ΤΕΛΙΚΗ ΑΝΑΦΟΡΑ

PROJECT 2019-2020

ΜΕΛΗ:

1115201600183 ΘΕΟΔΩΡΟΣ ΤΣΟΛΑΚΟΣ

1115201300226 ΧΡΗΣΤΟΣ ΠΑΡΑΣΚΕΥΑΣ ΔΗΜΗΤΡΟΠΟΥΛΟΣ

1115201600021 ΧΡΥΣΟΥΛΑ ΒΙΡΒΙΔΑΚΗ

Χρήσιμο για την ανάλυση της εργασίας θα ήταν να τη χωρίσουμε σε τρία δομικά μέρη, 1) την είσοδο των δεδομένων από τα αρχεία και την δημιουργία των αρχικών δομών, 2) την εκτέλεση των επί μέρους queries. Το μέρος 3) αναφέρεται στο παράγοντα του χρόνου της υλοποίησης και στα στατιστικά που καταγράφηκαν κατά τη διάρκεια της εργασίας.

1) 1^{ο} ΜΕΡΟΣ: ΕΙΣΟΔΟΣ ΔΕΔΟΜΕΝΩΝ-ΔΗΜΙΟΥΡΓΙΑ ΔΟΜΩΝ

Command line flags:

Το πρόγραμμα δέχεται από τη γραμμή εντολών ένα πρώτο υποχρεωτικό όρισμα που είναι είτε –s είτε –m , όπου στη πρώτη περίπτωση επεξεργάζεται τα δεδομένα του small φακέλου, ενώ στη δεύτερη περίπτωση του medium φακέλου. Ακόμα έχει τη δυνατότητα να ορίσει από τη γραμμή εντολών πόσα threads θα αξιοποιούνται για την παραλληλοποίηση των queries (2^{ο} όρισμα), πόσα για την παραλληλοποίηση του sort ενός query (3^{ο} όρισμα), και ανά πόσα jobs θα χωρίζονται τα υπάρχοντα buckets στην διαδικασία της ταξινόμησης (4^{ο} όρισμα). Σημείωση: ο αριθμός των queryThreads και sortThreads πρέπει να είναι μεγαλύτερος από 0 και για τα δύο καθώς εάν τα πρώτα είναι 0 προφανώς δεν λύνεται κανένα query και αν τα δεύτερα είναι 0 δημιουργείται deadlock καθώς τα queryThreads περιμένουν να γίνει το sort από τα sortThreads για να συνεχίσουν.

Παράδειγμα εκτέλεσης: ./prog –m 3 8 4

Σε αυτή τη περίπτωση επεξεργαζόμαστε τα medium αρχεία ενώ έχουμε 3 queryThreads, 8 sortThreads και κάθε ταξινόμηση χωρίζεται σε 4 επιμέρους jobs που διαμοιράζονται στα sort threads.

Countrows:

Πρώτο βήμα είναι να διαβάσουμε το αρχείο init από το οποίο θα μπορέσουμε να δούμε πόσα είναι τα αρχεία εισόδου και πώς λέγονται. Η μεταβλητή που δείχνει τον συνολικό αριθμό αρχείων εισόδου είναι η num_of_files. Επόμενο βήμα είναι να μετρήσουμε πόσα

είναι τα συνολικά queries που έχουμε να εκτελέσουμε, μεταβλητή η οποία υπολογίζεται μέσω της συγκεκριμένης συνάρτησης και αποθηκεύεται στη μεταβλητή num_of_queries.

ReadInputFiles:

Στη συνέχεια διαβάζουμε μέσω αυτής της συνάρτησης ένα-ένα τα αρχεία εισόδου και αποθηκεύουμε κατά στήλες τα δεδομένα τους. Αφού διαβάσουμε όλα τα δεδομένα μίας στήλης ενός αρχείου, υπολογίζουμε και τα στατιστικά της. Το πλήθος των δεδομένων της στήλης Fa είναι φυσικά ο αριθμός των γραμμών του αρχείου, η μικρότερη τιμή της στήλης Ia υπολογίζεται κατά την ανάγνωση των δεδομένων, κρατώντας σε μία μεταβλητή min την έως τώρα μικρότερη τιμή που έχει διαβαστεί και με ανάλογο τρόπο υπολογίζεται και η μεγαλύτερη τιμή της στήλης Ua, με τη βοήθεια της μεταβλητής max. Για το πλήθος των μοναδικών τιμών της στήλης Da χρησιμοποιούμε έναν boolean πίνακα check ο οποίος έχει Ua – Ia +1 θέσεις. Αν αυτό το νούμερο ξεπερνάει όμως έναν μεγάλο πρώτο αριθμό N (εμείς ορίσαμε N = 6700417), τότε ο πίνακας αυτός θα έχει N θέσεις. Αρχικοποιούμε όλο τον πίνακα με false και στη συνέχεια διατρέχουμε ξανά την στήλη. Για κάθε τιμή x της στήλης κάνουμε true την θέση x – Ia, αν ο πίνακας έχει Ua – Ia +1 θέσεις, ή την θέση (x-Ia) % N, αν ο πίνακας έχει N θέσεις. Κάθε φορά που κάνουμε μία τιμή του πίνακα true ελέγχουμε εάν προηγουμένως ήταν false, κι αν ήταν αυξάνουμε έναν μετρητή. Η τιμή αυτού του μετρητή εν τέλει θα ισούται και με το πλήθος των μοναδικών τιμών της στήλης.

ΔΟΜΕΣ :

Για τις λειτουργίες που έχουν έως τώρα αναλυθεί χρειάστηκαν οι ακόλουθες δομές:

Initial_Table* Table

Η μεταβλητή Table είναι ένας πίνακας από δομές τύπου Initial_Table , με θέσεις τόσες όσες και ο αριθμός των αρχείων εισόδου με τις εγγραφές. Κάθε θέση του πίνακα έχει τον αριθμό των γραμμών του συγκεκριμένου αρχείου, τον αριθμό των στηλών του αρχείου και έναν δείκτη στον πίνακα που εν τέλει θα αποθηκευτούν τα δεδομένα του αρχείου αυτού.

uint64_t ***arrayname

Η μεταβλητή arrayname είναι ένας τρισδιάστατος πίνακας. Ουσιαστικά δηλαδή είναι ένας πίνακας από πίνακες με τα δεδομένα όλων των αρχείων. Ένας πίνακας με τόσους πίνακες, όσα είναι τα αρχεία εισόδου με τα δεδομένα, στην περίπτωση μας, δηλαδή, πίνακας με num_of_files πίνακες. Αν έχουμε, για παράδειγμα, τον arrayname[2], σημαίνει ότι είναι ένας δείκτης στον πίνακα με τα δεδομένα του τρίτου αρχείου εισόδου, του r2. Τώρα κάθε επί μέρους πίνακας αποθηκεύει τα δεδομένα ενός αρχείου κατά στήλες και όχι κατά γραμμές, διότι εμείς θα επεξεργαζόμαστε στη συνέχεια δεδομένα μίας στήλης, οπότε με αυτόν τον τρόπο αποθήκευσης θα γλυτώσουμε σημαντικό χρόνο προσπέλασης. Άρα για παράδειγμα ο arrayname[2][1][20] συμβολίζει την τιμή της $21^{ης}$ γραμμής και της $2^{ης}$ στήλης του $3^{ου}$ πίνακα δεδομένων δηλαδή του $3^{ου}$ αρχείου εισόδου.

Statistics* initStats[num_of_files]

Η μεταβλητή initStats είναι ένας δείκτης σε πίνακες από δομές τύπου Statistics. Συγκεκριμένα ένας δείκτης σε τόσους πίνακες, όσα είναι και τα αρχεία δεδομένων εισόδου (num_of_files). Κάθε ένας πίνακας από αυτούς έχει τόσες θέσεις όσες και οι στήλες του συγκεκριμένου αρχείου. Αν για παράδειγμα θέλουμε να ξέρουμε πόσες θέσεις έχει ο πίνακας για το αρχείο r1 που είναι το δεύτερο στη σειρά αρχείο εισόδου, αρκεί να δούμε πόσες στήλες έχει αυτό το αρχείο. Την πληροφορία αυτή μας την παρέχει, όπως είπαμε και παραπάνω η μεταβλητή Table[1].columns . Μία δομή τύπου Statistics περιέχει τα στατιστικά στοιχεία μίας στήλης ενός αρχείου. Δηλαδή, περιέχει το πλήθος των στοιχείων της στήλης, την ελάχιστη και την μέγιστη τιμή της στήλης, καθώς και το πλήθος των μοναδικών τιμών της στήλης. Τέλος, περιέχει και έναν boolean πίνακα check, ο οποίος όπως είδαμε χρησιμοποιείται για την εύρεση του πλήθους των μοναδικών τιμών της στήλης.

