Τμήμα Πληροφορικής & Τηλεπικοινωνιών

K23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Τρίτο Μέρος

Χειμερινό Εξάμηνο 2019 – 2020

Καθηγητής Ι. Ιωαννίδης

Άσκηση 3 – Παράδοση: Κυριακή 19 Ιανουαρίου 2020

Στο τρίτο μέρος της εργασίας θα ασχοληθούμε με τη βελτιστοποίηση του κώδικα που έχετε δημιουργήσει στα προηγούμενα μέρη της άσκησης. Αυτό θα επιτευχθεί τόσο με τη χρήση παραλληλίας όσο και με τη δημιουργία ενός βελτιστοποιητή ερωτημάτων (query optimizer).

Πολυνηματισμός

Ο πολυνηματισμός αποτελεί ένα ευρέως διαδεδομένο μοντέλο προγραμματισμού και εκτέλεσης διεργασιών, το οποίο επιτρέπει την ύπαρξη πολλών νημάτων μέσα στα πλαίσια μιας και μόνο διεργασίας. Τα νήματα αυτά μοιράζονται τους πόρους της διεργασίας και μπορούν να εκτελούνται ανεξάρτητα. Επειδή όλα τα νήματα διαχειρίζονται το ίδιο τμήμα μνήμης, θα πρέπει να δοθεί προσοχή στις συνθήκες ταυτόχρονης προσπέλασης και τροποποίησης της διαμοιραζόμενης μνήμης.

