ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

- Join Queries Optimization (Radix Hash Join)-

Καλοπίσης Ιωάννης - Μυστιλόγλου Θεόδωρος
1115201500059 - 1115201500107
19 Ιανουαρίου 2019

Περιεχόμενα

1 Σύνοψη Εφαρμογής

Η εφαρμογή μας προσωμοιώνει την εκτέλεση επερωτήσεων (queries) sql σε γλώσσα C. Η ιδέα της εφαρμογής είναι στηριγμένη στον προγραμματιστικό διαγωνισμό SIGMOD 2018 και στόχος του προγράμματος είναι να τρέξει ένα σύνολο δωθέντων επερωτήσεων όσο πιο γρήγορα γίνεται. Το πρόγραμμα διαβάζει binary αρχεία που περιέχουν αριθμητικά δεδομένα και αντιπροσωπεύουν το σύνολο των σχέσεων της βασης δεδομένων μας, τρέχει sql κατηγορήματα πάνω σε αυτά σύμφωνα με ένα αρχείο επερωτήσεων και επιστρέφει τα αντίστοιχα αποτελέσματα. Για έλεγχο εγκυρότητας των αποτελεσμάτων και χρονομέτρηση γίνεται επικοινωνία με μερικά αρχεία που δίνονται από τον διαγωνισμό. Τα βήματα που ακολουθήσαμε, καθώς και οι λόγοι που επιλέξαμε αυτά τα βήματα έναντι άλλων, για να κάνουμε αυτήν την διαδικασία όσο πιο γρήγορη γίνεται αναλύονται εις βάθος στην συνέχεια.

2 Διαχείριση Επερωτήσεων

Προσομοιώνοντας την sql, οι επερωτήσεις περιέχουν 3 πεδία (FROM, WHERE, SELECT) που μας έρχονται με αυτήν τη σειρά και τα αναλύουμε. Στο πεδίο FROM μας ενημερώνει για το ποιες σχέσεις της ”βάσης δεδομένων” μας θα χρησιμοποιήθουν στην συγκεκριμένη επερώτηση. Το πεδίο WHERE περιέχει ένα πλήθος από τα εξής κατηγορήματα:

- Κατηγόρημα φίλτρου (Filter Predicate), όπου η στήλη ενός πίνακα σκανάρεται και κρατάμε μόνο τις εγγραφές οι οποίες ικανοποιούν το ζεύγος τελεστή και αριθμού. Οι τελεστές που υποστηρίζονται είναι οι ”>”,”<” και ”=”. Για παράδειγμα, το κατηγόρημα ”0.1>300” μας ζητάει να κρατήσουμε από τη στήλη 1 του πίνακα 0 μόνο τις γραμμές που η τιμή τους είναι > 300.

- Κατηγόρημα ζεύξης (Conjuction Predicate), όπου αφορά τις περιπτώσεις Join μεταξύ 2 διαφορετικών πινάκων ή αυτοσυσχέτιση για τον ίδιο πίνακα. Για παράδειγμα, το κατηγόρημα ”0.1=3.2” μας ζητάει να κάνουμε Join τη στήλη 1 του πίνακα 0 με τη στήλη 2 του πίνακα 3. Σε αυτήν την κατηγορία ανήκουν και περιπτώσεις όπου και οι 2 πίνακες βρίσκονται στον ίδιο πίνακα ενδιάμεσων αποτελεσμάτων. Σε εκείνη την περίπτωση απλώς σκανάρουμε τον πίνακα αυτόν.

Τέλος, στο πεδίο SELECT υπολογίζουμε το άθροισμα των στηλών που μας ζητάει η συγκεκριμένη επερώτηση (Projections).

Επιπλέον, έχουμε μια δομή που θέτει προτεραιότητες για κάθε ένα από τα κατηγορήματα (Predicate Parser). Η σειρά με την οποία θέτονται τα κατηγορήματα αναλύεται αργότερα, στο τμήμα ”Βελτιστοποιητής Επερωτήσεων”.

2.1 Ενδιάμεσα Αποτελέσματα

Κάθε φορά που εκτελούμε ένα κατηγόρημα πρέπει να κρατάμε τα αποτελέσματα του στην μνήμη, ώστε να χρησιμοποιήθουν σε μελλοντικά ερωτήματα. Για αυτόν τον σκοπό χρησιμοποιείται μία δομή για να αποθηκεύονται τα ενδιάμεσα αποτελέσματα (Intermediate Result).