Οι πίνακες αυτοί δημιουργούνται και αρχικοποιούνται στο πρώτο μέρος της εργασίας και δεν μεταβάλλονται οι τιμές τους καθ' όλη τη διάρκεια εκτέλεσης του προγράμματος. Ουσιαστικά, λοιπόν , μέχρι αυτό το σημείο έχουμε αποθηκεύσει σε πίνακες όλα τα δεδομένα των αρχείων, μπορούμε να γνωρίζουμε άμεσα πόσες γραμμές και πόσες στήλες έχει ο κάθε πίνακας που αντιπροσωπεύει τα δεδομένα ενός αρχείου, ενώ έχουμε υπολογίσει και αποθηκεύσει για κάθε στήλη καθενός πίνακα τα στατιστικά της δεδομένα.

2) 2^{ο} ΜΕΡΟΣ: ΕΚΤΕΛΕΣΗ ΤΩΝ QUERIES

Ερχόμαστε τώρα στο δεύτερο και σημαντικότερο μέρος της εργασίας, την εκτέλεση των queries. Για την ευκολότερη κατανόηση μπορούμε να χωρίσουμε και το κομμάτι αυτό σε τρία επιμέρους μέρη, 2α) την αναδιάταξη των κατηγορημάτων με βάση τα στατιστικά στοιχεία τους, 2β) την εκτέλεση των κατηγορημάτων, 3β) και τον υπολογισμό των αθροισμάτων των προβολών.

2α) ΑΝΑΔΙΑΤΑΞΗ ΚΑΤΗΓΟΡΗΜΑΤΩΝ

readCommand:

Σε πρώτο στάδιο διαβάζουμε ένα query και αποθηκεύουμε τα κατηγορήματα από τα οποία αποτελείται σε μία λίστα, όπου κάθε κόμβος είναι ένα κατηγόρημα. Ακόμα αποθηκεύουμε σε μία ξεχωριστή λίστα όλες τις προβολές, για τις οποίες θα πρέπει στο τέλος να υπολογιστούν τα αθροίσματά τους.

commandListSort:

Στην συνάρτηση αυτή δίνεται η λίστα με τα κατηγορήματα ενός query και μεταφέρει στην αρχή της λίστας αυτής όλα τα κατηγορήματα που είναι φίλτρα, είτε της μορφής R.A = k, R.A > k, R.A < k είτε της μορφής R.A = R.B (που αλλιώς μπορούν να ονομαστούν self-join). Στο τέλος της λίστας αυτής θα έχουν απομείνει μόνο τα joins.

newSort:

Στην συνάρτηση αυτή δίνεται η λίστα και πάλι των κατηγορημάτων, στην οποία πλέον είναι τα φίλτρα στην αρχή της και τα join στο τέλος της. Για τα φίλτρα αυτά , ενημερώνονται κατάλληλα τα στατιστικά των στηλών που συμμετείχαν στο φίλτρο , καθώς και των υπόλοιπων στηλών των πινάκων αυτών. Μετά από την αλλαγή των στατιστικών γίνεται έλεγχος του πλήθους των τιμών των στηλών του πίνακα που συμμετείχαν στα φίλτρα, και αν η τιμή αυτή είναι ίση με 0 σταματάει η διαδικασία και δεν οδηγούμαστε ποτέ στο επόμενο βήμα της εκτέλεσης των κατηγορημάτων. Αυτό συμβαίνει, διότι για να βγει το στατιστικό ίσο με 0 σημαίνει πως η λύση του query αυτού είναι NULL, οπότε μπορούμε να εξοικονομήσουμε πολύτιμο χρόνο. Με βάση αυτά τα νέα στατιστικά ακολουθεί η διαδικασία του Job Enumeration.

evaluate:

Η συνάρτηση αυτή είναι που υλοποιεί την λειτουργία του Job Enumeration. Εάν υπάρχει μόνο ένα join στη λίστα των κατηγορημάτων, τότε δεν υπάρχει κάτι να κάνει. Εάν είναι δύο τα joins τότε πάει και ελέγχει με βάση τα στατιστικά που μας έχουν δοθεί και από την εκφώνηση ποιό από τα δύο join θα έχει λιγότερο κόστος, δηλαδή λιγότερο πλήθος αποτελεσμάτων. Αυτό με το μικρότερο κόστος θα είναι και το πρώτο join που θα εκτελεστεί. Αν τέλος, είναι πάνω από δύο τα join τότε, σε πρώτο επίπεδο ψάχνουμε να βρούμε ποιά είναι η καλύτερη δυάδα από joins, ποιά έχει το συνολικό μικρότερο κόστος. Αν κατά τη διάρκεια αυτών των ελέγχων βρεθεί κάποιο join με κόστος ίσο με 0, δηλαδή πλήθος αποτελεσμάτων ίσο με 0, τότε όπως και προηγουμένως σταματάει η διαδικασία και εκτυπώνεται απευθείας NULL, χωρίς να γίνει η εκτέλεση των κατηγορημάτων. Αφού βρούμε τη βέλτιστη δυάδα την τοποθετούμε ακριβώς μετά από τα φίλτρα στη λίστα με τα κατηγορήματα. Στη συνέχεια έχοντας πλέον ως δεδομένα τα δύο πρώτα join βλέπουμε ποιό από τα εναπομείναντα έχει το μικρότερο κόστος και το τοποθετούμε στη λίστα μετά από τα προηγούμενα δύο. Έτσι η λίστα με τα κατηγορήματα καταλήγει να είναι στην ιδανικότερη σειρά για την γρήγορη εκπόνηση του query.

ΔΟΜΕΣ:

Για τις λειτουργίες που αναλύθηκαν στο κομμάτι αυτό χρειάστηκαν οι ακόλουθες δομές:

COMMANDLIST *commandListHead

Η δομή αυτή αποθηκεύει όλες τις απαραίτητες πληροφορίες που χρειαζόμαστε να γνωρίζουμε για το query. Συγκεκριμένα αποθηκεύει την λίστα με τα κατηγορήματα, την λίστα με τις προβολές, τα αθροίσματα των οποίων θα πρέπει να εκτυπωθούν, μία λίστα από τα ονόματα των πινάκων που συμμετέχουν στο συγκεκριμένο query, το πλήθος αυτών των πινάκων, το πλήθος των προβολών και τον αύξοντα αριθμό του query, ο οποίος χρησιμεύει για να γνωρίζουμε με ποιά σειρά πρέπει να εκτυπωθούν τα αποτελέσματα όταν ολοκληρωθεί το batch.

PROJECTION *projectionHead

Η δομή αυτή είναι εκείνη που περιέχει τη λίστα των projections ενός query. Κάθε κόμβος, δηλαδή, αποτελείται από τον πίνακα και την στήλη του ,από την οποία πρέπει να υπολογιστεί το άθροισμα.

