Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

20.01.2019

- Παναγιωτόπουλος Γεώργιος : 1115201400136
- Παπαστάμου Ιωάννης : 1115201400252

Περιεχόμενα

Πρόλογος

Η άνοδος της τεχνολογίας στον τομέα του hardware έχει οδηγήσει, τα τελευταία χρόνια, στη παραγωγή επεξεργαστών με πολλούς πυρήνες αλλά και σε μεγάλη μείωση της σχέσης τιμής ανά GB για μνήμες RAM. Η παρούσα εργασία αποτελεί μία αποδοτική υλοποίηση του τελεστή ζεύξη ισότητας στις σχεσιακές βάσεις δεδομένων, αξιοποιώντας τα σημερινά τεχνολογικά δεδομένα όσον αφορά την παραλληλία στους επεξεργαστές, αλλά και τη διαθέσιμη μνήμη RAM.

Η εργασία, υλοποιημένη σε γλώσσα C, δομήθηκε σε τρία τμήματα, τα οποία κατά τη διάρκεια της εκπόνησης της συνενώθηκαν με τέτοιο τρόπο, ώστε να πετύχουμε ένα ολοκληρωμένο αποτέλεσμα, σύμφωνα με τις οδηγίες που μας δόθηκαν.

Τέλος, θα πρέπει να τονιστεί ότι το συγκεκριμένο πρότζεκτ είναι βασισμένο στο *SIGMOD Programming Contest 2018*[1] και συνεπώς ακολουθήθηκαν οι προδιαγραφές του εν λόγω διαγωνισμού, ενώ για τις μετρήσεις χρησιμοποιήθηκαν τα *small* και *public* datasets που χρησιμοποιήθηκαν σε αυτόν.

[1] "ACM SIGMOD 2018 Programming Contest - TUM." http://sigmod18contest.db.in.tum.de/. Accessed 20 Jan. 2019.

Υλοποίηση

❖ Διάβασμα Σχέσεων

Το διάβασμα των ονομάτων των σχέσεων (relations), γίνεται κανονικά, όπως ακριβώς ζητήθηκε, με τη βοήθεια του *harness* προγράμματος, το οποίο και μας δόθηκε έτοιμο. Πιο συγκεκριμένα οι παρακάτω συναρτήσεις αποτελούν τον βασικό κορμό για το διάβασμα της εισόδου όσον αφορά τα relations :

- **void setup(struct Joiner *joiner) :** Τροφοδοτείται από το harness με το όνομα του εκάστοτε relation και με τη σειρά της καλεί την παρακάτω

- **void addRelation(struct Joiner *joiner,char *fileName) :** Αφού καλέσει την παρακάτω, προσθέτει ένα δείκτη προς το νέο relation structure στο joiner structure (Ο joiner περιέχει ένα πίνακα από δείκτες σε relations και τον συνολικό αριθμό των relations)

- **void createRelation(struct Relation **rel,char *fileName) :** Δημιουργεί ένα νέο relation και καλεί την *void loadRelation(struct Relation *rel,char *fileName) .*

Σημείωση : Οι σχέσεις (relations) φορτώνονται στη μνήμη με χρήση της *mmap*.

❖ Αποκωδικοποίηση Query

Κάθε κωδικοποιημένη επερώτηση ζεύξης (query) διαβάζεται από τη main συνάρτηση και στη συνεχεια ξεκινάει η αποκωδικοποίησή της μέσω μιας ακολουθίας κλήσεων των παρακάτω συναρτήσεων :

- **createQueryInfo(struct QueryInfo **qInfo,char *rawQuery):** Δεσμέυει χώρο για τη δομή ενός query (struct QueryInfo) και καλεί την *void parseQuery(struct QueryInfo *qInfo,char *rawQuery),* η οποία με τη σειρά της καλεί τις τρεις επόμενες :

- **void parseRelationIds(struct QueryInfo *qInfo,char *rawRelations) :** Αποκωδικοποιεί και αποθηκεύει τους αριθμούς των σχέσεων που θα συζευχθούν (πρώτο τμήμα της επερώτησης).

- **void parsePredicates(struct QueryInfo *qInfo,char *rawPredicates) :** Αποκωδικοποιεί και αποθηκεύει τα κατηγορήματα (δεύτερο τμήμα της επερώτησης).

- **void parseSelections(struct QueryInfo *qInfo,char *rawSelections)**

  Αποκωδικοποιεί και αποθηκεύει τις προβολές (τρίτο τμήμα της επερώτησης).