3 Radix Hash Join

Όταν χρειαστεί να εκτελέσουμε ένα κατηγόρημα ζεύξης (Join) μεταξύ διαφορετικών σχέσεων που δεν ανήκουν και οι 2 στον ίδιο πίνακα ενδιάμεσων αποτελεσμάτων, χρησιμοποιούμε τον αλγόριθμο Radix Hash Join (RHJ). Τα δεδομένα που συμμετέχουν είναι ζεύγη από τιμές των στηλών (Payload) και αριθμούς γραμμής (RowIds) του πίνακα που προήλθε η τιμή. Ο αλγόριθμος χωρίζεται στα εξής 3 κομμάτια (RHJ stage1,2,3):

1. Στην αρχή περνάμε και τις 2 στήλες που συμμετέχουν στη ζεύξη απο μια συνάρτηση κατακερματισμού, έστω H1, και τις ταξινομούμε με βάση το hash value τους. Για αυτήν την διαδικασία χρησιμοποιείται ένα ιστόγραμμα (Histogram) για την εύρεση των τελικών θέσεων των στοιχείων για την ταξινόμηση. Επίσης κρατάμε και έναν πίνακα, έστω pSum, που μαζί με έναν ταξινομημένο πίνακα αποτελεί ένα hash table.

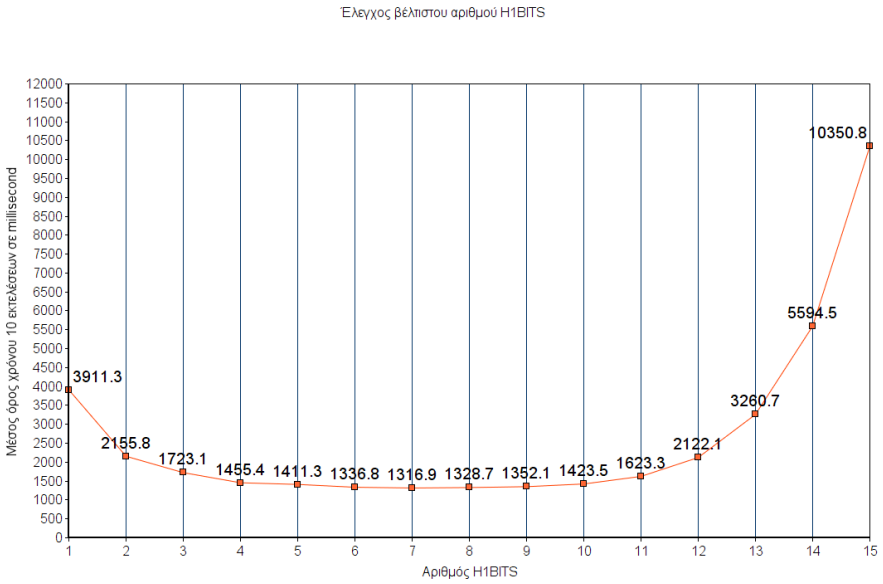
2. Έπειτα επιλέγουμε το μικρότερο από τους 2 πίνακες και δημιουργούμε ένα ευρετήριο πάνω σε αυτόν. Το ευρετήριο είναι και αυτό ένα hash table, έστω με συνάρτηση κατακερματισμού H2, και το κατασκευάζουμε χρησιμοποιόντας ένα σύνολο από πίνακες (Chains & Buckets).

3. Τέλος, διατρέχουμε ένα ένα τους κάδους του μεγαλύτερου πίνακα, δηλαδή εκείνου που δεν έχει ευρετήριο, περνάμε τις τιμές του στην H2 και χρησιμοποιώντας το ευρετήριο που κατασκευάσαμε ελέγχουμε αν τα 2 στοιχεία είναι ίσα. Όσα στοιχεία είναι ίσα τα κρατάμε σε μία δομή αποτελεσμάτων (Result).