Arrays* arraylist

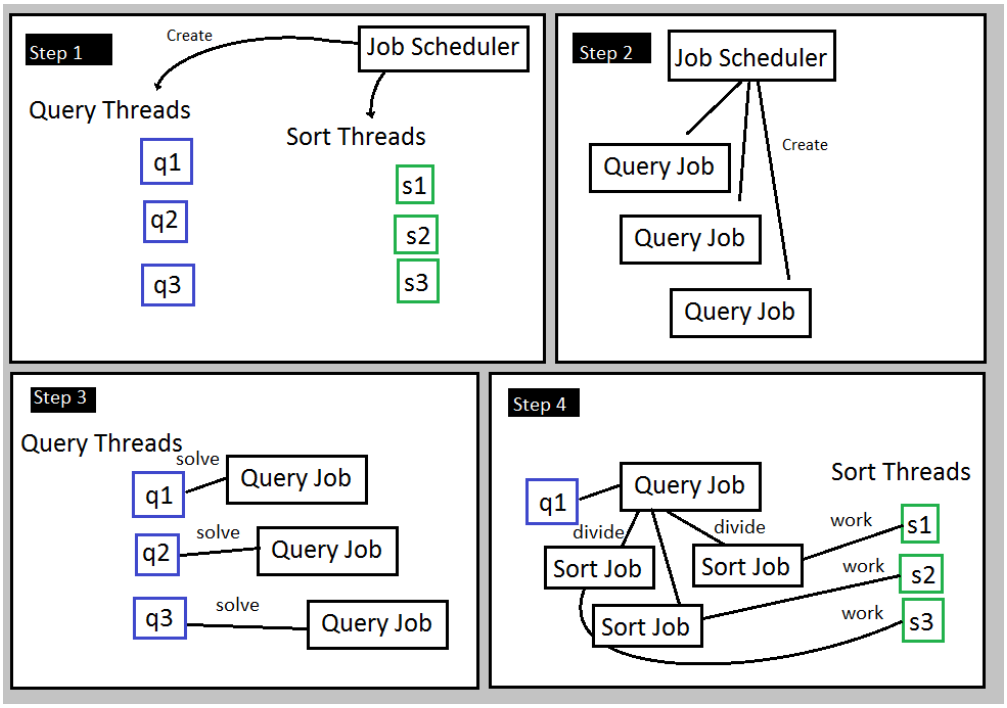
Η δομή αυτή είναι εκείνη που αποθηκεύει τη λίστα των πινάκων που συμμετέχουν στο συγκεκριμένο query. Μέσα σε αυτή τη λίστα τα ονόματα των πινάκων είναι τα πραγματικά ονόματα που έχουν και τα αρχεία που αντιπροσωπεύουν. Δηλαδή, εάν στο πρώτο μέρος του query υπάρχουν οι πίνακες 5 0 3, τότε στη λίστα θα αποθηκευτούν αυτοί οι αριθμοί αυτούσιοι, και όχι η μετέπειτα ονομασία που θα έχουν στο δεύτερο και τρίτο μέρος του query (0, 1, 2). Αυτό χρειάζεται γιατί κατά τη διάρκεια εκτέλεσης του query θα χρειαστούμε πληροφορίες που αφορούν τους πίνακες αυτούς και που έχουν αποθηκευτεί στη δομή Table, που όπως είπαμε προηγουμένως έχει αποθηκευμένους εξ αρχής όλους τους πίνακες.

NODE *nodeHead

Η δομή αυτή περιέχει τη λίστα με τα κατηγορήματα του query. Κάθε κόμβος της αποτελείται από το νούμερο του πρώτου πίνακα και της στήλης αυτού που συμμετέχει στο συγκεκριμένο κατηγόρημα καθώς και το όνομα του δεύτερου πίνακα και της στήλης αυτού. Στην λίστα αυτή όταν λέμε όνομα του πίνακα εννοούμε σύμφωνα με την αρίθμηση που έχει στο συγκεκριμένο query και όχι το αρχικό του όνομα. Ακόμα, στη δομή αποθηκεύουμε μία μεταβλητή type, η οποία μας βοηθάει να διαχωρίζουμε τα φίλτρα μεταξύ τους. Όταν η τιμή της μεταβλητής είναι ίση με 0 έχουμε R.A = k, όταν είναι ίση με 1 έχουμε R.A > k και όταν είναι ίση με 2 έχουμε R.A < k. Όταν το κατηγόρημα δεν είναι φίλτρο πάλι η μεταβλητή έχει τιμή 0. Στα κατηγορήματα που είναι φίλτρα και συνεπώς δεν υπάρχει δεύτερος πίνακας, η τιμή του δεύτερου πίνακα γίνεται -1 , ενώ της δεύτερης στήλης γίνεται όσο και η τιμή του k.

2β) ΕΚΤΕΛΕΣΗ ΚΑΤΗΓΟΡΗΜΑΤΩΝ

Όλες οι παραπάνω λειτουργίες πραγματοποιούνται από τον Job Scheduler. Τώρα ήρθε η ώρα να μιλήσουμε για νήματα. Ο Job Scheduler δημιουργεί έναν αριθμό από queryThreads και έναν αριθμό από sortThreads τα οποία μετά τη δημιουργία τους περιμένουν σε ένα σηματοφόρο (ξεχωριστό για κάθε τύπο νήματος) μέχρι να προστεθεί κάποιο job που να τα αφορά. Έπειτα από τη δημιουργία της λίστας που αναπαριστά ένα query με την συνάρτηση readCommand() ο Job Scheduler προσθέτει στην λίστα των query jobs το παραπάνω query ώστε να επιλυθεί. Έπειτα ξυπνάει ένα από τα queryThreads το οποίο αναλαμβάνει την υλοποίηση. Έτσι, συνεχίζει ο Job Scheduler με κάθε query περιμένοντας μετά το τέλος ενός batch τα νήματα να επιλύσουν τα queries του batch και να τυπώσουν τα αποτελέσματα. Οι λύσεις των queries προστίθενται σε μία priority queue με key τον αριθμό του query ώστε να τυπώνονται στη σειρά. Για επιπλέον βοήθεια παραθέτουμε ένα σχήμα που θα διευκολύνει την κατανόηση.



FindSolution

Η συνάρτηση αυτή αποτελεί την βασική λειτουργία ενός queryThread και ουσιαστικά είναι η επίλυση ενός δοσμένου query. Έχουμε έως τώρα μιλήσει για την αναδιάταξη των κατηγορημάτων, και ερχόμαστε στο σημείο να αναλύσουμε τον τρόπο με τον οποίο εκτελούνται ένα-ένα τα κατηγορήματα και τον τρόπο με τον οποίο αποθηκεύονται τα ενδιάμεσα αποτελέσματα σε δομές.

ΔΟΜΕΣ:

Πρωτίστως όμως πρέπει να εξηγήσουμε τις δομές που θα χρησιμοποιήσουμε για να γίνει μετά κατανοητή η ανάλυση των επί μέρους συναρτήσεων.

Current_Table* Cur_Table

Ο δείκτης αυτός δείχνει σε έναν πίνακα από δομές τύπου Current_Table, ο οποίος έχει τόσες θέσεις όσοι οι πίνακες που συμμετέχουν στο συγκεκριμένο query. Κάθε θέση του Cur_Table μας δίνει πληροφορίες για τα δεδομένα του πίνακα τα οποία εξακολουθούν να βρίσκονται στο ενδιάμεσο αποτέλεσμα, μετά δηλαδή από την εκτέλεση κάποιου μέρους των κατηγορημάτων. Πιο αναλυτικά, μας ενημερώνει για το πόσες από τις γραμμές του αρχικού πίνακα έχουν απομείνει στο ενδιάμεσο αποτέλεσμα, για το ως προς ποιά στήλη είναι ταξινομημένος ο πίνακας, (αν ο πίνακας βρίσκεται στο ενδιάμεσο αποτέλεσμα αλλά δεν είναι ταξινομημένος ως προς κάποια στήλη η τιμή του πεδίου αυτού είναι -1, ενώ αν δεν βρίσκεται καθόλου στο ενδιάμεσο αποτέλεσμα η τιμή του πεδίου είναι -2), και τέλος έχει έναν δείκτη προς μία λίστα filtered, η οποία χρησιμοποιείται για τα φίλτρα και τα self-join και που θα αναλυθεί καλύτερα στη συνέχεια.

