Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Radix Hash Join

- Νεαμονίτης Ευάγγελος, **A.M.**: 1115201400123
- Ορφανίδης Ελευθέριος, **A.M.**: 1115201400133

Compilation: make
Execute: Για να εκτελέσετε το testharness, αφού κάνετε make εκτελέστε την εντολή **./submission/runTestharness.sh** . Για να τρέξετε χειροκίνητα το πρόγραμμα με δικά σας query κάντε cd submission/build/release και τρέξτε το πρόγραμμα με την εντολή: **./radixhash < small.init** . Στην συνέχεια γράφετε Done και πατάτε enter, πλέον το πρόγραμμα είναι έτοιμο να δεχτεί queries. Μόλις εισάγεται το batch γράφετε F ώστε να εκτελεστεί και να εμφανιστούν τα αποτελέσματα και για να σταματήσετε το πρόγραμμα γραψτε απλα Exit. Ο αριθμός των thread καθώς και ο αριθμός των n least significant bytes που θα χρησιμοποιηθουν για το hashing των δεδομένων γίνεται define στο structs.h .

Συνοπτική περιγραφή αλγορίθμου

Αρχικά διαβάζουμε απο την είσοδο τα path των relations που θα χρειαστούν για την εκτέλεση του προγράμματος, μέχρι να δωθεί η λέξη Done, όπου και σταματάμε και δημιουργούμε την δομή **relation_map.** Το rel_map περιέχει για κάθε σχέση όλα τα tuples αποθηκεύμενα σε row stored format καθώς και τα αρχικά στατιστικά της. Στην συνέχεια το πρόγραμμα είναι έτοιμο να δεχτεί queries τα οποία τα αποθηκεύει σε μία λίστα ώστε να τα εκτελούμε σε batches. Κάθε batch τελειώνει με F, που μόλις διαβαστεί, ξεκινάμε και εκτελούμε σειριακά τα queries. Για κάθε query καλείται η ExecuteQuery η οποία δημιουργεί αρχικά την δομή των ενδιάμεσων αποτελεσμάτων, εκτελεί όλα τα φίλτρα και ανανεώνει τα στατιστικά των σχέσεων αυτών. Με την ολοκλήρωση των φίλτρων καλείται η JoinEnum που κάνει sort την predicate_list ώστε να εκτελέσουμε τα predicates με βέλτιστο τρόπο, μειώνοντας έτσι το πλήθος των ενδιάμεσων αποτελεσμάτων. Τέλος, εκτελούμε τα υπόλοιπα predicates με τον παρακάτω αλγόριθμο:

Ένωση:

Η συνάρτηση RadixHashJoin δέχεται ως όρισμα δύο δομές **relation** που η κάθε μια είναι ένας πίνακας από δομές tuple. Η δομή **tuple** περιέχει μια σειρά μιας σχέσης αποθηκευμένης κατά στήλες (RowId, Value). Μόλις οι δυο relation περάσουν τον κατακερματισμό με την H1 δημιουργούνται δύο δομές **reordered_relation**, που περιέχουν το psum, το histogram καθώς και την δομή relation η οποία έχει γίνει sort όπως ορίζεται στην εκφώνηση. Στην συνέχεια, για κάθε bucket που έχει στοιχεία και από τις δύο σχέσεις δημιουργούμε το κατάλληλο ευρετήριο στον κάδο με το μικρότερο μέγεθος. Έστω R η σχέση με τα περισσότερα στοιχεία στον κάδο και S η μικρότερη σχέση για την οποία δημιουργήσαμε το ευρετήριο. Έτσι, διατρέχουμε κάθε τιμή της σχέσης R που ανήκει στο bucket και για κάθε τέτοια τιμή αναζητούμε στο ευρετήριο της σχέσης S αν υπάρχει κάποιο match. Αν βρούμε κάποιο ζεύγος row_id