3.1 Hashing

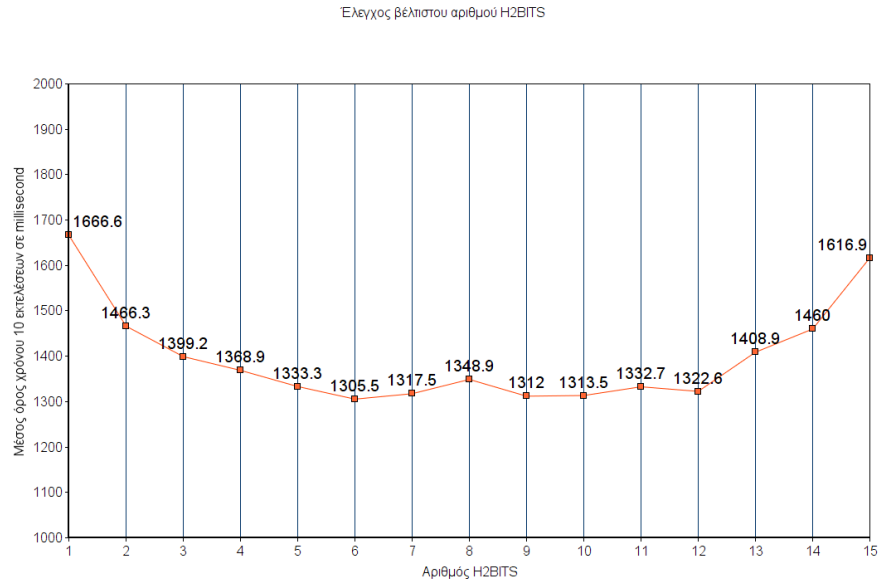
Στον αλγόριθμο Radix Hash Join χρησιμοποιούμε δύο διαφορετικές συναρτήσεις κατακερματισμού, H1 και H2 (HashFunctions). Και στις δύο συναρτήσεις είναι προκαθορισμένο (Defined) το πλήθος των ψηφίων των κλειδιών που δημιουργούν (H1BITS, H2BITS) και συνεπώς και το πλήθος των κάδων που θα δημιουργηθούν (H1BUCKETS, H2BUCKETS). Έχουμε κάνει τις εξής επιλογές για την κάθε μία:

- H1. Η πρώτη συνάρτηση κατακερματισμού κρατάει τα H1BITS λιγότερο σημαντικά ψηφία του αριθμού. Το H1BITS επιλέγεται με τρόπο έτσι ώστε ο μεγαλύτερος κάδος που θα δημιουργηθεί να χωράει στην Cache του επεξεργαστή. Ακολουθεί ένα διάγραμμα χρονικής απόδοσης εκτέλεσης για τις διάφορες τιμές του H1BITS. Οι εκτελέσεις έγιναν στο small dataset και με H2BITS = 5, Threadnum = 4, και με Join Enumeration.



Με βάση το παραπάνω επιλέξαμε το βέλτιστο H1BITS = 7.

- H2. Για τη δεύτερη συνάρτηση κατακερματισμού στηριχθήκαμε σε έρευνες που έχουν γίνει ήδη πάνω σε αποδοτικές συναρτήσεις κατακερματισμού. Μέτα από πειραματισμό βασιστήκαμε σε αυτό το blog post (https://nullprogram.com/blog/2018/07/31/) για την επιλογή της συνάρτησής μας. Για να μη δημιουργηθούν πάρα πολλοί κάδοι περιορίζουμε το αποτέλεσμα της συνάρτησης κρατώντας μόνο τα H2BITS λιγότερο σημαντικά ψηφία. Και εδώ πειραματιστήκαμε με διαφορετικό αριθμό από H2BITS. Οι εκτελέσεις έγιναν στο small dataset και με H1BITS = 7, Threadnum = 4, και με Join Enumeration.



Όπως φαίνεται πάνω, καλύτερη χρονική απόδοση είχαμε για H2BITS = 6.