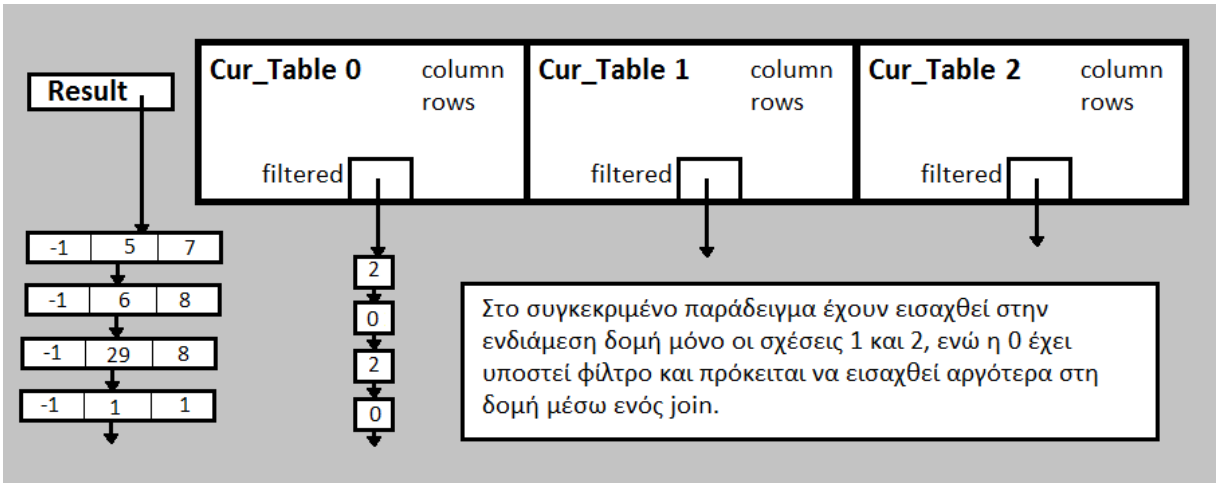
RowIds* Result

Ο δείκτης αυτός δείχνει στη λίστα η οποία κρατάει όλα τα ενδιάμεσα αποτελέσματα των row ids που έχουν ως τώρα βρεθεί από την εκτέλεση των κατηγορημάτων. Πιο συγκεκριμένα αποτελείται από tuples, όπου το κάθε tuple είναι ένας πίνακας που αποθηκεύει ζευγάρια από row ids και έχει τόσες θέσεις όσοι είναι και οι πίνακες που συμμετέχουν στο query. Αυτή η ενδιάμεση δομή ενημερώνεται μόλις ξεκινήσουν να εκτελούνται τα joins. Όταν γίνει το πρώτο δημιουργούνται τα πρώτα tuples, στα οποία αποθηκεύονται τα ζευγάρια των γραμμών που έκαναν join σε αυτούς τους δύο πίνακες που συμμετείχαν, ενώ σε όλες τις υπόλοιπες θέσεις των πινάκων βάζουμε την τιμή -1, και περιμένουμε να εκτελεστεί κάποιο επόμενο κατηγόρημα στο οποίο θα συμμετέχουν και αυτοί.

FilRowIds* filtered

Ο δείκτης αυτός δείχνει σε μία λίστα από row ids τα οποία αφορούν αποκλειστικά έναν από τους πίνακες που συμμετέχουν στο query. Αν ένας πίνακας δηλαδή συμμετέχει σε κατηγόρημα τύπου φίλτρο ή, και self-join τότε τα αποτελέσματα αυτών των πράξεων θα αποθηκευτούν σε μία ενδιάμεση δομή τέτοιου τύπου, δείκτης στην οποία όπως είδαμε περιέχεται στον Cur_Table του κάθε πίνακα. Αν ένας πίνακας δε συμμετέχει καθόλου σε τέτοιου είδους πράξεις τότε ο δείκτης αυτός θα παραμείνει NULL .

Για την καλύτερη κατανόηση των δομών αυτών αλλά και των συναρτήσεων που θα αναλυθούν αμέσως μετά παραθέτουμε το επόμενο σχήμα:



Σημαντικότερες Συναρτήσεις:

Filter:

Όταν εντοπίσουμε ένα κατηγόρημα το οποίο είναι τύπου φίλτρο καλούμε αυτή τη συνάρτηση. Μέσω της συνάρτησης CreateNewIndex, που θα εξηγήσουμε αμέσως μετά, μπορούμε να γνωρίζουμε πόσες γραμμές από τον πίνακα υπάρχουν τη δεδομένη στιγμή που εκτελούμε το φίλτρο. Δηλαδή, μαθαίνουμε εάν έχει συμμετέχει στο παρελθόν ο συγκεκριμένος πίνακας σε άλλο κατηγόρημα , και συνεπώς έχει αλλάξει ο αριθμός των γραμμών, ή αν είναι ο αρχικός αριθμός των γραμμών του αρχείου. Ταυτόχρονα μέσω της εσωτερικής αυτής συνάρτησης ο πίνακας που συμμετέχει στο φίλτρο ταξινομείται ως προς την στήλη που γίνεται το φίλτρο, με αποτέλεσμα να είμαστε έτοιμοι στη συνέχεια να επιλέξουμε τις γραμμές αυτές που ικανοποιούν το κατηγόρημα. Πιο συγκεκριμένα μέσα στην Filter ανάλογα με το type του φίλτρου διακρίνονται οι εξής περιπτώσεις: 1) αν είναι ίσο με 2 τότε έχουμε φίλτρο τύπου ‘’μικρότερο’’ όπου κρατάμε όλες τις γραμμές του πίνακα από την αρχή του μέχρι να συναντήσουμε γραμμή της οποίας η τιμή είναι ίση με το key του φίλτρου, 2) αν είναι ίσο με 1 τότε έχουμε φίλτρο τύπου ‘’μεγαλύτερο’’ όπου κρατάμε από την πρώτη γραμμή που έχει τιμή μεγαλύτερη από το key του φίλτρου μέχρι και την τελευταία γραμμή του πίνακα, 3) αν είναι ίσο με 0 τότε έχουμε φίλτρο τύπου ‘’ίσο’’ όπου κρατάμε αποκλειστικά τις γραμμές που έχουν τιμή ίση με το key του φίλτρου. Οι γραμμές αυτές που επιλέχθηκαν θα εισέλθουν σε μία λίστα η οποία θα κρατηθεί στην κατάλληλη δομή Cur_Table που σχετίζεται με τον πίνακα που συμμετείχε στο φίλτρο, και συγκεκριμένα στο δείκτη filtered αυτής της δομής. Ακόμα θα ενημερωθεί ο αριθμός των γραμμών της δομής για αυτόν τον πίνακα και στο πεδίο της στήλης θα μπει ο αριθμός της στήλης που έγινε το φίλτρο καθώς πλέον ο πίνακας αυτός είναι ταξινομημένος ως προς αυτή τη στήλη. Αν ο αριθμός των γραμμών της δομής γίνει ίσος με μηδέν, σημαίνει ότι καμία γραμμή δεν ικανοποιεί το κατηγόρημα και συνεπώς επιστρέφουμε -1 για να

ειδοποιήσουμε να σταματήσει η εκτέλεση του κατηγορήματος και για να εκτυπωθεί απευθείας NULL .

CreateNewIndex

Η συνάρτηση αυτή καλείται σε διάφορα σημεία στο κώδικα και είναι ιδιαίτερα χρήσιμη. Η λειτουργία της είναι να παίρνει έναν πίνακα και συγκεκριμένα μία στήλη αυτού, να ταξινομεί τον πίνακα ως προς τη στήλη αυτή και να επιστρέφει τον αριθμό των γραμμών που έχει τη δεδομένη στιγμή ο πίνακας. Για να το κάνει αυτό ελέγχει εάν ο πίνακας έχει τοποθετεί σε κάποια ενδιάμεση δομή, δηλαδή εάν έχει συμμετέχει σε κάποιο προηγούμενο κατηγόρημα. Συνεπώς ελέγχει το δείκτη filtered της αντίστοιχης δομής. Αν δεν είναι NULL σημαίνει πως στον πίνακα έχει εκτελεστεί κάποιου είδους φίλτρο. Παίρνουμε, λοιπόν, από τη λίστα αυτή τις εναπομείναντες γραμμές, τον αριθμό των οποίων όπως προείπαμε έχουμε αποθηκεύσει στη δομή Cur_Table, και δημιουργούμε με αυτές έναν Index πίνακα, τον οποίο στη συνέχεια ταξινομούμε με τη συνάρτηση Sort, που θα εξηγήσουμε αμέσως μετά. Αντίθετα αν ο πίνακας δεν έχει συμμετάσχει σε προηγούμενο κατηγόρημα τότε δημιουργούμε τον Index πίνακα για την στήλη που θέλουμε στο συγκεκριμένο κατηγόρημα και τον ταξινομούμε και πάλι ως προς αυτή τη στήλη. Και στις δύο περιπτώσεις επιστρέφουμε τον αντίστοιχο αριθμό γραμμών του πίνακα.