Σημείωση : Χρησιμοποιήσαμε παρόμοια λογική με εκείνη του κώδικα που μας δόθηκε σε στο “submission.tar.gz”

❖ Εκτέλεση Query

Εφόσον ολοκληρωθεί η αποκωδικοποίηση του query και γίνουν κάποιες εκτιμήσεις σχετικά με τα στατιστικά των στηλών των πινάκων που λαμβάνουν μέρος σε αυτό (βλ. Optimization) ξεκινάει η εκτέλεση αυτή καθαυτή με την εξής σειρά :

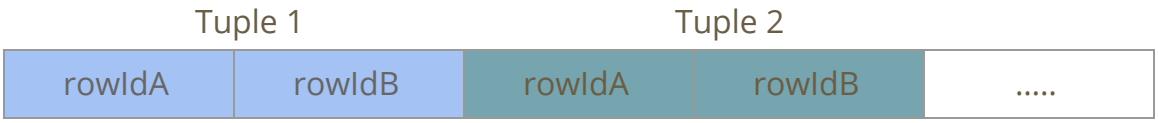
1. Εκτέλεση των column equalities (ισότητα μεταξύ στηλών του ίδιου πίνακα) (είδος φίλτρου επί της ουσίας)
2. Εκτέλεση των φίλτρων
3. Εκτέλεση των ζεύξεων

Ενδιάμεσα Αποτελέσματα :

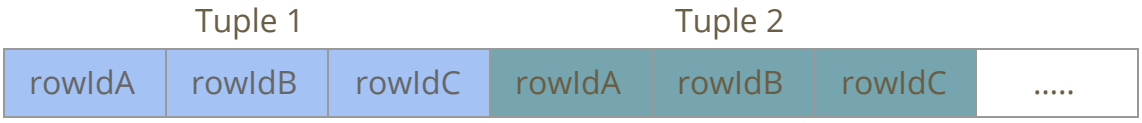
Το **struct InterMetaData** (αρχείο *Intermediate.h*) αποτελεί τη βασική δομή που κρατά τα ενδιάμεσα αποτελέσματα.

```
struct InterMetaData
{
        struct Vector  ***interResults;
        unsigned  **mapRels;
        unsigned queryRelations;
        unsigned maxNumOfVectors;
};
```

Συγκεκριμένα αποφασίσαμε για λόγους ταχύτητας (*spatial locality και caching*) να αποθηκεύουμε τα αποτελέσματα μιας ζεύξης σε **ένα vector από ακεραίους** που αποτελούν tuples με row ids των relations που συμμετείχαν στη ζεύξη, αντί για μια λίστα με buffers.



Ενδιάμεσο αποτέλεσμα μετά από ζεύξη A ⋈ B



Ενδιάμεσο αποτέλεσμα μετά από ζεύξη (A ⋈ B) ⋈ C

Στο σημείο αυτό θα πρέπει να τονιστεί ότι, μετά τη δεύτερη ζεύξη στον πίνακα με τα ενδιάμεσα αποτελέσματα, δηλαδή στον πίνακα **interResults** του **struct InterMetaData** δεν θα υπάρχουν και τα δύο vectors, παρά μόνο το δεύτερο. Ουσιαστικά, σε κάθε ζεύξη δημιουργούμε ένα νέο vector με μεγαλύτερο μέγεθος tuple (tupleSize) και διαγράφουμε το παλιό.

Με τον ίδιο τρόπο αποθηκεύουμε τα αποτελέσματα μας όταν έχουμε να εκτελέσουμε κάποιο φίλτρο ή κάποια ζεύξη μεταξύ στηλών του ίδιου πίνακα, με τη μόνη διαφορά, ότι το μέγεθος του κάθε tuple είναι 1, αφού έχουμε μία σχέση.

Ο πίνακας **mapRels** δείχνει σε ποιά θέση του κάθε tuple βρίσκεται το rowId του εκαστοτε relation. Για παράδειγμα :

mapRels :

rel0	rel1	rel2
2	0	1

Ο παραπάνω πίνακας μας δείχνει ότι σε κάθε tuple του ενδιάμεσου αποτελέσματος (δηλαδή του vector) ισχύουν τα εξής :

- Στη θέση 2 του tuple βρισκεται rowId της *relation0*
- Στη θέση 0 του tuple βρισκεται rowId της *relation1*
- Στη θέση 1 του tuple βρισκεται rowId της *relation2*

❖ Εκτέλεση Ζεύξης

Για την εκτέλεση μιας ζεύξης απαιτείται πλήθος παραμέτρων που θα πρέπει να περνούν από τη μία συνάρτηση στην επόμενη.

