η οποία παίρνει σαν ορίσματα τους δύο μας ταξινομημένους πίνακες. Η διαδικασία που ακολουθεί είναι αρχικά να διατρεχει τον έναν από τους δύο πίνακες (τον πιο μικρό σε μέγεθος - από επιλογή-) και να προχωράει στον άλλο μέχρι να βρει κάποιο match. Αν βρεθεί, γίνεται εισαγωγή στην λίστα αυξάνεται ένας μετρητής matches και ένας ακόμα num\_of\_matches και προχωράει ο δείκτης μας στο επόμενο στοιχείο του δεύτερου πίνακα. Χρησιμοποιείται μια μεταβλητή η οποία αυξάνεται κάθε φορά που γίνεται κάποιο join για να μπορούμε στην περίπτωση που έχουμε διπλοτυπία στον πίνακα τον οποίο έχουμε επιλέξει να διατρεχούμε, να ξέρουμε ακριβώς που θα βρούμε τα αποτελέσματα μας και να μην χρειάζεται κάθε φορά να διατρεχούμε και τον δεύτερο πίνακα για να βρισκούμε αποτελέσματα. Επομένως το matches μηδενίζεται κάθε φορά που αλλάζει το στοιχείο από τον πρώτο πίνακα. Το num\_of\_matches είναι ένας μετρητής ο οποίος μας κρατάει τα συνολικά joins που γίνονται. Η λίστα τώρα είναι μια απλή λίστα όπου κρατάμε δύο δείκτες στην αρχή της και στο τέλος της. Σε κάθε κομμάτι της λίστας κρατάμε τα payloads που μας ενδιαφέρουν ανά δυάδες, ακριβώς όπως γίνονται τα matches

και εναν δεικτη για τον επομενο κομβο. Η εισαγωγή 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 εισαγωγή των αποτελεσμάτων μας.

Υπαρχουν και συναρτησεις εκτυπωσης της λιστας αλλα και απελευθερωσης της απο τη μνημη.

(PrintResults(), freelist()).

### Ενδιαμεση Δομη:

Η Ενδιαμεση Δομη μας αναπαρισταται απο ενα struct (Intermediate\_Result) το οποιο αποτελείται απο εναν μονοδιαστατο πινακα και απο δυο δυσδιαστατους πινακες. Η χρησιμοτητα 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 οποια 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 αποτελεσμάτων, τις δυο μας σχεσεις καθως και το συνολικο αριθμο των σχεσεων 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

στοιχεία μέσα στην ενδιαμεση γίνονται τα εξής: Αρχικά Δημιουργούμε μια νέα ενδιαμεση δομη , μέσα στην οποία για την σχεση που έχουμε αυτη τη στιγμή στη συναρτηση γραφουμε τα στοιχεία της οπως μας ηρθαν απο τη λιστα μετα το join,στην συνεχεια αντιγραφουμε για ολες τις αλλες σχεσεις τα στοιχεία που υπαρχουν μέσα 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 μέσα στην ενδιαμεση η οποια συσχετιζεται με την σχεση που έχουμε αυτη τη στιγμή μέσα στην συναρτηση μας. Ελεγχουμε λοιπον αν η σχεση μας εχει καποι συσχετιση με αλλες σχεσεις. Σε περιπτωση που δεν εχει,αποδεσμευσουμε 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 στοιχεία 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 μέσα σε δυο πινακες με τον ιδιο τροπο και εκμεταλλευομαστε το γεγονος οτι τα στοιχεία 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 στοιχεία της συσχετισμενης σχεσης γραφουν στην νεα ενδιαμεση. Για να γλυτωσουμε καποιες επιπλεον αναζητησεις στον πινακα της παλιας ενδιαμεσης πειραζουμε καποιες μεταβλητες οι οποιες μας δινουν την δυνατοτητα να ελεγχουμε συγκεκριμενα στοιχεία 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 στοιχεία που παιρνουμε απο την εφαρμογη του φιλτρου για την σχεση που το εφαρμοσαμε.

#### Εφαρμογη των φιλτρων σε μια σχεση(>,<=) :

Ολα τα φιλτρα εφαρμοζονται στην συναρτηση Equalizer η οποια δεχεται την κολωνα της σχεσης που μας ενδιαφερε να φιλτραρουμε καθως και το mode (>,<=) . Η συναρτηση διατρεχει ολο τον πινακα και βρισκει τα στοιχεία που ικανοποιουν τη συνθηκη και τα τοποθετει σε εναν πινακα.

Ο πίνακας αυτός στην συνέχεια στέλνεται στην ενδιαμεση δομη και γίνεται η παραπάνω διαδικασία.