Sort

Η λειτουργία της ταξινόμησης υλοποιείται σύμφωνα με τις οδηγίες και τις απαιτήσεις του πρώτου μέρους της εργασίας, με κάποιες βελτιστοποιήσεις. Δέχεται ως όρισμα έναν πίνακα και πρώτο βήμα είναι να δημιουργήσει έναν πίνακα hist, ο οποίος ουσιαστικά έχει 256 θέσεις, μία για κάθε πιθανή τιμή του πρώτου byte των τιμών του πίνακα. Συνεπώς διατρέχουμε τον πίνακα και ανάλογα με την τιμή του πρώτου byte των αριθμών αυξάνουμε την κατάλληλη τιμή του πίνακα hist.

Έχοντας, λοιπόν, έτοιμο τον hist, δεύτερο βήμα είναι η δημιουργία του πίνακα psum, ο οποίος έχει επίσης 256 θέσεις. Ο πίνακας αυτός έχει τα offsets από τα οποία ξεκινάει κάθε μία από τις διαφορετικές τιμές του πρώτου byte των αριθμών. Δηλαδή αν υπάρχουν 3 αριθμοί των οποίων το πρώτο byte έχει τιμή 0, τότε στον psum θα ισχύει: psum[0] = 0, psum[1] = 3 .

Τρίτο βήμα είναι η αναδιάταξη του αρχικού πίνακα σύμφωνα με τις έως τώρα πληροφορίες, δηλαδή σύμφωνα με τις τιμές που έχει το πρώτο byte των αριθμών του. Με την συνάρτηση Reorder δημιουργείται ένας δεύτερος πίνακας ο οποίος έχει όσες ακριβώς θέσεις είχε ο αρχικός μόνο που τα στοιχεία μεταφέρονται σε αυτόν ταξινομημένα ως προς το πρώτο byte. Αυτός ο πίνακας, που μπορούμε να τον αποκαλούμε και αντίγραφο, θα βοηθήσει σε όλη τη διαδικασία της ταξινόμησης, καθώς τα δεδομένα θα μεταφέρονται διαρκώς από αυτόν προς τον αρχικό και αντίστροφα, μέχρι να έχουμε το τελικό αποτέλεσμα. Όλες οι λειτουργίες που περιγράφουμε τώρα θα γίνουν επαναληπτικά

καλώντας τις ίδιες συναρτήσεις και για τα επόμενα bytes των αριθμών μέχρι να ταξινομηθεί πλήρως ο αρχικός πίνακας.

Στη συνέχεια, τέταρτο βήμα είναι να καλέσουμε τη συνάρτηση CreateBucketList, η οποία σύμφωνα με τις τιμές των πινάκων hist και psum χωρίζει τον αρχικό πίνακα σε τόσα buckets όσες είναι οι διαφορετικές τιμές του πρώτου byte των αριθμών του πίνακα, με μέγιστο φυσικά αριθμό το 256. Τα buckets αυτά τα τοποθετεί σε μία λίστα που είναι τύπου bucket. Στο κάθε κόμβο αυτής της δομής αποθηκεύεται η πρώτη καθώς και η τελευταία θέση του αρχικού πίνακα τις οποίες καταλαμβάνει το συγκεκριμένο bucket, το νούμερο του byte του αριθμού στο οποίο βρίσκεται αυτή τη στιγμή το στάδιο της ταξινόμησης και τέλος ένα flag το οποίο μας βοηθάει να γνωρίζουμε εάν στο στάδιο της ταξινόμησης στο οποίο βρισκόμαστε δουλεύουμε πάνω στον αρχικό πίνακα ή στο αντίγραφό του.

Έπειτα, πέμπτο βήμα είναι η κλήση της συνάρτησης FinalizeTables, η οποία παίρνει ορίσματα τον αρχικό πίνακα, το αντίγραφό του και την λίστα με τα buckets και αναλαμβάνει να ταξινομήσει ένα-ένα αυτά τα buckets. Ωστόσο, εάν συναντήσει κάποιο bucket το οποίο είναι πάνω από 64KB σε μνήμη τότε επαναλαμβάνεται η προηγούμενη διαδικασία, φτιάχνονται οι πίνακες hist, psum και με τη βοήθεια της DivideBucket χωρίζεται το προηγούμενο bucket σε επί μέρους με βάση αυτή τη φορά το επόμενο byte των αριθμών. Αντίθετα αν το bucket που συναντήσουμε είναι μικρότερο από 64KB σε μέγεθος, τότε είμαστε πια σε θέση να ταξινομήσουμε τους αριθμούς του με βάση τον αλγόριθμο quick sort. Εάν σε κάποια φάση της διαδικασίας τα buckets που έχουν δημιουργηθεί από την DivideBucket είναι μεγαλύτερα από τον αριθμό BUCKET_LIMIT ορισμένο είτε default σε 3 είτε από το command line, το querThread χωρίζει τα buckets σε BUCKET_LIMIT jobs τα οποία βάζει στην ουρά προς υλοποίηση για τα sortThreads. Έπειτα περιμένει για την ταξινόμηση του πίνακα και όταν αυτή πραγματοποιηθεί συνεχίζει την ολοκλήρωση του Query.

SelfJoin

Όταν εντοπίσουμε ένα κατηγόρημα που ανήκει στην κατηγορία του self-join, που δηλαδή ο πρώτος πίνακας του κατηγορήματος είναι ο ίδιος με τον δεύτερο (R.A = R.B), τότε καλούμε την συνάρτηση αυτή. Αρχικά ελέγχουμε αν αυτός ο πίνακας έχει συμμετάσχει προηγουμένως σε άλλο κατηγόρημα, και δηλαδή αν έχει αποθηκευτεί κάποια σε ενδιάμεση δομή. Ο έλεγχος αυτός γίνεται μέσω της τιμής της στήλης στην δομή Cur_Table για τον πίνακα αυτόν. Όπως έχουμε πει το πεδίο αυτό έχει τιμή ίση με την στήλη ως προς την οποία είναι ταξινομημένος ο πίνακας, -1 αν βρίσκεται στην ενδιάμεση δομή αλλά δεν είναι ταξινομημένος, και -2 αν δεν έχει χρησιμοποιηθεί ξανά ο πίνακας αυτός. Αν λοιπόν η τιμή αυτή είναι -2 τότε καλούμε και πάλι την CreateNewIndex, ώστε να ταξινομηθεί ο πίνακας ως προς την μία από τις στήλες που συμμετέχουν στο self-join και στη συνέχεια εισάγονται σε μία λίστα μόνο οι γραμμές όπου οι δύο αυτές στήλες έχουν την ίδια τιμή. Η

λίστα η οποία έχει στο τέλος όλες τις γραμμές που ικανοποιούν το κατηγόρημα, θα αποθηκευτεί στην αντίστοιχη δομή για τον πίνακα Cur_Table[R]->filtered. Παράλληλα, θα ενημερωθεί η τιμή στο πεδίο των γραμμών και θα γίνει ίση με αυτές που παρέμειναν στον πίνακα μετά το κατηγόρημα και η τιμή της στήλης θα γίνει ίση είτε με τη μία είτε με την άλλη στήλη που συμμετείχαν στο self-join.

Στην περίπτωση που η τιμή Cur_Table[R]->column > -2 , συνειδητοποιούμε ότι ο πίνακας έχει συμμετάσχει σε προηγούμενο κατηγόρημα. Όμως, θυμίζουμε ότι έχουμε ήδη φροντίσει πριν από ένα self-join να μπορεί να βρίσκεται στη λίστα των κατηγορημάτων ενός query μόνο φίλτρο. Άρα ο πίνακας θα έχει συμμετάσχει σε φίλτρο. Αυτό σημαίνει πως το αποτέλεσμα του φίλτρου θα το βρούμε από το δείκτη filtered και τη λίστα στην οποία θα δείχνει. Το μόνο που μένει λοιπόν να κάνουμε είναι για τις γραμμές που βρίσκονται σε αυτή τη λίστα να ελέγξουμε ποιές δεν ικανοποιούν το κατηγόρημα αυτό. Αυτές δηλαδή τις γραμμές για τις οποίες δεν ισχύει ότι R[A] = R[B] , θα τις διαγράψουμε από την λίστα αυτή. Συνεπώς θα πρέπει να μειώσουμε και τον αριθμό των γραμμών που υπάρχουν στο ενδιάμεσο αποτέλεσμα, εφόσον φυσικά υπήρχαν γραμμές που χρειάστηκε να διαγράψουμε. Την τιμή της στήλης στην δομή δε θα την αλλάξουμε αφού δεν κάναμε κάποια νέα ταξινόμηση της λίστας.