που ταιριάζουν, τότε δημιουργούμε μία δομή **result_tuples** η οποία περιέχει 2 uint64_t που αντιστοιχούν σε: row_id_R και row_id_S και την εισάγουμε στην λίστα με τα αποτελέσματα μας. Μόλις ολοκληρωθεί αυτή η διαδικασία για όλα τα buckets, επιστρέφουμε την λίστα των αποτελεσμάτων.
Τα φίλτρα εκτελούνται με την παρακάτων διαδικασία:

Φίλτρο:

Η συνάρτηση Filter δέχεται ως όρισματα τα ενδιάμεσα αποτελέσματα (περιγράφονται παρακάτω), το relation_map όπου είναι αποθηκευμένες όλες οι σχέσεις καθώς και ένα predicate που περιέχει φίλτρο. Αρχικά ελέγχουμε αν η σχέση που ορίζεται απο το predicate βρίσκεται μέσα στα ενδιάμεσα αποτελέσματα ή αν θα πρέπει να την κάνουμε access απο το relation_map. Στην συνέχεια κοιτάμε τι είδος φίλτρου έχουμε στο predicate (< , > , =) και πάμε στο κατάλληλο case. Διατρέχουμε όλη την σχέση (είτε απο τα ενδιάμεσα αποτελέσματα είτε απο το relation_map) και για κάθε tuple που ικανοποιεί το φίλτρο, εισάγουμε το row_id του σε μία λίστα result. Μόλις ολοκληρωθεί αυτή η διαδικασία επιστρέφουμε την λίστα των αποτελεσμάτων.

Ενδιάμεσα αποτελέσματα

Η δομή των ενδιάμεσων αποτελεσμάτων αποτελείται από έναν πίνακα table (int**) διαστάσεων [num_of_relations] x [num_tuples] , όπου num_tuples ο αριθμός των αποτελεσμάτων που υπάρχουν στην λίστα result την οποία θα εισάγουμε στα intermediate results. Επίσης δίνεται η δυνατότητα να δείχνει σε άλλο κόμβο ενδιάμεσων αποτελεσμάτων σε περίπτωση που υπάρχει ήδη κόμβος inter_res με ενεργές σχέσεις, αλλά εμείς κάνουμε κάποιο κατηγόρημα στο οποίο όλες οι σχέσεις είναι ανενεργές. Με αυτόν τον τρόπο μπορούμε να έχουμε παραπάνω instances των ενδιάμεσων αποτελεσμάτων, αν τα κατηγορήματα σε ένα query δεν συνδέονται μεταξύ τους. Αρχικά τα ενδιάμεσα αποτελέσματα αρχικοποιούνται με NULL σε όλες τις θέσεις του πίνακα table[num_of_relations] (καθώς καμία σχέση δεν είναι ενεργή). Στην συνέχεια:

- Αν εκτελεστεί κάποιο φίλτρο, εισάγουμε σε έναν άδειο κόμβο inter_res τα αποτελέσματα του φίλτρου. Το table του κόμβου έχει διαστάσεις [num_of_relations] x [num_tuples] αλλά η μόνη ενεργή σχέση είναι αυτή στην οποία εφαρμόσαμε το φίλτρο.
- Αν έχουμε κάποιο join τότε ελέγχουμε τις εξής περιπτώσεις:
    - Να μην είναι καμία σχέση ενεργή στα ενδιάμεσα αποτελέσματα. Σε αυτή την περίπτωση εισάγουμε τα row_ids των σχέσεων που βρίσκονται μέσα στην λίστα result σε έναν άδειο κόμβο των ενδιάμεσων αποτελεσμάτων.
    - Να είναι ενεργή μία από τις 2 στα ενδιάμεσα αποτελέσματα. Σε αυτή την περίπτωση βρίσκουμε τον κόμβο των ενδιάμεσων αποτελεσμάτων στον οποίο είναι ενεργή η σχέση και δημιουργούμε ένα καινούριο table (temp_array) με num_tuples τον αριθμό των αποτελεσμάτων που βρίσκονται στην λίστα result. Σε αυτή την περίπτωση όμως τα row_ids της ενεργής σχέσης που βρίσκονται στην λίστα result δεν αναφέρονται στις γραμμές της αρχικής σχέσης, αλλά στις γραμμές του προηγούμενου instance των ενδιάμεσων αποτελεσμάτων. Έτσι εκτελούμε τον ακόλουθο αλγόριθμο:
        - Για κάθε i που είναι αποτέλεσμα στην λίστα results:
            - Αναθέτω στην μεταβλητή old_pos το row_id της ενεργής σχέσης.
            - Για κάθε σχέση j που ήταν ενεργή στον ίδιο κόμβο με την δική μας σχέση:
                - temp_array[j][i] = table[j][old_pos] . Αντιγράφουμε δηλαδή σε κάθε θέση i του καινούριου table όλα τα περιεχόμενα της πλειάδας