4 Βελτιστοποίηση

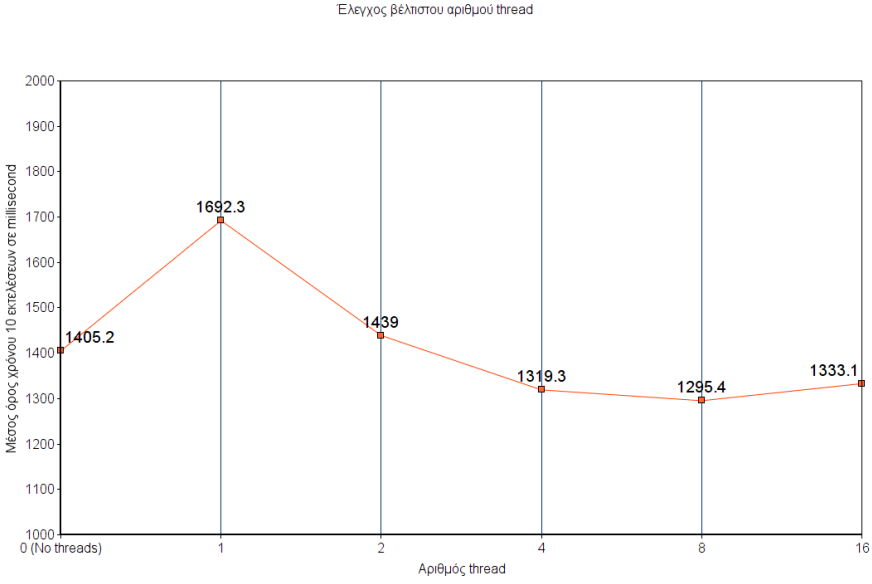
Σε αυτό το τμήμα περιγράφουμε κυρίως τις χρονικές αλλά και τις χωρικές βελτιστοποιήσεις που υλοποιήσαμε για την εφαρμογή.

4.1 Πολυνηματισμός

Πολλές από τις διαδικασίες που απαιτούνται κατά την εκτέλεση των επερωτήσεων μπορούν να χωριστούν σε αυτόνομες δουλείες (Jobs) και να εκτελεσθούν παράλληλα μέσω νημάτων. Για την διαχείριση των νημάτων και τον χρονοπρογραμματισμό των δουλειών χρησιμοποιείται ένας Job Scheduler. Κάθε δουλειά μπαίνει σε μια κυκλική ουρά και αναθέτεται στο πρώτο ελεύθερο thread που υπάρχει. Ακολουθείται η λογική FCFS (First Come, First Serve) για την ανάθεση.

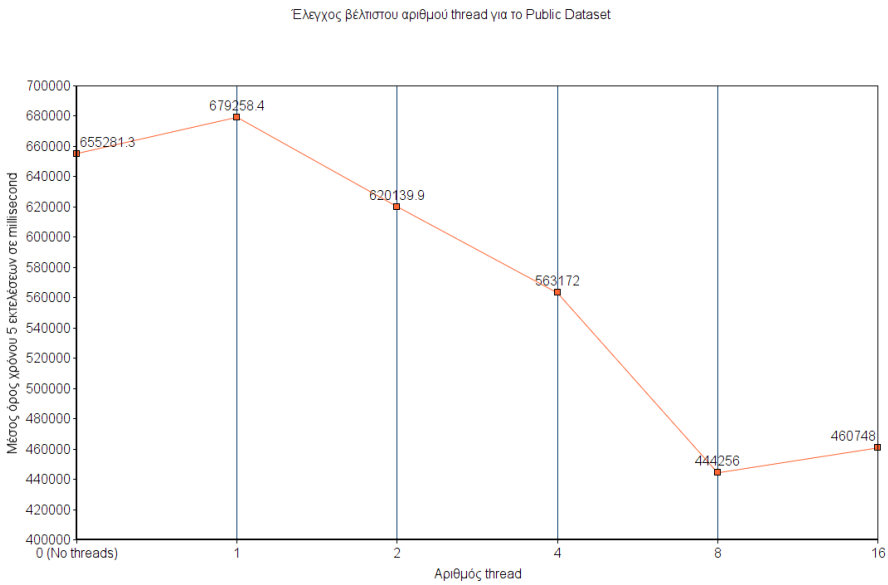
Πειραματιστήκαμε με τα σημεία όπου μπορούμε να χωρίσουμε μια διαδικασία σε διάφορα Jobs: Histogram & Partition Jobs στο πρώτο στάδιο του RHJ, InitializeBucket & InitializeIndex Jobs στο δεύτερο στάδιο, Join Job στο τρίτο στάδιο και Projection Job για την προβολή των αποτελεσμάτων των επερωτήσεων. Από αυτά τα Jobs, ελέγχοντάς τα στην πράξη, κρατήσαμε μόνο αυτά που συνείσφεραν στη βελτίωση του συνολικού χρόνου εκτέλεσης (Histogram, Partition, InitializeIndex, Join, Projection Jobs).

Επίσης πειραματιστήκαμε και με τον αριθμό των νημάτων που θα εκτελούν Jobs και τα αποτελέσματα φαίνονται παρακάτω.