Αν ο αριθμός των γραμμών της δομής γίνει ίση με μηδέν, σημαίνει ότι καμία γραμμή δεν ικανοποιεί το κατηγόρημα και συνεπώς επιστρέφουμε -1 για να ειδοποιήσουμε να σταματήσει η εκτέλεση του query και για να εκτυπωθεί απευθείας NULL .

Join

Όταν εντοπίσουμε ένα κατηγόρημα που ανήκει στην κατηγορία των join θα κληθεί αυτή η συνάρτηση. Διακρίνουμε σε αυτό το σημείο τρεις διαφορετικές περιπτώσεις που μπορεί να συναντήσουμε κατά την εκτέλεση ενός join: 1) να βρίσκονται ήδη και οι δύο πίνακες στην ενδιάμεση δομή Result, δηλαδή να έχουν συμμετάσχει σε κάποιο κατηγόρημα (τύπου join συγκεκριμένα) προηγουμένως, 2) να βρίσκεται ο ένας από τους δύο πίνακες μόνο στην ενδιάμεση δομή Result, 3) να μην βρίσκεται κανείς από τους δύο πίνακες στην ενδιάμεση δομή Result (αυτό δε σημαίνει απαραίτητα όπως είπαμε ότι οι πίνακες δεν έχουν συμμετάσχει σε κάποιο κατηγόρημα τύπου φίλτρο ή, και self-join).

$1^{η}$ περίπτωση (και οι δύο πίνακες στη δομή Result):

Στην περίπτωση αυτή καλούμε την συνάρτηση EasyJoin, η οποία αυτό που κάνει είναι να ελέγχει για τα ζευγάρια των γραμμών της δομής Result που ανήκουν στους δύο πίνακες που συμμετέχουν στο join αν έχουν την ίδια τιμή στις κατάλληλες στήλες. Όποιες εγγραφές υπήρχαν στην ενδιάμεση δομή αλλά δεν ικανοποιούν το κατηγόρημα που εκτελούμε τη δεδομένη στιγμή θα διαγράφονται από την ενδιάμεση δομή. Στην πραγματικότητα δηλαδή όταν και οι δύο πίνακες είναι στην ενδιάμεση δομή το join λειτουργεί περισσότερο σα

φίλτρο αφού ή διατηρεί ίδιο το πλήθος των αποτελεσμάτων ή το μειώνει. Η μεταβλητή με τις γραμμές του κάθε πίνακα που υπάρχουν στο ενδιάμεσο αποτέλεσμα θα ενημερωθούν κατάλληλα αν χρειάζεται. Αν ο αριθμός των γραμμών γίνει μηδέν, τότε σημαίνει πως δεν έμεινε καμία εγγραφή στο ενδιάμεσο αποτέλεσμα και συνεπώς πρέπει απευθείας να εκτυπωθεί NULL ως λύση του query.

2η περίπτωση (ο ένας από τους δύο πίνακες στη δομή Result):

Έστω ότι ο πρώτος από τους δύο πίνακες που συμμετέχουν στο κατηγόρημα βρίσκεται στη δομή Result. Τότε αρχικά καλούμε την συνάρτηση CreateNewIndex για τον δεύτερο πίνακα και την στήλη που συμμετέχει στο join. Έτσι έχουμε έναν πίνακα Index ο οποίος περιέχει την στήλη του δεύτερου πίνακα που χρειαζόμαστε ταξινομημένο ως προς αυτή. Έπειτα, πρέπει να ελέγξουμε αν οι γραμμές του πρώτου πίνακα που βρίσκονται στη δομή Result είναι ταξινομημένες ως προς τη στήλη που συμμετέχει στο τωρινό join. Αν είναι είμαστε έτοιμοι να κάνουμε την συγχώνευση και να προσθέσουμε και τον δεύτερο πίνακα στην δομή Result (Synchronize). Αν δεν είναι θα πρέπει πρώτα να αναδιατάξουμε την σειρά όλων των tuples της δομής Result σύμφωνα με την σωστή στήλη του πρώτου πίνακα. Αυτή η διαδικασία χωρίζεται σε τρία βήματα. Αρχικά μετατρέπουμε τα tuples σε πίνακες index. Στη συνέχεια ταξινομούμε τους πίνακες αυτούς με τη βοήθεια των sortThreads, και τέλος ξαναμετατρέπουμε τους index πίνακες σε tuples. Για την υλοποίηση αυτών βοήθησαν οι εξής συναρτήσεις: IndexFromTuples, SortTriple, CreateHistTriple, ReorderTriple, PartitionTriple, MyqsortTriple, CopyBucketTriple, FinalizeTablesTriple, TuplesFromIndex. Όλες οι συναρτήσεις με κατάληξη Triple επιτελούν με μικρές αποκλίσεις τις λειτουργίες των αντίστοιχων συναρτήσεων που εξηγήσαμε κατά την ανάλυση της ταξινόμησης, για αυτό και δε χρειάζονται περεταίρω ανάλυση. Καταλήγουμε, λοιπόν και πάλι στο σημείο που έχουμε έναν Index πίνακα με τις γραμμές του δεύτερου πίνακα ταξινομημένο ως προς τη στήλη που χρειαζόμαστε και μία δομή Result της οποίας τα tuples είναι ταξινομημένα ως προς τη στήλη του πρώτου πίνακα που θέλουμε. Στο σημείο αυτό καλούμε τη συνάρτηση Synchronize.

Synchronize:

Η συνάρτηση αυτή είναι κάπως ιδιαίτερη για αυτό να αναλυθεί λεπτομερώς. Έχουμε πάρει από τα ορίσματα τον πίνακα Index που δημιουργήσαμε προηγουμένως και τη δομή Result, την οποία την αναδιατάξαμε στην δεύτερη περίπτωση προκειμένου να είναι ταξινομημένη όπως τη θέλουμε. Έχουμε, λοιπόν, από έναν δείκτη στις δύο αυτές δομές και αρχίζουμε να τις διατρέχουμε ώστε να βρούμε τα αποτελέσματα του join. Θυμίζουμε ότι στην δομή Result στην θέση του δεύτερου πίνακα τα tuples έχουν τιμή -1, διότι ο πίνακας τώρα μπαίνει πρώτη φορά στην ενδιάμεση αυτή δομή.

Άρα ξεκινάμε και εξετάζουμε τις τιμές στις οποίες δείχνουν οι δείκτες. Αν το row id του δείκτη που δείχνει στη δομή Result έχει μικρότερη τιμή από την τιμή του row id του δείκτη που δείχνει στον Index, τότε συνειδητοποιούμε ότι αφού ο πίνακας είναι ταξινομημένος

αποκλείεται σε κάποια επόμενη θέση του πίνακα να υπάρχει κάποιο row id με τέτοια τιμή. Συνεπώς θα πρέπει να διαγραφεί από τη δομή Result το tuple αυτό που περιέχει το row id του πρώτου πίνακα με την τιμή αυτή. Το διαγράφουμε λοιπόν με τη συνάρτηση DeleteId και μειώνουμε και τον αριθμό των tuples που έχει η δομή Result. Βάζουμε τον δείκτη της δομής Result να δείχνει στον επόμενο κόμβο από αυτόν που μόλις διαγράψαμε και συνεχίζουμε. Όσο βρίσκουμε μικρότερες τιμές από αυτήν στην οποία δείχνει ο άλλος δείκτης διαγράφουμε tuples.