old_pos που δείχνει στο προηγούμενο instance των ενδιάμεσων αποτελεσμάτων.

- Μετά την αντιγραφή όλων των αποτελεσμάτων, το παλιό instance δεν μας χρειάζεται και το διαγράφουμε. Στην θέση του βάζουμε το καινούριο table (temp_array) που δημιουργήσαμε.

- Αν έχουμε join μεταξύ 2 σχέσεων που είναι ενεργές στα ενδιάμεσα αποτελέσματα, αλλά σε διαφορετικούς κόμβους τότε ακολουθούμε την ίδια διαδικασία με το από πάνω bullet , αλλά χρειάζεται και η κλήση της συνάρτησης MergeResults η οποία αναζητά στους κόμβους των ενδιάμεσων αποτελεσμάτων αν υπάρχει κάποια σχέση ενεργή σε παραπάνω από ένα κόμβους. Αν βρει τέτοια σχέση τότε γνωρίζουμε ότι τα row_ids αυτής της σχέσης στον πρώτο κόμβο αναφέρονται στις γραμμές της σχέσης στον δεύτερο κόμβο (ο δεύτερος κόμβος περιέχει τα row_ids που αναφέρονται στις γραμμές της αρχικής σχέσης). Έτσι ενώνουμε τους δύο αυτούς κόμβους εισάγοντας σε κάθε γραμμή του πρώτου κόμβου, ολόκληρη την πλειάδα του δεύτερου κόμβου την οποία μας ορίζει η τιμή first_node[active_relation][i] (αντιστοιχεί σε row_id του δεύτερου κόμβου).
- Αν κάποιο join μας επιστρέψει NULL ως αποτέλεσμα, αποδεσμέυουμε τις δομές που χρησιμοποιούμε για το συγκεκριμένο query, τυπώνουμε NULL για κάθε ζητούμενη προβολή και συνεχίζουμε στο επέμενο query. Ακολουθούμε αυτή την λογική γιατί είναι άσκοπο να υπολογίσουμε τα υπόλοιπα predicates του query καθώς όποια και αν είναι τα αποτελέσματα τους στο τέλος θα συνδυαστούν με κάποια απο τις σχέσεις που έχει 0 tuples και θα έχουμε συνολικό αποτέλεσμα 0. Με αυτόν τον τρόπο εξοικονομούμε και χρόνο και μνήμη.

Πολυνηματισμός

Job Scheduler: Η δομή scheduler αποτελείται από έναν πίνακα που περιέχει δείκτες στα thread (ο αριθμός των thread δίνεται ως όρισμα στην συνάρτηση SchedulerInit), από την ουρά των jobs (δομή jobqueue_node, θα εξηγηθεί παρακάτω) . Επίσης περιέχει έναν σημαφόρο στον οποίο κάνουν wait() τα thread και κάνει post() η PushJob μόλις ολοκληρώσει μία εισαγωγή δουλειάς στην ουρά. Με αυτόν τον τρόπο τα ενεργά threads δεν κάνουν busy waiting περιμένοντας να τους ανατεθεί δουλειά, αλλά “ενεργοποιούνται” από την ίδια την PushJob μόλις χρειαστεί.
Barrier: Επειδή τα είδη των Jobs που εισάγουμε στην ουρά γίνεται σειριακά (πρώτα τελειώνει το ένα είδος Job και μετά ξεκινάμε να εισάγουμε το επόμενο) , χρησιμοποιούμε μια μεταβλητή answers_waiting η οποία μόλις γίνει 0 μας δείχνει ότι τελείωσαν όλα τα Jobs που είχαν εισαχθεί στην ουρά. Το answers_waiting αρχικοποιείται σε:

1. thread_num πριν δημιουργηθούν τα HistJobs
2. thread_num πριν δημιουργηθούν τα PartitionJobs
3. answers πριν δημιουργηθούν τα JoinJobs, όπου answers είναι ο αριθμός των bucket τα οποία ειναι active και για τις 2 σχέσεις.

Μόλις ένα thread τελειώσει το Job του, καλεί την JobDone η οποία μειώνει το answers_waiting κατα ένα και ελέγχει αν η τωρινή του τιμή είναι ίση με 0. Αν είναι 0 τότε κάνει signal στην condition variable στην οποία περιμένει το Barrier.

Jobqueue_node: Κάθε κόμβος της ουράς του scheduler περιέχει έναν int ο οποίος ορίζει το είδος του Job (0 για HistJob, 1 για PartitionJob και 2 για JoinJob) ένα void* για τα ορίσματα της συνάρτησης και έναν δείκτη στον επόμενο κόμβο. Το ThreadFunction μόλις ξυπνήσει ελέγχει αν η μεταβλητή exit_all

είναι ίση με 1, αν είναι τότε κάνει pthread_exit, αλλιώς εκτελεί την κατάλληλη συνάρτηση ανάλογα με την τιμή της job->function.

Thread Jobs:

1. **HistJobs:** Για την δημιουργία των ιστογραμμάτων , δημιουργούμε number_of_threads HistJobs τα οποία παίρνουν ένα τμήμα του πίνακα το καθένα και υπολογίζουν ιστόγραμμα για το αντίστοιχο κομμάτι. Ο διαχωρισμός του πίνακα γίνεται με τέτοιο τρόπο ώστε τα πρώτα n-1 threads να έχουν τον ίδιο αριθμό στοιχείων και το τελευταίο thread να παίρνει όσες παραπάνω τιμές έχουν παραμείνει απο την διαίρεση (tuples / thread_num). Πριν την δημιουργία των HistJobs δημιουργούμε ένα πίνακα μεγέθους thread_num, κάθε θέση του οποίου περιέχει ένα δείκτη σε πίνακα histogram (πίνακας που περιέχει int και έχει μέγεθος τον αριθμό buckets) . Με αυτόν τον τρόπο κάθε thread παίρνει ως όρισμα την αντίστοιχη θέση αυτού του πίνακα και δημιουργεί το κατάλληλο histogram. Μόλις ολοκληρωθούν όλα τα HistJobs διατρέχουμε μία φορά κάθε κουβά του κάθε πίνακα και δημιουργούμε το ολικό histogram της σχέσης μας.
2. **PartitionJobs:** Μόλις δημιουργήσουμε το Hist και το Psum κάνουμε allocate την reordered_relation και αναθέτουμε στα PartitionJob το τμήμα του reordered array που πρέπει να συμπληρώσουν. Ο τρόπος που γίνεται αυτο είναι ο εξής:
    a. Βρίσκουμε αρχικά τον κουβά στον οποίο ανήκει το start καθώς και πόσες τιμές αυτού του κουβά πρέπει να κάνουμε skip μέχρι να συμπληρώσουμε το πρώτο στοιχείο.
    b. Στην συνέχεια διατρέχουμε τον reordered array απο το start μέχρι το end και για κάθε στοιχείο που πρέπει να εισάγουμε:
        i. Διατρέχουμε τον original array απο το previous_encounter (αρχικοποιείται σε 0) μέχρι το τέλος. Χρησιμοποιούμε το previous_encounter ώστε να μην διατρέχουμε τον original_array καθε φορα από την αρχή ενώ γνωρίζουμε που βρίσκεται η τελευταία τιμή του current_bucket.
        ii. Κάνουμε hash την τιμή του original array και αν αυτή είναι ίση με τον αριθμό του κουβά που βρισκόμαστε τώρα, ελέγχουμε αν η τιμή skip_values είναι 0. Αν δεν είναι 0, την μειώνουμε κατα 1 και συνεχίζουμε να διατρέχουμε τον original array. Αν είναι 0, τότε την εισάγουμε στον reordered array , ενημερώνουμε το previous_encounter με την θέση του original array που βρισκόμαστε τώρα και ελέγχουμε αν η επόμενη τιμή που πρέπει να εισάγουμε είναι στον ίδιο bucket ή όχι. Αν δεν είναι στον ίδιο bucket, τότε ενημερώνουμε την μεταβλητή current_bucket και αναθέτουμε στο previous_encounter την τιμή 0.
3. **JoinJobs:** Μετά την δημιουργία των reordered relations για τις 2 σχέσεις, αν δεν είναι κάποια από τις 2 ίση με NULL τότε διατρέχουμε τα hist arrays των δυο σχέσεων και δημιουργούμε JoinJobs για κάθε bucket που έχει στοιχεία και από τις δύο σχέσεις. Πριν δημιουργησουμε τα JoinJobs εχουμε διατρέξει τα histograms των δύο σχέσεων ώστε να ξέρουμε πόσα JoinJobs θα δημιουργηθούν και να δημιουργήσουμε τον πίνακα res_array του οποίου το κάθε κελί δείχνει σε μία λίστα result με μέγεθος buffer 128KB. Το μέγεθος του res_array είναι όσο και ο αριθμός των JoinJobs που θα δημιουργήσουμε. Μόλις ολοκληρωθούν τα JoinJobs καλείται η MergeResults που δημιουργεί μια ενιαία λίστα με μεγαλύτερο buffer και την επιστρέφουμε.