## ● Συναρτήσεις

Ακολουθεί μια εξήγηση όλων των συναρτήσεων που έχουν 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 κατηγορήματα ενος query
- *int \* Join Enumeration(relation\*,int ,struct Predicates \*,int \*)*: Υπολογισμος προτεραιότητας για τα κατηγορήματα 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 κατηγορημα
- *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(): Δημιουργία λίστας αποτελεσμάτων για ένα join μεταξύ 2 σχέσεων
- void InsertResult(uint64 t,uint64 t,Result\*): Εισαγωγή των αποτελεσμάτων στη λίστα
- void PrintResults(Result\*): Εκτύπωση 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 αρχείου που περιέχει τις σχέσεις με τις εγγραφές.Φόρτωση των σχέσεων στη μνήμη.Αρχικοποίηση και Υπολογισμός στατιστικών για τις κολώνες πριν οποιοδήποτε κατηγορήμα τα οποία γενικά αλλάζουν τα στατιστικά
- uint64 t \* loadRelation(char\*): Φόρτωση των σχέσεων στη μνήμη
- 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 \*): Υλοποίηση merge-join
- column\_data load\_column\_data(relation \*, int rel,int col): Φόρτωση εγγραφών μιας κολώνας απο την μνήμη σε ένα struct column\_data
- column\_data load\_from\_IR(relation \*,int ,int ,uint64 t ,uint64 t \*): Φόρτωση εγγραφών μιας κολώνας απο την ενδιάμεση δομή σε ένα struct column\_data
- Result \* scan(column\_data ,column\_data ,int \*): Σάρωση μιας κολώνας στην περίπτωση που έχουμε κατηγορήμα με μορφή Join αλλά στην ίδια κολώνα της ίδιας 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 ένα διαθέσιμο thread από 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 τελειώσουν την δουλειά τους όλα τα threads τα οποία έχουν πάρει δουλειές από την ουρά των jobs. Ουσιαστικά χρησιμοποιείται έτσι ώστε να εκτελείται πρώτα το ένα batch για να εκτυπωθούν τα αθροίσματα και να συνεχίσουμε με το άλλο 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 κάθε κολώνα κάθε σχέσης. Μικρότερη τιμή, μεγαλύτερη τιμή , πλήθος εγγραφών , διακριτές τιμές.

```
struct Predicates{
    int relation1;
    int colum1;
    int relation2;
    int colum2;
    int num;
```

```
char op;  
int prio;  
};
```

Πληροφορίες για κάθε κατηγορία. Δηλαδή ποιες σχέσεις συμμετέχουν ποιες κολώνες των σχέσεων αυτών συμμετέχουν, ποια είναι η πράξη που γίνεται ,ποιος είναι ο αριθμός σε περίπτωση που το κατηγορημα είναι φίλτρο και ποια είναι η προτεραιότητα του κατηγορήματος αυτού.

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

```
};
```

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

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

Μια εγγραφή που έχει ένα αναγνωριστικό id και μια τιμή

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

Αυτό το struct χρησιμοποιείται από την Sort μας για να κατηγοριοποιεί τις εγγραφές με βάση τα byte τους όπως μας ζητείται

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

Αυτο το struct το χρησιμοποιούμε 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 κατηγορημα

```
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 όπως τα ίδια τα threads τα id τους, με ακεραίους αριθμούς που μας πληροφορούν κάθε στιγμή για το ποσα threads δουλεύουν ,ποσα είναι ζωντανά, πόσα δουλειές έχουν τελειώσει ,ποσες απομένουν. Επίσης η ουρά με τις “δουλειές” και φυσικά κάποια mutexes και κάποιες 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 κάθε thread για να κάνει την δουλειά που του έχουμε αναθέσει

- **Χρόνοι εκτέλεσης και μνήμη που δεσμεύτηκε 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 εφόσον οι διαφορές με τα υπόλοιπα είναι πολύ μικρές , επιλέξαμε το threadpool που δημιουργούμε να αποτελείται από 4 threads.

Ακολουθεί πίνακας με χρόνους εκτέλεσης του προγράμματος πριν και μετά 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 σε μηχανή 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 έχει υλοποιηθεί κατάλληλο Makefile οπότε αρκεί εντολή : make

Στο Makefile μεταγλωττίζεται κάθε source αρχείο μόνο του έτσι ώστε να δημιουργηθούν τα κατάλληλα object files και τέλος δημιουργείται το executable αρχείο από όλα τα εκτελέσιμα.

Επίσης με `make clean` διαγράφονται τα `object` αρχεία και το `executable`  
Για την εκτέλεση του προγράμματος αρκεί η εντολή : `./main <file1> <file2>`  
,όπου

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