Posix Threads

Στα πλαίσια της εργασίας θα χρησιμοποιήσετε POSIX Threads. Οι βασικές ρουτίνες των POSIX Threads είναι:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void*(*start_routine) (void),
                   void *arg)
```

Ο στόχος της ρουτίνας pthread_create είναι να δημιουργήσει ένα νέο νήμα. Αφού αρχικοποιηθούν τα χαρακτηριστικά του, γίνεται διαθέσιμο για χρήση. Στη μεταβλητή thread επιστρέφεται το αναγνωριστικό του νήματος που μόλις δημιουργήθηκε. Με βάση την τιμή στο πεδίο attr θα αρχικοποιηθούν τα χαρακτηριστικά του νέου νήματος. Στο όρισμα start_routine ορίζεται η ρουτίνα που θα εκτελέσει το νέο thread όταν δημιουργηθεί και στο πεδίο arg θα οριστούν οι παράμετροι αυτής.

void pthread_exit(void *value_ptr)

Η ρουτίνα pthread_exit, θα τερματίσει ένα ήδη υπάρχον νήμα και θα αποθηκεύσει την κατάσταση τερματισμού για χρήση από άλλα νήματα που θα προσπαθήσουν να συνενωθούν με αυτό. Επιπρόσθετα, απελευθερώνει όλα τα δεδομένα του νήματος, συμπεριλαμβανομένης και της στοίβας του νήματος. Είναι σημαντικό τα αντικείμενα συγχρονισμού νημάτων, όπως τα mutexes και οι μεταβλητές κατάστασης (condition variables) που αποθηκεύονται στη στοίβα του νήματος, να καταστραφούν πριν κλήση της ρουτίνας pthread_exit.

int pthread_join(pthread_t thread, void **value_ptr)

Η ρουτίνα pthread_join μπλοκάρει το τρέχον νήμα μέχρι να τερματίσει το νήμα που προσδιορίζεται από το πεδίο thread. Η κατάσταση τερματισμού του νήματος αυτού επιστρέφεται στο πεδίο value_ptr. Αν το συγκεκριμένο νήμα έχει ήδη τερματίσει (και δεν είχε προηγουμένως αποσπαστεί) το τρέχον νήμα δε θα μπλοκαριστεί.

Mutexes

Όταν πολλαπλά νήματα χρησιμοποιούν τις ίδιες δομές, θα πρέπει να βρεθεί ένας τρόπος συγχρονισμού της πρόσβασης σε αυτές, ώστε να εξασφαλιστεί η απαραίτητη συνέπεια στο αποτέλεσμα των ενεργειών που εκτελούν τα νήματα. Η POSIX παρέχει τους mutexes. Οι mutexes έχουν δύο καταστάσεις locked, unlocked. Όσο κάποιο νήμα, έχει κλειδωμένο ένα mutex τότε αν κάποιο άλλο νήμα προσπαθήσει να κλειδώσει τον ίδιο mutex, το δεύτερο νήμα θα “κολλήσει” μέχρι να ξεκλειδωθεί. Άρα αν η πρόσβαση στις κοινές δομές γίνεται ανάμεσα σε κλείδωμα και ξεκλείδωμα του αντίστοιχου mutex, αποφεύγουμε τις ταυτόχρονες αλλαγές και τα λάθη που μπορεί να προκαλέσουν. Χρησιμοποιώντας ένα mutex ανά κοινή δομή, μπορούμε να περιορίσουμε την πρόσβαση σε αυτή σε ένα μόνο νήμα κάθε στιγμή. Οι βασικές ρουτίνες των POSIX mutexes είναι οι εξής:

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)

Αρχικοποιεί έναν mutex, με attributes που καθορίζονται από την μεταβλητή attr. Εάν η μεταβλητή attr είναι NULL, τότε χρησιμοποιούνται τα προκαθορισμένα attributes. Κατα την επιτυχή αρχικοποίηση ενός mutex, η κατάσταση του mutex είναι unlocked.

int pthread_mutex_lock(pthread_mutex_t *mutex)

Αν ο mutex είναι ξεκλείδωτος, τότε η κατάστασή του γίνεται locked. Αν ο mutex είναι κλειδωμένος, τότε το νήμα που τον κάλεσε παγώνει μέχρι ο mutex να γίνει διαθέσιμος.

int pthread_mutex_unlock(pthread_mutex_t *mutex)

Κάνει την κατάσταση του mutex unlocked και ξεπαγώνει κάποιο από τα νήματα που περιμένουν λόγω του συγκεκριμένου mutex.

int pthread_mutex_destroy(pthread_mutex_t *mutex)

Καταστρέφει ένα mutex.

Condition Variables

Ένα ακόμα εργαλείο συγχρονισμού που μπορεί να φανεί χρήσιμο είναι τα condition variables. Ένα condition variable είναι ένα εργαλείο που επιτρέπει στα POSIX threads να αναβάλουν την εκτέλεση τους μέχρι μια έκφραση να γίνει αληθής. Δύο είναι οι βασικές πράξεις πάνω σε ένα condition variable: wait() που αδρανοποιεί το νήμα που την κάλεσε και signal() που ξυπνά ένα από τα νήματα που είναι απενεργοποιημένα πάνω στο ίδιο condition variable. Ένα condition variable χρησιμοποιείται μαζί με ένα mutex. Οι βασικές ρουτίνες των POSIX condition variable είναι οι εξής:

int pthread_cond_init(pthread cond_t *cond, pthread_condattr_t *cond attr)