Query Optimization

Αλγόριθμος: Ο σκοπός του JoinEnum είναι να επιλέξουμε με κατάλληλη σειρά εκτέλεσης των predicates ώστε να μειώσουμε τον αριθμό των ενδιάμεσων αποτελεσμάτων. Ο αλγόριθμος που χρησιμοποιήσαμε για να το πετύχουμε, είναι αυτός που δόθηκε στην εκφώνηση. Έχουμε μια τριπλή επανάληψη όπου:

1. Η πρώτη ελέγχει το μέγεθος των relation που βρίσκονται στο υποσύνολο S.
2. Η δεύτερη, ανάλογα με το μέγεθος του υποσυνόλου, διατρέχει το best_tree και επιλέγει τα κατάλληλα υποσύνολα.
3. Η τρίτη επιλέγει τις σχέσεις Rj που δεν ανήκουν στο S, και ελέγχει αν είναι Connected. Για να είναι connected πρέπει να υπάρχει κάποιο predicate που περιέχει την Rj και μία σχέση απο αυτές που ανήκουν στο S. Η συνάρτηση Connected επιστρέφει NULL αν δεν είναι συνδεδεμένα τα Rj, S αλλιώς επιστρέφει τον predicate_listnode με τον οποίο συνδέονται. Στην συνέχεια χτίζουμε το δέντρο χρησιμοποιώντας το best_Tree[S] και κάνοντας append σε αυτό το predicate που συνδέει την Rj . Πλέον έχουμε μία σειρά εκτέλεσης για το υποσύνολο S' (S' = S ∪ Rj) την οποία κοστολογούμε. Αν το κόστος της είναι μικρότερο από το best_Tree[S'] τότε το αντικαθιστούμε. Τέλος επιστρέφουμε την σειρά εκτέλεσης των predicates που υπάρχει στην τελευταία θέση του best_Tree.