Έστω τώρα ότι βρεθούν δύο row ids των οποίων οι τιμές είναι ίδιες. Τότε θα πάμε να βάλουμε μέσα στο tuple που δείχνει ο δείκτης του Result, στη θέση του δεύτερου πίνακα του join, το row id που δείχνει ο δείκτης του πίνακα Index, αφού πριν στη θέση αυτή υπήρχε -1. Μετακινούμε τον δείκτη του πίνακα στην επόμενη θέση. Υπάρχει πιθανότητα και το επόμενο row id να έχει την ίδια τιμή με το row id του πρώτου πίνακα που δείχνει τώρα ο δείκτης της δομής. Τώρα όμως δε μπορούμε απλά να πάμε να βάλουμε το row id στη κατάλληλη θέση του tuple, γιατί δεν είναι πια -1. Οπότε δημιουργούμε ένα αντίγραφο του tuple αυτού, το τοποθετούμε ακριβώς από πάνω του στη δομή Result και βάζουμε τώρα στη θέση του δεύτερου πίνακα το νέο row id με την τιμή αυτή. Τη διαδικασία εισαγωγής νέου tuple στη δομή Result την υλοποιεί η συνάρτηση InsertId2. Όσο προχωράμε τον δείκτη του πίνακα προς τα κάτω και βρίσκουμε row id με την ίδια τιμή με αυτή του row id της δομής δημιουργούμε νέα tuples. Ωστόσο έχουμε συγκρατήσει σε έναν δείκτη temp την θέση στον πίνακα όπου συναντήσαμε για πρώτη φορά row id με τιμή που ισούται με κάποιου row id της δομής Result. Αυτό είναι απαραίτητο, διότι όταν κάποια στιγμή ο δείκτης του πίνακα βρει τιμή μεγαλύτερη από τις προηγούμενες, τότε ο δείκτης της δομής θα προχωρήσει μία θέση. Όμως υπάρχει περίπτωση η θέση που θα πάει να έχει την ίδια τιμή το row id με το προηγούμενο. Συνεπώς όλα τα ζευγάρια που βρέθηκαν προηγουμένως θα πρέπει να δημιουργηθούν και με το νέο αυτό row id της δομής Result. Άρα ξαναγυρνάμε τον δείκτη του πίνακα πίσω στη θέση από όπου ξεκίνησαν οι επαναλαμβανόμενες τιμές και δημιουργούμε τον κατάλληλο αριθμό από tuples. Για κάθε νέο tuple αυξάνουμε τον αριθμό των εγγραφών στην ενδιάμεση δομή Result.

Αν κάποια στιγμή οι δύο δείκτες δείχνουν σε δύο row ids για τα οποία ισχύει ότι η τιμή του row id της δομής είναι μεγαλύτερη από την τιμή του row id του πίνακα, τότε ουσιαστικά απλώς προσπερνάμε το row id αυτό του πίνακα και μετακινούμε τον δείκτη. Δεν έχει θέση δηλαδή στο τελικό αποτέλεσμα αυτό το row id του δεύτερου πίνακα. Από την στιγμή που και η δομή Result είναι ταξινομημένη ως προς τη στήλη του πρώτου πίνακα δεν υπάρχει περίπτωση να συναντήσουμε πιο κάτω στη δομή κάποιο row id με τέτοια τιμή. Όσο βρίσκουμε row id στο πίνακα που η τιμή του είναι μικρότερη από του row id της δομής προχωράμε τον δείκτη του πίνακα.

Όλη αυτή η διαδικασία συνεχίζεται όσο οι δύο δείκτες δεν είναι NULL. Ωστόσο αν τελειώσουν οι εγγραφές στον πίνακα Index και βγούμε από την επανάληψη θα πρέπει να ελέγξουμε αν υπάρχουν κι άλλα tuples πιο κάτω στη δομή Result με τιμές μεγαλύτερες από

την τελευταία εγγραφή του πίνακα. Όλα αυτά τα tuples θα διαγραφούν από την δομή Result και θα μειωθεί αντίστοιχα το πλήθος των tuples της δομής. Αν ο αριθμός ο τελικός των εγγραφών της ενδιάμεσης δομής γίνει μηδέν τότε σταματάμε την εκτέλεση των υπόλοιπων κατηγορημάτων του query και εκτυπώνουμε απευθείας NULL .

Αφού, λοιπόν, ολοκληρωθεί η συνάρτηση Synchronize πρέπει να ενημερώσουμε τις τιμές των Cur_Tables όλων των πινάκων του query. Για τους δύο πίνακες που συμμετείχαν στο join στο πεδίο της στήλης θα μπει το νούμερο της στήλης που συμμετείχε στο join καθώς η ενδιάμεση δομή είναι πλέον ταξινομημένη ως προς τις στήλες αυτές. Για όλους τους υπόλοιπους πίνακες που βρίσκονται μέσα στη δομή αυτή η μεταβλητή θα γίνει -1, κάτι που όπως έχουμε πει σημαίνει ότι ο πίνακας ναι μεν είναι στη δομή Result, δεν είναι όμως ταξινομημένος ως προς κάποια στήλη του. Επίσης όλων των πινάκων που βρίσκονται στην δομή ο αριθμός των γραμμών θα γίνει ίσος με τον αριθμό των tuples που είναι πλέον στη δομή.

3^{η} περίπτωση (κανείς από τους δύο πίνακες στη δομή Result):

Αν κανείς από τους δύο πίνακες δε βρίσκονται στη δομή Result τότε καλούμε για κάθε έναν από αυτούς τη συνάρτηση CreateNewIndex και δημιουργούνται δύο Index πίνακες οι οποίοι περιέχουν όλα τα row ids των πινάκων που υπάρχουν μέχρι αυτή τη στιγμή, ταξινομημένα ως προς τη στήλη του κάθε ενός που συμμετέχει στο κατηγόρημα. Στη συνέχεια καλούμε τη συνάρτηση JoinList η οποία ουσιαστικά έχοντας δύο πίνακες ταξινομημένους κάνει την συγχώνευση αυτών, όπως κάναμε και στο πρώτο μέρος της εργασίας. Τα αποτελέσματα που προκύπτουν τα αποθηκεύει απευθείας μέσα στην ενδιάμεση δομή Result. Ενημερώνουμε έπειτα τα πεδία των Cur_Tables να έχουν στην μεταβλητή της στήλης τη στήλη του ο κάθε ένας που συμμετείχε στο join και στη μεταβλητή των γραμμών το σύνολο των tuples που προέκυψαν έπειτα από το join αυτό. Αν ο αριθμός αυτός είναι μηδέν τότε και πάλι σταματάμε τις διαδικασίες και εκτυπώνουμε NULL ως λύση αυτού του query.

2γ) ΥΠΟΛΟΓΙΣΜΟΣ ΤΩΝ PREDICATES

Αφού ολοκληρώσουμε την εκτέλεση όλων των κατηγορημάτων στην ενδιάμεση δομή Result θα βρίσκονται μόνο όσες εγγραφές ανήκουν στο τελικό αποτέλεσμα. Από αυτές τις εγγραφές, λοιπόν, θα πρέπει να πάρουμε τα τελικά αθροίσματα. Για κάθε μία από τις προβολές, που θυμίζω αποθηκεύσαμε σε δομή τύπου PROJECTION θα καλέσουμε την

συνάρτηση FindResult, η οποία διατρέχει την δομή Result και προσθέτει τις τιμές των row ids του πίνακα και της στήλης αυτού που γίνεται η προβολή. Το τελικό άθροισμα το αποθηκεύουμε σε έναν πίνακα. Αφού υπολογίσουμε όλες τις προβολές του query έχουμε ολοκληρώσει την διαδικασία και είμαστε έτοιμοι να ασχοληθούμε με το επόμενο.

Όταν διαβάσουμε F θα εκτυπώσουμε στην οθόνη όλους τους πίνακες με τα αθροίσματα των queries αυτού του batch και μετά θα τα διαγράψουμε αφού δε τα χρειαζόμαστε άλλο.

ΔΟΜΕΣ:

Για τις λειτουργίες που αναλύθηκαν στο κομμάτι αυτό χρειάστηκε η ακόλουθη δομή:

Solution* Sol_List

Σε μία λίστα τέτοιας δομής αποθηκεύονται τα αθροίσματα ενός batch. Κάθε κόμβος αποτελεί τα αθροίσματα ενός query του batch. Περιέχει έναν μετρητή που δηλώνει το πόσες προβολές έχει το συγκεκριμένο query και έναν πίνακα με τόσες θέσεις όσες είναι και οι προβολές. Στον πίνακα αυτό αποθηκεύονται τα αθροίσματα που υπολογίστηκαν προηγουμένως.

Υλοποίηση 2^{ου} αλγορίθμου (πιο αργός από τον πρώτο που επιλέχθηκε τελικά):

Ο αλγόριθμος έχει υλοποιηθεί στα αρχεία statistics.c , statistics.h.