Γι αυτό καταφύγαμε στη λύση ενός struct , ονόματι **RadixHashJoinInfo** . Το struct αυτό περιέχει όλες τις απαραίτητες πληροφορίες όπως, ο αριθμός της σχέσης, ο αριθμός της στήλης, το ενδιάμεσο αποτέλεσμα-vector στο οποίο υπάρχει ήδη αυτή η σχέση κ.α , τα οποία χρειάζεται κάθε συνάρτηση που λαμβάνει μέρος στην εκτέλεση της ζεύξης.

Πιο συγκεκριμένα η λογική πορεία που ακολουθείται είναι η εξής :

- **void initializeInfo(struct InterMetaData *inter,struct QueryInfo *q,struct SelectInfo *s,struct Joiner *j,RadixHashJoinInfo *arg)**

  Δημιουργία το εν λόγω struct πληροφορίας για κάθε μία εκ των δύο σχέσεων

- **void applyProperJoin(struct InterMetaData *inter,RadixHashJoinInfo* argLeft,RadixHashJoinInfo* argRight)**

  Επίλυση του είδους της ζεύξης

  - joinNonInterNonInter
  - joinNonInterInter
  - joinInterNonInter
  - joinInterInter

- **void partition(RadixHashJoinInfo *info)**
  - Εύρεση ιστογράμματος (είτε με χρήση threads, είτε σειριακά)
  - Εύρεση αθροιστικού ιστογράμματος
  - Δημιουργία *ταξινομημένης-ανά-κάδο* στήλης (είτε με χρήση threads, είτε σειριακά)

- **void build(RadixHashJoinInfo *infoLeft,RadixHashJoinInfo *infoRight)**

  Δημιουργία ευρετηρίου πάνω στη **μικρότερη** εκ των δύο σχέσεων με τη μορφή *bucket chaining*. Στην τελική έκδοση της εργασία μας αυτό γίνεται παράλληλα με 1 job για κάθε κάθε κάδο (βλ. Πολυνηματισμός)

- **void joinFunc(void *arg)**

  Διαδικασία εύρεσης αποτελεσμάτων της ζεύξης βάσει του ευρετηρίου που φτιάξαμε προηγουμένως. Η συγκεκριμένη συνάρτηση “δουλεύει” πάνω σε δύο κάδους, έναν από τη μία σχέση και έναν από την άλλη, και εκτελείται από ένα thread (βλ. Πολυνηματισμός)

Έτσι, θα έχουμε δημιουργήσει ένα νέο ενδιάμεσο αποτέλεσμα το οποίο το “καρφιτσώνουμε” στον **interResults** πίνακά μας, “ξεκαρφιτσώνοντας”, δηλαδή διαγράφοντας, το προηγούμενο vector της κάθε σχέσης(εφόσον αυτή ήταν ήδη σε κάποιο ενδιάμεσο αποτέλεσμα). Αυτό το βήμα

πραγματοποιείται στις συναρτήσεις joinNonInterNonInter, joinNonInterInter κλπ, που βρίσκονται στο αρχείο Operations.c

❖ Εκτέλεση Φίλτρων

Η εκτέλεση των φίλτρων γίνεται με πολύ πιο απλό τρόπο, σαρώνοντας τη στήλη και εισάγοντας στο ενδιάμεσο αποτέλεσμα-vector τα rowids που ικανοποιούν το φίλτρο. Παρομοίως λειτουργεί και η ζεύξη μεταξύ στηλών του ίδιου πίνακα.

❖ Παρατηρήσεις

- Ο παράμετροι RADIX_BITS και HASH_RANGE_1 παίρνουν τιμές δυναμικά ανάλογα με τα μεγέθη των δοθέντων relations (Joiner.c :: setRadixBits).

- Το ίδιο ισχύει και για το αρχικό μέγεθος του vector μας, προκειμένου να αποφεύγεται η κλήση της *realloc()* , αλλά και να ελαχιστοποιείται η σπατάλη μνήμης.

- Αξίζει να σημειωθεί, ότι έχουμε λάβει υπόψην μας τη περίπτωση του να δοθούν σε ένα query πολλαπλά φίλτρα πάνω στην ίδια σχέση ή και ακόμη πάνω στην ίδια στήλη, αλλά και πολλαπλές ζεύξεις ισότητας μεταξύ δύο στηλών της ίδιας σχεσης.