Μετά την ολοκλήρωση αυτής της διαδικασίας, αν υπάρχουν παραπάνω από ένα join μεταξύ δύο ίδιων σχέσεων τότε στα predicates της τελευταίας θέσης του best_Tree θα περιέχεται μόνο το βέλτιστο απο αυτά τα join. Για αυτό τον βρίσκουμε τα υπόλοιπα predicates που δεν περιέχονται στην τελική λίστα του best_Tree και τα τοποθετούμε αμέσως μετά απο το βέλτιστο join, γιατί λειτουργούν σαν φίλτρο στην δομή των ενδιάμεσων αποτελεσμάτων. Αυτό το κάνουμε γιατί μειώνουμε το πλήθος των tuples αυτών των σχέσεων πριν γίνουν join με κάποια άλλη σχέση.

BestTree: Για να κρατάμε τις βέλτιστες εκτελέσεις χρησιμοποιούμε το hashtable BestTree που έχει $2^{relations}$ θέσεις. Η θέση 0 του πίνακα είναι κενή καθώς μας ενδιαφέρουν τα bit που έχουν την τιμή 1 στην δυαδική αναπαράσταση του index του πίνακα. Αποθηκεύουμε έτσι τα στοιχεία της R_1 στο index 1 (001), της R_2 στο index 2 (010), της R_3 στο index 4 (100) κοκ. Με αυτόν τον τρόπο είναι εύκολο και φθηνό να κάνουμε access κάθε υποσύνολο συγκεκριμένου πλήθους σχέσεων. Διατρέχοντας το BestTree γνωρίζουμε ότι αν το πλήθος των bit του index είναι 1, τότε έχουμε υποσύνολο με μόνο μία σχέση μέσα, αν είναι 2 τότε έχουμε υποσύνολα με 2 σχέσεις κοκ. Για παράδειγμα η θέση με index 5 με δυαδική αναπαράσταση 101 , είναι ένα σύνολο που περιέχει τις σχέσεις R_1 , R_3 .

Κάθε θέση του BestTree περιέχει:

- Μία λίστα με τα predicates που περιέχει η σχέση καθώς και τον αριθμό των predicates.
- Τον αριθμό των active bits που έχει το index της συγκεκριμένης θέσης ώστε να ξέρουμε το πλήθος των σχέσεων που περιέχει η συγκεκριμένη θέση
- Το κόστος της συγκεκριμένης θέσης με τον τωρινό συνδυασμό που περιέχεται στην λίστα με τα predicates.
- Τα στατιστικά των original σχέσεων για κάθε column.

Υπολογισμός κόστους: Μετά την εκτέλεση όλων των joins, ανεξάρτητα από τη σειρά με την οποία εκτελέστηκαν, θα πάρουμε τον ίδιο αριθμό από tuples. Έτσι σε κάθε βήμα του αλγόριθμου για το νέο κόστος του CurrTree προσθέτουμε τον αριθμό των tuples μετά την εκτέλεση του predicate στο μέχρι τότε κόστος. Με τον τρόπο αυτό όσο λιγότερα tuples έχουν τα ενδιάμεσα αποτελέσματα τόσο μικρότερο βγαίνει το συνολικό κόστος. Ένα παράδειγμα είναι το εξής: το Join μεταξύ των σχέσεων 0 και 1 παράγει 7000 tuples ενώ το Join μεταξύ των σχέσεων 1 και 2 παράγει 2000 tuples. Μετά την εκτέλεση και των δυο Joins το αποτέλεσμα έχει 3000 tuples. Έτσι με την συνάρτηση κόστους που υλοποιήσαμε το δέντρο (01)2 θα δώσει κόστος 7000 + 3000 = 10000 ενώ το δέντρο (12)0 θα δώσει κόστος 2000 + 3000 = 5000 και θα είναι και αυτό που θα επιλεγεί ως καλύτερο.