Στην αρχή καλείται η arrayStatsInit προκειμένου να αρχικοποιηθούν οι δομές που χρησιμοποιούνται για την εύρεση των στατιστικών.

Στη συνέχεια καλείται η commandListStatsCost, όπου παίρνει σαν όρισμα commandListHead, stats. Η συγκεκριμένη συνάρτηση κάνει sort τις εντολές κάθε query με βάση τα κόστη τους. Στην αρχή βάζει πρώτα στην τελική λίστα όλα τα φίλτρα, και με βάση αυτά ξανά υπολογίζει τα κόστη του πίνακα. Στη συνέχεια κάνει bubblesort και επιλέγει το πιο φθηνό node (μικρότερο fA), το προσθέτει στη λίστα και ξανά υπολογίζει τα κόστη. Έπειτα ξανακάνει bubblesort, επιλέγει το επόμενο φθηνότερο node που μπορεί να μπει στη λίστα, και κάνει ξανά υπολογισμό κόστους. Αυτό επαναλαμβάνεται μέχρι όλα τα nodes να μπουν στην τελική λίστα. Τέλος επιστρέφει την τελική-sorted commandList.

Η χρήση της bubblesort, γίνεται προκειμένου να αποφευχθούν οι πολλές προσπελάσεις του πίνακα(nodes), σε περίπτωση που το πρώτο (min cost)node δεν μπορούσε να εκτελεστεί. Εάν το πρώτο (min cost)node δεν συνδεόταν με τα προηγούμενα που έχουν μπει στην τελική λίστα, και δεν είχαμε κάνει bubblesort, τότε θα έπρεπε να προσπελάσουμε από την αρχή τον πίνακα με τα nodes και να βρούμε το αμέσως φθηνότερο σε κόστος node (κρατώντας επιπλέον δομές για το ποια node ως τώρα έχουμε ελέγξει). Αντί αυτού γίνεται

στην αρχή η bubblesort, και σε περίπτωση μη-συμβατού node, απλά το προσπερνάμε και κρατάμε το επόμενο (φθηνότερο- συμβατό).

Προσοχή! Προκειμένου να μικρύνουμε την πολυπλοκότητα, κάθε φορά που προσθέτουμε ένα node στην τελική λίστα, αυξάνουμε τη μεταβλητή offset κατά ένα και κάνουμε swap το node που βρίσκεται στη θέση offset, με αυτό που βάλαμε στη νέα λίστα. Κάθε καινούρια προσπέλαση ξεκινά με αρχικό node το offset+1.

3) 3^{ο} ΜΕΡΟΣ: ΧΡΟΝΟΙ ΥΛΟΠΟΙΗΣΗΣ ΚΑΙ ΣΤΑΤΙΣΤΙΚΑ

Αρχικά, από το δεύτερο μέρος της εργασίας, ο χρόνος υλοποίησης του αρχείου medium ήταν 3.48 min ενώ ο χρόνος του small 3.45 second. **(Σημείωση: όλοι οι χρόνοι που καταγράφονται αποτελούν μέσους όρους αποτελεσμάτων από πολλές δοκιμές στα μηχανήματα τις σχολής τα οποία παρουσιάζουν μικροδιαφορές από εκτέλεση σε εκτέλεση.)**

Η επιτάχυνση της εκτέλεσης χωρίζεται σε δύο μέρη, τη χρήση *στατιστικών και optimization* των κατηγορημάτων και την *παραλληλοποίηση* μέρους του κώδικα.

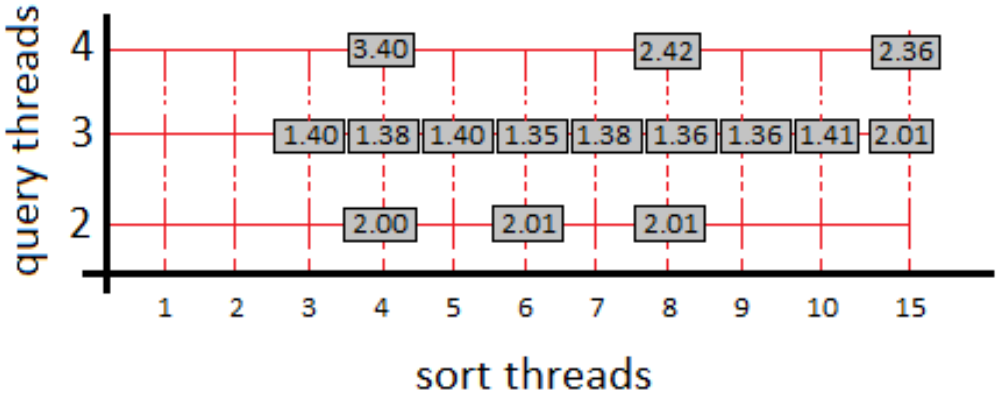
Μετά την χρήση των στατιστικών που προτείνονται στην εκφώνηση ο χρόνος του αρχείου medium έγινε 3.16 min και έπειτα από βελτιστοποίηση των στατιστικών ώστε όταν παρουσιάζεται query με αποτέλεσμα NULL να μην ξεκινά η διαδικασία solving ο χρόνος έφτασε τα 3.03 min.

Έπειτα από την παραλληλοποίηση των Queries (χωρίς χρήση στατιστικών) ο χρόνος εκτέλεσης του medium ήταν 2.21 min, εφόσον ο Job Scheduler περιμένει το τέλος κάθε batch για να ξεκινήσει το επόμενο. (*Σημείωση*: με τη δημιουργία 5 ή περισσοτέρων νημάτων για τα queries υπάρχει πιθανότητα η διεργασία να γίνει Killed λόγω έλλειψης χώρου καθώς ορισμένα συνεχόμενα queries στο πρώτο batch έχουν μεγάλες χωρικές απαιτήσεις.)

Τέλος, έπειτα από την παραλληλοποίηση σε επίπεδο Query και Sorting αλλά και την χρήση του ιδανικού optimization των κατηγορημάτων ο χρόνος έπεσε κάτω από 2 λεπτά και εξαρτάται πλέον από την εκλογή του αριθμού των νημάτων για Queries, για Sorting, αλλά και τον αριθμό jobs στα οποία χωρίζεται ένα sorting για να δοθεί στα threads BUCKET_LIMIT. Παραλληλοποίηση σε join, ενώ υλοποιήθηκε αποδείχθηκε πως επιβράδυνε την διεργασία ελαφρώς οπότε αφαιρέθηκε.

Παρατήρηση: Ο μικρότερος χρόνος που έχει παρατηρηθεί για το αρχείο small ήταν 2.01 δευτερόλεπτα κατά την παραλληλοποίηση μόνο σε επίπεδο query. Αντίθετα αυτή τη στιγμή που έχει προστεθεί παραλληλοποίηση και στο sort, το small εκτελείται σε 3.30 δευτερόλεπτα περίπου.

Παρακάτω παρουσιάζεται διάγραμμα που στον άξονα y έχει τον αριθμό των query threads, στον x τον αριθμό των sort threads και καταγράφει τα αποτελέσματα σε λεπτά.



Παρατηρείται ότι έχουμε τα καλύτερα αποτελέσματα όταν τα query threads = 3 και sort threads = 6-8. Έτσι, βασισμένοι στα αποτελέσματα αυτά, στο παρακάτω διάγραμμα παρουσιάζονται οι χρόνοι με τις συγκεκριμένες παραμέτρους όταν αλλάζουμε την τιμή του BUCKET_LIMIT.

BUCKET_LIMIT	3q 6s	3q 7s	3q 8s
3	1.38	1.39	1.37
4	1.34	1.38	1.37
5	1.35	1.39	1.35
6	1.35	1.37	1.33
7	1.45	1.39	1.52

Εν κατακλείδι, οι ταχύτεροι χρόνοι είναι δυνατοί όταν τα Query Threads είναι 3, τα Sort Threads είναι από 6 έως 8 και το BUCKET_LIMIT είναι από 4 έως 6.

Παραλληλοποίηση Join:

Η παραλληλοποίηση της λειτουργίας join πραγματοποιήθηκε και βρίσκεται στα αρχεία με κατάληξη _JOIN όμως επιλέχθηκε να μη χρησιμοποιηθεί καθώς δεν επιτάχυνε την διαδικασία.