Πολυνηματισμός

Όσον αφορά το κομμάτι του multithreading προσπαθήσουμε να αποκτήσουμε όσο περισσότερο χρονικό όφελος μπορούσαμε, εφαρμόζοντας παραλληλία σε πολλά σημεία της υλοποίησής μας.

Πιο συγκεκριμένα εφαρμόσαμε multithreading στα εξής :

- Εκτέλεση φίλτρων

  (Operations.c :: void filterFunc(void *arg))

- Εκτέλεση ζεύξης μεταξύ στηλών του ίδιου ενδιάμεσου πίνακα

(Vector.c :: void colEqualityFunc(void *arg))

- Υπολογισμός ιστογράμματος
  (Partition.c :: void histFunc(void* arg))

- Δημιουργία ταξινομημένης-ανά-κάδο σχέσης
  (Partition.c :: void partitionFunc(void* arg))

- Δημιουργία ευρετηρίου για κάθε κάδο
  (Build.c :: void buildFunc(void* arg))

- Εκτέλεση της ζεύξης ανά κάδο
  (Probe.c :: void joinFunc(void *arg))

- Εύρεση αθροίσματος
  (Vector.c :: void checkSumFunc(void *arg))

Παρατηρήσεις

- Η ουρά με τα jobs υλοποιήθηκε με χρήση πίνακα για καλύτερο locality και ταχύτητα, παρά σα συνδεδεμένη λίστα με κόμβους.

- Τα jobs δημιουργούνται εξ αρχής στην *void createJobArrays(struct JobScheduler* js)*, πριν αρχίσουμε να διαβάζουμε τα queries, ώστε να γλυτώσουμε το overhead των συνεχόμενων malloc/free.

- Ο αριθμός των threads δίνεται από τη #define'd σταθερά THREAD_NUM στο αρχείο JobScheduler.c

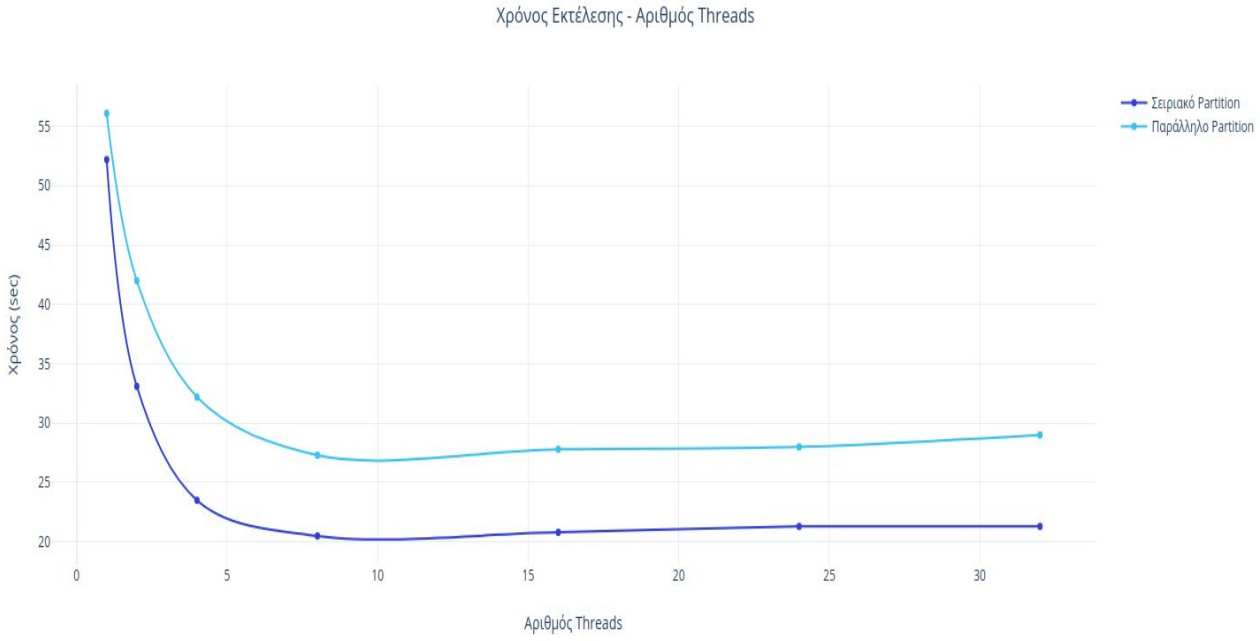
- Στο κομμάτι της partitionFunc αναγκαστήκαμε να χρησιμοποιήσουμε mutexes προκειμένου να αποφύγουμε τα race conditions. Πιο συγκεκριμένα καταφύγαμε στη λύση του *one-mutex-per-bucket* .