Συνεπώς επιλέξαμε να χρησιμοποιούμε 8 νήματα.

Παρακάτω βλέπετε και τα αποτελέσματα του προγράμματός μας και για το public dataset που μας δίνεται:



Βλέπουμε και σε αυτό το chart (όπως και στο προηγούμενο) ότι ο βέλτιστος αριθμός νημάτων για το πρόγραμμά μας είναι 8 νήματα.

Παρατηρούμε ότι χωρίς τη χρήση νημάτων έχουμε καλύτερη απόδοση από τη χρήση ενός νήματος. Αυτό συμβαίνει γιατί αντί να εκτελείται το πρόγραμμα από την main, εκτελείται από 1 thread μόνο και έτσι οι συναρτήσεις που καλούνται για την σωστή λειτουργία του thread καθυστερούν το πρόγραμμα. Επίσης είναι πιθανό και το λειτουργικό από μόνο του να κάνει κάποια υποτυπώδη παραλληλοποίηση του προγράμματος. Ακόμα επειδή ο κάθε πυρήνας του επεξεργαστή έχει 2 thread, με 1 δεν εκμεταλλευόμαστε όλες του τις δυνατότητες.

4.2 Βελτιστοποιητής Επερωτήσεων

Δεν μπορούμε να ξέρουμε τη βέλτιστη σειρά εκτελέσης ενός συνόλου κατηγορημάτων αλλά μπορούμε να την εκτιμήσουμε κρατώντας μερικά στατιστικά για τις στήλες των πινάκων μας ώστε, μέσω ενός αλγορίθμου Join Enumeration, να την προσεγγίσουμε για να μειώσουμε το μέγεθος των ενδιάμεσων αποτελεσμάτων και εν τέλει να μειώσουμε τον συνολικό χρόνο εκτέλεσης της επερώτησης.

4.2.1 Στατιστικά

Μόλις φορτώνουμε τη ”βάση δεδομένων” μας στην μνήμη, υπολογίζουμε για κάθε στήλη κάθε πίνακα μια τετράδα στατιστικών: τη μέγιστη και ελάχιστη τιμή της στήλης, το πλήθος των γραμμών και το πλήθος των μοναδικών τιμών της. Επειδή η διαδικασία εύρεσης του πλήθους των μοναδικών τιμών μιας στήλης μπορεί να είναι χρονοβόρα, την προσεγγίζουμε χρησιμοποιώντας ένα άνω όριο διαφορετικών τιμών (N). Η συγκεκριμένη τεχνική λειτουργεί καλά μόνο για τα δεδομένα που μας δίνονται από τα αρχεία του διαγωνισμού και η ευρεία χρήση της δεν ενδείκνυται.

Με βάση τα στατιστικά αυτά, και θεωρώντας πως:

- Όλες οι τιμές είναι ομοιόμορφα κατανεμημένες, και

- Οι τιμές όλων των στηλών έχουν διαμοιραστεί ανεξάρτητα

μπορούμε να χρησιμοποιήσουμε ένα σύνολο από μαθηματικούς τύπους, ένα για κάθε είδος κατηγορήματος, ώστε να βρούμε προσεγγιστικά τα στατιστικά που προκύπτουν αν εκτελεστεί ένα οποιοδήποτε κατηγόρημα (Stats).

4.2.2 Join Enumeration

Έχοντας υπολογίσει τα στατιστικά κάθε στήλης και έχοντας μαθηματικούς τύπους για τον υπολογισμό καινούριων, μπορούμε να τα χρησιμοποίησουμε σε συνδυασμό με έναν αλγόριθμο Join Enumeration για την προσεγγιστική εύρεση της βέλτιστης σειράς εκτέλεσης των κατηγορημάτων της επερώτησης (Join Enumeration). Αρχικά θεωρούμε πως τα φίλτρα εκτελούνται πάντα πρώτα, έστω με μια τυχαία σειρά καθώς όπως διαπιστώσαμε δεν έχει μεγάλη σημασία η σειρά με την οποία εκτελούνται. Έπειτα, για το σύνολο των σχέσεων που ανήκουν σε κατηγορήματα Join, υπολογίζουμε το κόστος εκτέλεσης (ως κόστος ορίζουμε το πλήθος των ενδιάμεσων αποτελεσμάτων που εκτιμούμε πως θα δημιουργηθεί μετά την εκτέλεση του συγκεκριμενου κατηγορήματος) των συνδυασμών των διάφορων σειρών και καταλήγουμε σε μία. Για τον υπολογισμό της βέλτιστης σειράς εκτέλεσης χρησιμοποιήσαμε τον παρακάτω αλγόριθμο:

Algorithm 1 Join Enumeration

```
1:  function JoinEnumeration(Relations, Predicates)
2:      for (i = 1; i <= n; ++i) do
3:          BestTree({R_i}) = R_i;
4:      end for
5:      for (i = 1; i < n; ++i) do
6:          for all S ⊂ {R_1, ..., R_n}, |S| = i do
7:              for all R_j ∈ {R_1, ..., R_n}, R_j ∉ S do
8:                  if (NoCrossProducts !connected({R_j}, S)) then
9:                      continue;
10:                 end if
11:                 CurrTree = CreateJoinTree(BestTree(S), R_j);
12:                 S' = S ∪ {R_j};
13:                 if (BestTree(S') == NULL || cost(BestTree(S') > cost(CurrTree))
    then
14:                     BestTree(S') = CurrTree;
15:                 end if
16:             end for
17:         end for
18:     end for
19:     return BestTree({R_1, ..., R_n});
20: end function
```

Το BestTree που αναφέρεται πάνω είναι στην ουσία ένας πίνακας κατακερματισμού 2^n θέσεων για γρήγορη πρόσβαση, στις θέσεις του πίνακα, μέσω της συνάρτησης κατακερματισμού. Το κάθε κελί του BestTree αντιστοιχεί σε ένα subset του αρχικού set $\{R_1, \ldots, R_n\}$ και περιέχει την βέλτιστη μέχρι εκείνη τη στιγμή σειρά (συνδυασμού) εκτέλεσης και το κόστος του. Έτσι σαν τελικό αποτέλεσμα, στην θέση που αντιστοιχεί στο σύνολο του αρχικού set $\{R_1, \ldots, R_n\}$, μας επιστρέφει τη βέλτιστη σειρά εκτέλεσης, σύμφωνα με τα στατιστικά που έχουν υπολογιστεί. Παρότι ο συνολικός αιρθμός λογικών πράξεων, για την εύρεση των θέσεων στον πίνακα, που θα

γίνουν είναι της τάξης $\mathcal{O}(n2^n)$, το μέγεθος του n είναι μικρό και έτσι δεν επηρεάζει πρακτικά τη χρονική πολυπλοκότητα του αλγορίθμου μας.
Επίσης, όσον αφορά τον αλγόριθμο για την εύρεση όλων των subsets συγκεκριμένου μεγέθους, στη γραμμή 6 του αλγορίθμου Join Enumeration, χρησιμοποιήθηκε αλγόριθμος που έχει ως βάση τη ταυτότητα του Pascal (https://en.wikipedia.org/wiki/Pascal's_rule).

4.2.3 Αποτελέσματα Join Enumeration

Παρακάτω βλέπετε τα αποτελέσματα από τις εκτελέσεις του προγράμματός μας με και χωρίς τον αλγόριθμο του Join Enumeration και στα 2 dataset που μας δώθηκαν:

	No Join Enumeration	**With Join Enumeration**
Small Dataset	1406.2 ms	1295.4 ms
Public Dataset	470549.2 ms	444256 ms

Όπως παρατηρείτε, με τη χρήση του αλγόριθμου του Join Enumeration βλέπουμε βελτίωση του χρόνου εκτέλεσης και στα 2 dataset. Τέλος βλέπουμε ότι όσο μεγαλώνουν τα dataset, έχουμε και μεγαλύτερη βελτίωση στο χρόνο εκτέλεσης.

4.3 Μνήμη

Παρατηρήσαμε ότι όσο αυξάναμε τα bits στις hash function αυξανώταν και η κατανάλωση μνήμης. Επιλέξαμε όμως να δώσουμε προτεραιότητα στη βελτίωση του χρόνου, αφού το μέγεθος της μνήμης που κατανάλωνε το πρόγραμμα δεν ήταν απαγορευτικό. Σε περίπτωση που βλέπαμε ότι η κατανάλωση μνήμης στο βέλτιστο χρόνο ήταν μεγάλη, θα προσπαθούσαμε να φτάσουμε σε μία ισορροπία ανάμεσα σε κατανάλωση μνήμης - χρονική απόδοση του προγράμματος.