Βελτιστοποιήσεις

- **InsertResult:** Η InsertResult επιστρέφει τον κόμβο στον οποίο έγινε η τελευταία εισαγωγή, δίνοντας μας έτσι την δυνατότητα οι εισαγωγές στην δομή result να γίνεται χωρίς να χρειαστεί να διατρέξουμε όλη την λίστα μέχρι τον τελευταίο ελεύθερο κόμβο. Με αυτόν τον τρόπο μειώθηκε η πολυπλοκότητα της εισαγωγής των αποτελεσμάτων του join στην δομή result καθώς στην χειρότερη περίπτωση θα γίνει ένα άλμα από τον κόμβο που στέλνουμε στην InsertResult μέχρι να βρεθεί κατάλληλος κόμβος για εισαγωγή.
- **InsertRowIdResult:** Η InsertRowIdResult τροποποιήθηκε ώστε να λειτουργεί όπως λειτουργεί και η InsertResult μειώνοντας έτσι άσκοπα traverses της λίστας των αποτελεσμάτων κατα την εισαγωγή τους.
- **FindResultTuples:** Μέχρι και το 2ο part της εργασίας για να βρούμε το i-οστό στοιχείο μέσα σε μια λίστα result χρησιμοποιούσαμε την FindResultTuples η οποία δεχόταν την κεφαλή της λίστας και έψαχνε σειριακά τους κόμβους μέχρι να βρει αυτόν που περιέχει το στοιχείο που αναζητάμε. Αυτή η διαδικασία ήταν πολύ αργή, ειδικά όταν η λίστα των αποτελεσμάτων μας αποτελούνταν από πολλούς κόμβους. Για αυτό τον λόγο αντί να καλούμε την FindResultTuples, κρατάμε ένα προσωρινό δείκτη στην κεφαλή της λίστας των αποτελεσμάτων και διαβάζουμε αμέσως το αποτέλεσμα που αναζητούμε με μόνο overhead αυτής της διαδικασίας να είναι ο έλεγχος για το εάν πρέπει ο δείκτης να προχωρήσει στον επόμενο κόμβο ή όχι.
- **MergeResults:** Η αρχική λειτουργία της MergeResults ήταν να παίρνει τον res_array ως όρισμα και να ενώνει τις λίστες κάθε θέσης σε μια ενιαία λίστα. Παρατηρήσαμε όμως ότι αυτό οδηγούσε σε μια λίστα η οποία είχε πολλούς κόμβους, οι οποίοι συνήθως περιείχαν πολύ λίγα στοιχεία κάτι το οποίο καθυστερούσε την διαδικασία εύρεσης κάποιου συγκεκριμένου στοιχείου κατα την δημιουργία των ενδιάμεσων αποτελεσμάτων μετά το join. Για αυτό τον λόγο αλλάξαμε την λειτουργία της συνάρτησης έτσι ώστε να μην ενώνει απλά τις ηδη υπάρχουσες λίστες με μέγεθος buffer 128KB , αλλά να δημιουργεί μία νέα λιστα με μεγεθος buffer 1MB (το μέγεθος που χρησιμοποιούσαμε στα πρώτα 2 part της εργασίας) και να γράφει σε αυτή την νέα όλα τα περιεχόμενα των μικρών λιστών. Αυτή η αλλαγή, αν και καθυστερεί την εκτέλεση της MergeResults, βελτιώνει αισθητά την εκτέλεση του testharness καθώς έχουμε πολύ πιο γρήγορο access στα δεδομένα της λίστας.
- **PartitionJobs:** Στην αρχική του λειτουργία το PartitionJob χρησιμοποιούσε ένα αντίγραφο του Psum για κάθε thread ώστε να αυξάνει την τιμή του Psum[current_bucket] κάθε φορά που εισάγαμε ένα στοιχείο στον reordered array. Αυτό οδηγούσε στην άσκοπη δημιουργία