Βελτιστοποίηση ερωτημάτων

Όσον αφορά τη βελτιστοποίηση των ερωτημάτων, παρότι συλλέγουμε τα στατιστικά σύμφωνα με τους τύπους που μας δόθηκαν και κάνουμε τις ανάλογες εκτιμήσεις πληθικότητας, δεν καταφέραμε να υλοποιήσουμε τον αλγόριθμο δυναμικού προγραμματισμού που μας δόθηκε.

Πειραματιστήκαμε με πιο απλές υλοποιήσεις για βελτιστοποίηση και *costFunction* = *fA* * *fB* ,όπου *fA* πλήθος δεδομένων της μιας στήλης και *fB* πλήθος δεδομένων της άλλης, χωρίς όμως να πάρουμε τα αποτελέσματα που θέλαμε από άποψη χρόνου.

Μετρήσεις

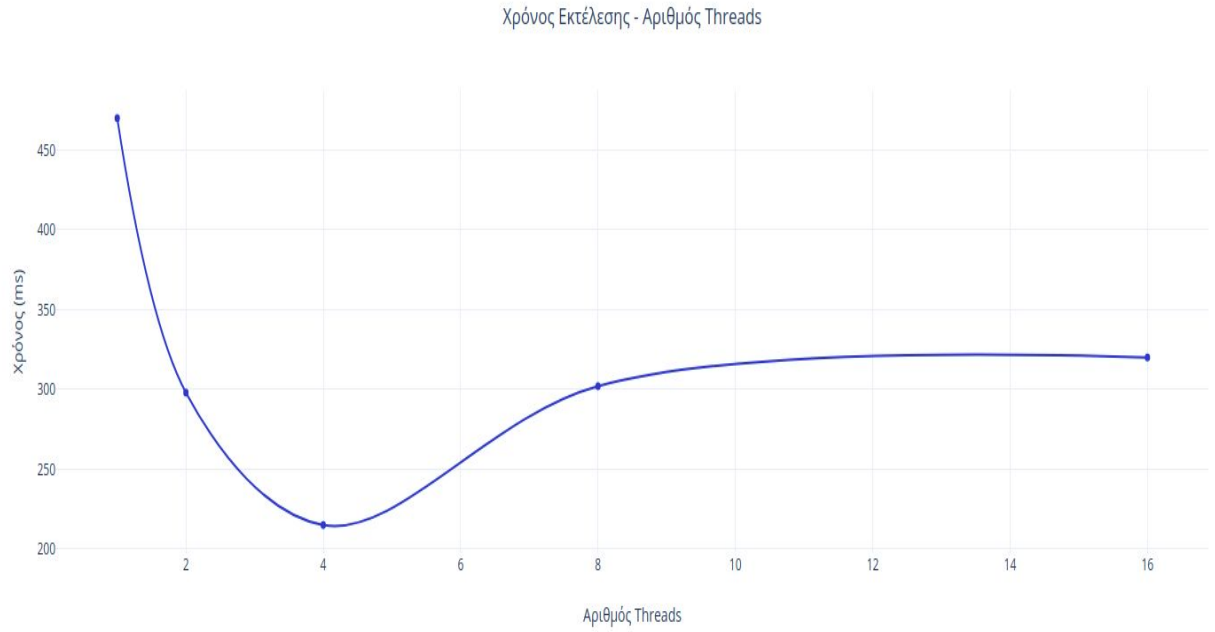


Η παραπάνω μετρήσεις αφορούν το **Public** dataset και έγιναν σε *Ryzen 2400G στα 3.6 GHz 4 cores-8 threads , dual-channel 16GB RAM*.

Σχόλια :

- Όπως είναι λογικό ο καλύτερος χρόνος εμφανίζεται για αριθμό threads ίσο με 8.
- Το παράλληλο partition είναι πιο άργο λόγω της χρήσης mutex που δημιουργεί μεγάλη συμφόρηση (congestion)

Σειριακό partition στο μηχάνημα



της σχολής :