Η μνήμη που καταναλώνεται για τις δομές που μας προσφέρουν βέλτιστο χρόνο στο small dataset είναι: 3.037 Gb total heap usage.

Επίσης αρατηρήσαμε πως πολλές δομές μικρού και σταθερού μεγέθους χρησιμοποιούνται πολύ συχνά οπότε αποφασίσαμε να δεσμεύουμε χώρο για αυτές 1 φορά για όλο το πρόγραμμα. Αυτή η διαδικασία γίνεται σε μια γενική συνάρτηση αρχικοποίησης (init) μέσω global/extern μεταβλητών. Φυσικά υπάρχει και η αντίστοιχη συνάρτηση για αποδέσμευση αυτών των μεταβλητών (destroy).

Επίσης προσπαθήσαμε να εκμεταλλευτούμε την χρήση τοπικών δεικτών για αναφορά σε global μεταβλητές και την χρήση const μεταβλητών για να μειώσουμε το συνολικό κώδικα assembly που δημιουργείται, αν και αυτή η δουλειά συνήθως γίνεται αυτόματα από τους μεταγλωττιστές.

Τέλος, συναρτήσεις που χρησιμοποιούμε συχνά, όπως συναρτήσεις ελέγχου λαθών και συναρτήσεις κατακερματισμού, τις έχουμε δηλώσει inline ή τις έχουμε κάνει Define.

5 Σχόλια

Οι καλύτεροι χρόνοι για τα 2 dataset που μας δώθηκαν είναι:

Small Dataset	Public Dataset
1209 ms	420556 ms

Οι δομές με τις οποίες πετύχαμε αυτά τα αποτελέσματα αναλύονται στα παραπάνω τμήματα.

Ο έλεγχος ορθότητας των αποτελεσμάτων γίνεται μέσα από το λογισμικό που παρέχεται από το διαγωνισμό SIGMOD 2018 (Harness program) και αντίστοιχα ο έλεγχος του κώδικα και των συναρτήσεων που υλοποιήσαμε γίνεται μέσα από το framework CUnit.

Όλες οι εκτελέσεις της εφαρμογής έγιναν στο εξής μηχάνημα:

- Lenovo ThinkPad T430s - Intel Core i5 3ης Γενιάς 3320M - 8GB RAM - 240GB SSD - 3072K L3 Cache

Επίσης, για τα charts των αποτελεσμάτων που παρουσιάζουμε παραπάνω, καθώς τρέχαμε το πρόγραμμά μας για να υπολογίσουμε τις τιμές του κάθε chart κρατάγαμε τις υπόλοιπες μεταβλητές σταθερές σύμφωνα με τη βέλτιστη τιμή τους που είχαμε υπολογίσει μέχρι τότε.

Ακόμα ο υπολογιστής που τρέχαμε το πρόγραμμά μας, λόγω της μικρής μνήμης που είχε σε σχέση με το μέγεθος του public dataset, δεν μας επέτρεπε πάντα να ολοκληρώσουμε την εκτέλεση του προγράμματος, αφού οι διεργασίες ”σκοτώνονταν” από το λειτουργικό λόγω των απαιτήσεών τους σε μνήμη, κάτι το οποίο επιβεβαιώσαμε και μέσα από τα log files του τερματικού.

Τέλος, πιστεύουμε ότι μία πολύ καλή βελτιστοποίηση που θα μπορούσε να γίνει, είναι να αναθέτεται κάθε query σε ξεχωριστό thread. Γενικότερα τα επίπεδα πολυνηματισμού και βελτιστοποιήσεων που θα μπορούσαν να εφαρμοστούν στο πρόγραμμά μας είναι πολλά ακόμα και χρήζουν περαιτέρω έρευνας και πειραματισμών πάνω σε αυτόν το τομέα.

6 Επίλογος

Η υλοποίηση της εφαρμογής αυτής έχει ως κύριο σκοπό τη μελέτη του αλγορίθμου Radix Hash Join και τη βελτιστοποίηση αυτού καθώς και τη βελτιστοποίηση της ανάλυσης επερωτήσεων. Η δημιουργία της εφαρμογής σε γλώσσα C είναι σημαντικό κομμάτι γιατί είναι αποδοτική και για αυτό προτιμάται σε εφαρμογές βάσεων δεδομένων.