thread_num πινάκων για κάθε σχέση που ήταν να γίνει join, καθυστερώντας την εκτέλεση του προγράμματός μας. Στην τελική της έκδοση, η συνάρτηση δεν χρησιμοποιεί καποιο αντίγραφο του psum, αλλα τον πρωτότυπο πίνακα, και χρησιμοποιεί μια μεταβλητή η οποία κάνει track πόσες τιμές απο το current_bucket έχουν εισαχθεί, ώστε το current_bucket να αλλάζει σωστά όταν εισαχθούν όλες οι τιμές του. Επίσης στις βελτιώσεις του PartitionJob είναι και η χρήση της μεταβλητής previous_encounter που μας επιτρέπει να αποφύγουμε άσκοπα σκαναρίσματα του relation.

Running Times

Όλες οι δοκιμές και οι χρονομετρήσεις πραγματοποιήθηκαν σε PC με τα εξής χαρακτηριστικά:

- CPU: Intel Core i3 4170 (2 cores / 4 threads)
- RAM: 8GB DDR3 (dual channel)
- Λειτουργικό σύστημα: Ubuntu 18.04.1 LTS 64-bit

Επίσης όλοι οι αναγραφόμενοι χρόνοι αποτελούν μέσο όρο απο 5 εκτελέσεις.

Τέλος του part 2: Το πρόγραμμα μας, στο τέλος του 2ου part δεν περιείχε ούτε τις βελτιώσεις που αναφέρονται παραπάνω, ούτε multithread λειτουργία για την επιτάχυνση του RadixHashJoin. Επίσης το query optimization ακολουθούσε την παρακάτω λογική:

1. Από τα predicates, επιλέγαμε πρώτα όλα τα φίλτρα.
2. Στην συνέχεια, αφού έχουμε εκτελέσει όλα τα φίλτρα, επιλέγουμε κάποιο join το οποίο έχει τουλάχιστον μία σχέση ενεργή στα ενδιάμεσα αποτελέσματα. Αν δεν υπάρχει τέτοιο join τότε απλά επιλέγουμε το πρώτο join που βρίσκεται στην predicates list.

Αυτή η έκδοση του προγράμματος μας αποτελεί και την βάση των βελτιώσεων που πραγματοποιήσαμε ώστε να μειώσουμε τον χρόνο εκτέλεσης σε πρώτη φάση, και την μνήμη που καταλαμβάνει κατά την εκτέλεση του σε δεύτερη φάση. Η συγκεκριμένη έκδοση τρέχει το testharness.cpp σε: **7.8 δευτερόλεπτα** καταλαμβάνοντας συνολικά: **1,12 GB μνήμης.**

Με τις βελτιστοποιήσεις και την multithread λειτουργία: Στην συνέχεια χρησιμοποιώντας τις παραπάνω βελτιστοποιήσεις και την multithread εκτέλεση του RHJ, χωρίς όμως να αλλάξουμε τον τρόπο που επιλέγουμε τα queries ο χρόνος εκτέλεσης του testharness μειώθηκε στα : **1,45 δευτερόλεπτα** καταλαμβάνοντας συνολικά: **1,5 GB μνήμης.**

Τελική έκδοση του προγραμματός μας:

N_LSB: n least significant bits , ο αριθμός που ορίζει τα bit τα οποία καθορίζουν σε ποιο bucket κατακερματίζεται κάθε τιμή.
Threads: ο αριθμός των thread που χρησιμοποιούνται για την παραλληλοποίση της προεπεξεργασίας και της διαδικασίας του join.

N_LSB	Threads	Χρόνος εκτέλεσης testharness
2	2	1.37 sec (1.4 s)

3	2	1.4 sec (1.42 s)
3	3	1.38 sec (1.36 s)
4	2	1.44 sec (1.49 s)
4	3	1.42 sec (1.45 s)
4	4	1.38 sec (1.39 s)
5	2	1.57 sec (1.66 s)
5	3	1.54 sec (1.6 s)
5	4	1.48 sec (1.52 s)
6	2	1.81 sec (2 s)
6	3	1.8 sec (1.92 s)
6	4	1.69 sec (1.77 s)