Η συνάρτηση αρχικοποιεί μια condition variable, αναφερόμενη ως cond, με attributes που αναφέρονται στη μεταβλητή attr. Αν η attr είναι NULL, τότε χρησιμοποιούνται τα προκαθορισμένα attributes.

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)

Η συνάρτηση αυτή παγώνει το νήμα μέχρι κάποιος το ξυπνήσει. Η συνάρτηση αυτή παίρνει σαν όρισμα έναν mutex, τον οποίο και αλλάζει σε κατάσταση unlocked. Για να ξυπνήσει το νήμα, θα πρέπει ο mutex να είναι σε κατάσταση unlock. Αν δεν είναι, τότε περιμένει μέχρι να γίνει.

int pthread_cond_signal(pthread_cond_t *cond)

Η συνάρτηση αυτή ξυπνάει κάποιο από τα νήματα που έχουν παγώσει στο συγκεκριμένο condition variable.

int pthread_cond_broadcast(pthread_cond_t *cond)

Η συνάρτηση αυτή ξυπνάει όλα τα νήματα που έχουν παγώσει στο συγκεκριμένο condition variable.

int pthread_cond_destroy(pthread_cond_t *cond)

Καταστρέφει ένα condition variable.

Στον παρακάτω κώδικα παρουσιάζεται ένα παράδειγμα χρήσης condition variables, όπου ένα νήμα-πατέρας δημιουργεί ένα νήμα-παιδί. Ο πατέρας μέσω της χρήσης μιας condition variable περιμένει έως ότου το παιδί τελειώσει την εργασια του.

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

Semaphores

Οι POSIX semaphores επιτρέπουν σε διεργασίες και νήματα να συγχρονίζουν τις ενέργειες τους. Ένας σημαφόρος, είναι ένας μη αρνητικός ακέραιος. Ακολουθούν οι βασικές ρουτίνες των POSIX Unnamed Semaphores:

int sem_init (sem_t *sem, int pshared, unsigned int value)

Αρχικοποιεί τον σημαφόρο sem με την τιμή value. Η μεταβλητή pshared δηλώνει πότε ο σημαφόρος χρησιμοποιείται μεταξύ νημάτων η διεργασιών. Αν η τιμή του είναι μηδέν, σημαίνει ότι χρησιμοποιείται από νήματα μιας διεργασίας. Αν η τιμή του είναι διαφορετική από μηδέν σημαίνει ότι χρησιμοποιείται μεταξύ διεργασιών.

int sem_destroy (sem_t *sem)

Καταστρέφει τον σημαφόρο sem.

Int sem_post (sem_t *sem)

Αυξάνει την τιμή του σημαφόρου κατά ένα.

Int sem_wait (sem_t *sem)

Μειώνει την τιμή του σημαφόρου κατά ένα. Αν η τιμή είναι μεγαλύτερη του μηδενός τότε το νήμα συνεχίζει κανονικά με την επεξεργασία. Αν είναι ίση με μηδέν τότε το νήμα παγώνει μέχρι η τιμή του σημαφόρου γίνει και πάλι μεγαλύτερη του μηδενός.

Int sem_getvalue (sem_t *sem, int *sval)

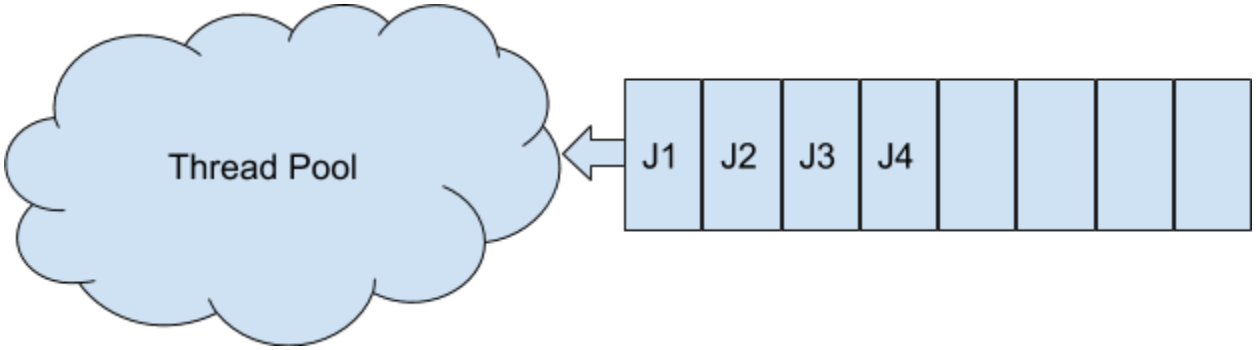
Επιστρέφεται στη μεταβλητή sval η μεταβλητή του σημαφόρου.

Job Scheduler

Η χρήση των νημάτων μπορεί να γίνει με δύο τρόπους είτε δημιουργώντας καινούργια νήματα για κάθε παράλληλο κομμάτι, είτε με την υλοποίηση ενός job scheduler. Η δεύτερη επιλογή, παρότι πιο σύνθετη σε επίπεδο υλοποίησης, είναι συνήθως προτιμότερη ειδικά σε εφαρμογές, όπου η παράλληλη επεξεργασία γίνεται ασύγχρονα κατά τη διάρκεια εκτέλεσης του προγράμματος και φυσικά αποτρέπει την πολλαπλή δημιουργία των δομών των νημάτων. Για τους παραπάνω λόγους θα υλοποιηθεί ένα job scheduler για την παράλληλη επεξεργασία.

Η βασική οντότητα του χρονοπρογραμματιστή είναι εκείνη της εργασίας (Job). Ενα Job είναι μια ρουτίνα κώδικα, η οποία θέλουμε να εκτελεστεί από κάποιο νήμα, πιθανότατα παράλληλα με κάποια άλλη. Οποιαδήποτε εργασία μπορεί να οριστεί ως job και να ανατεθεί στον χρονοπρογραμματιστή. Ο χρονοπρογραμματιστής ουσιαστικά δέχεται δουλειές και αναλαμβάνει την ανάθεση τους σε νήματα. Για προσωρινή αποθήκευση των εργασιών χρησιμοποιεί μια ουρά αναμονής. Έστω ότι έχουμε μια δεξαμενή νημάτων (thread pool) και μια συνεχόμενη ροή από ανεξάρτητες εργασίες (jobs). Όταν δημιουργείται μια εργασία, μπαίνει στην

ουρά αναμονής του χρονοπρογραμματιστή και περιμένει να εκτελεστεί. Οι εργασίες εκτελούνται με την σειρά που εισήχθησαν στην ουρά (first-in-first-out - FIFO). Κάθε νήμα, αν δεν έχει κάποια εργασία να εκτελέσει, παίρνει ένα Job από την ουρά. Αφού την εκτελέσει, επιστρέφει στην ουρά για να αναλάβει νέα εργασία.



Πιο κάτω δίνονται κάποια ενδεικτικά πρότυπα συναρτήσεων για την δημιουργία του Job Scheduler. Δεν είναι δεσμευτικό να χρησιμοποιήσετε αυτά τα πρότυπα:

```
// Abstract Class Job
class Job {
public:
  Job() = default;
  virtual ~Job() {}

  // This method should be implemented by subclasses.
  virtual int Run() = 0;
};

// Class JobScheduler
class JobScheduler {
 public:
  JobScheduler() = default;
  ~JobScheduler() = default;

  // Initializes the JobScheduler with the number of open threads.
  // Returns true if everything done right false else.
  bool Init(int num_of_threads);

  // Free all resources that are allocated by JobScheduler
  // Returns true if everything done right false else.
  bool Destroy();

  // Waits Until executed all jobs in the queue.
```

```
  void Barrier();

  // Add a job in the queue and returns a JobId
  JobID Schedule(Job* job);


  // Waits until all threads finish their job, and after that close all threads.
  void Stop();
};
```

Παραλληλοποίηση

Η παραλληλοποίηση του προγράμματός σας, θα γίνει σε πολλά επίπεδα.

Αρχικά η παραλληλοποίηση θα γίνεται σε επίπεδο επερώτησης. Κάθε επερώτηση θα πρέπει να εκτελείται σε διαφορετικό job. Αυτού του είδους η παραλληλία είναι σχετικά απλή, αφού απαιτείται ταυτόχρονη προσπέλαση σε μνήμη μόνο, χωρίς να χρειάζεται τροποποίηση διαμοιραζόμενων δομών, άρα και μικρές απαιτήσεις συγχρονισμού.

Στη συνέχεια η παραλληλοποίηση θα εκτελεστεί σε επίπεδο τελεστή ζεύξης. Συγκεκριμένα, ο αλγόριθμος ζεύξης που κατασκευάσαμε στο πρώτο μέρος μπορεί να παραληλοποιηθεί σε δύο επίπεδα. Το ένα επίπεδο αφορά την παραλληλοποίηση του μέρους του αλγορίθμου που ασχολείται με την ταξινόμηση καθενός από τους περιοχές με ίδιο πρόθεμα που δημιουργούνται κατά την πρώτη σάρωση, και το άλλο στο επίπεδο που ο αλγόριθμος κάνει ζεύξη στις αντίστοιχες περιοχές με το ίδιο πρόθεμα.

Για το πρώτο μέρος της παραλληλοποίησης, οι αρχικές σχέσεις R, S θα πρέπει να διαχωριστούν κατά πρόθεμα σε διαφορετικούς κάδους. Στη συνέχεια, θα δημιουργηθεί μία εργασία για την επιμέρους ταξινόμηση των διαφορετικών περιοχών (SortJob), ώστε στο τέλος να έχουμε δύο πλήρως ταξινομημένες σχέσεις. Η επιμέρους εργασία θα δουλεύει όπως έχει περιγραφεί και στο πρώτο μέρος για την ταξινόμηση.

Στην τελευταία φάση θέλουμε να εκτελέσουμε παράλληλα τη ζεύξη ανάμεσα στους πίνακες. Ένας απλός τρόπος να το πετύχουμε αυτό είναι να δημιουργήσουμε παράλληλα jobs (JoinJobs). Κάθε τέτοιο job θα είναι υπεύθυνο ώστε να εκτελέσει το Join ανάμεσα σε δύο περιοχές με ίδιο πρόθεμα, μία από τον R’ και μία από το S’. Τα αποτελέσματα του κάθε JoinJob θα γραφονται σε ξεχωριστή δομή Result. Για τον λόγο αυτό, προτείνεται να μειωθεί σε 128 KB, το συνολικό μέγεθος του κάθε κόμβου τις λίστας με τα αποτελέσματα.

Εκτός από τον τελεστή ζεύξης, μπορούμε να επιταχύνουμε το πρόγραμμά χρησιμοποιώντας παραλληλοποίηση σε πολλά ακόμα άλλα σημεία (φίλτρα, αθροίσματα κτλ). Όποιος θέλει μπορεί να ασχοληθεί με την παραλληλοποίηση και άλλων σημείων της άσκησης. Η ενασχόληση αυτή θα προσμετρηθεί ώς bonus.

Βελτιστοποιητής επερωτήσεων

Ένα ερώτημα μπορεί να εκτελεστεί με πολλούς διαφορετικούς τρόπους. Έστω για παράδειγμα ότι έχουμε το πιο κάτω ερώτημα sql:

```
Select *
From A, B, C, D
Where A.a = B.a and
      B.a = C.a and
      C.a = D.a;
```

Ένας τρόπος να εκτελέσουμε τις ζέυξεις του ερωτήματος είναι ((A⋈B)⋈C)⋈D, που σημαίνει ότι πρώτα θα εκτελέσουμε τα κατηγορήματα με την εξής σειρά:

A.a = B.a
B.a = C.a
C.a = D.a

Ένας άλλος τρόπος να εκτελέσουμε το ίδιο ερώτημα είναι ((C⋈B)⋈A)⋈D, που σημαίνει ότι θα εκτελέσουμε τα κατηγορήματα με την εξής σειρά:

B.a = C.a
A.a = B.a
C.a = D.a

Η σειρά με την οποία θα εκτελέσουμε της ζεύξης έχει πολύ μεγάλη σημασία στον χρόνο εκτέλεσης ενός ερωτήματος. Αυτό συμβαίνει γιατί το πλήθος των ενδιάμεσων αποτελεσμάτων μπορεί να διαφέρει πολύ μεταξύ διαφορετικών συνδυασμών εκτέλεσης. Σε αυτό το μέρος θα δούμε τους βασικούς τρόπους για την αποφυγή συνδυασμών που μπορεί να είναι χρονοβόροι.

Στατιστικά

Δυστυχώς, δεν μπορούμε να γνωρίζουμε εξαρχής ποιο θα είναι το πλήθος των αποτελεσμάτων ενός τελεστή αν πρώτα δεν εκτελέσουμε των τελεστή. Το μόνο που μπορούμε να κάνουμε, είναι να εκτιμήσουμε το πλήθος των αποτελεσμάτων, με βάση κάποια στατιστικά που έχουμε κρατήσει, για κάθε στήλη που συμμετέχει στον τελεστή. Στα πλαίσια της άσκησης θα κρατάμε τις εξής τέσσερις τιμές για κάθε στήλη:

$\{l_A, u_A, f_A, d_A\}$

Το A αναφέρεται στο όνομα μιας στήλης κάποιας σχέσης R. Η τιμη l_A, αναφέρεται στην μικρότερη τιμή της στήλης A. Η τιμή u_A αναφέρεται στην μεγαλύτερη τιμή της στήλης A. Η τιμή f_A, αναφέρεται στο πλήθος των δεδομένων της στήλης A, και τέλος η τιμή d_A, αναφέρεται στο πλήθος των μοναδικών τιμών της στήλης A.

Τα στατιστικά, θα υπολογίζονται κατα την διάρκεια που οι πίνακες φορτώνονται στην μνήμη. Ο χρόνος που θα πρέπει να σπαταληθεί γι αυτά θα πρέπει να είναι λιγότερος από ένα δευτερόλεπτο. Ο υπολογισμός των τιμών l_A,u_A,f_A, μπορεί να γίνει πολύ εύκολα, με ένα απλό περασμα της εκάστοτε στήλης. Οι εύρεση των distinct values, από την άλλη είναι μια πολύ χρονοβόρα διαδικασία ακόμα και αν γίνει με την χρήση ενός hashtable. Παρόλα αυτά δεν χρειάζεται να είναι ακριβείς, αλλά κοντα στις πραγματικές τιμές. Ένας τρόπος για να μετρήσουμε τις μοναδικές τιμές, είναι να δημιουργήσουμε έναν boolean πίνακα u_A - l_A + 1 θέσεων, αρχικοποιημένο σε false. Μετά από την δημιουργία αυτού του πίνακα, για κάθε τιμή (x) της στήλης, μπορούμε να κάνουμε true την θέση του πίνακα x - u_A.

Η συγκεκριμένη τεχνική, ενέχει τους κινδύνους της δέσμευσης ενός πολύ μεγάλου τμήματος μνήμης για τον boolean πίνακά μας. Για να αποφύγουμε τέτοιες καταστασεις, αν η τιμή u_A - l_A + 1 είναι μεγαλύτερη από ένα προκαθορισμένο αριθμό N, τότε θα δεσμευσουμε ένα boolean πίνακα με τόσα κελιά όσα και μας λεει η τιμή N. Αν τώρα η τρέχουσα τιμή είναι η τιμή x, θα κάνουμε true το κελί x - l_A % N. Μια τιμή μικρότερη του 50.000.000, θεωρείται μια καλή τιμή για τα συγκεκριμένα dataset. Να σημειωθεί πώς η τεχνική αυτή δουλεύει μόνο για τα δεδομενα που δόθηκαν, και δεν πρέπει να θεωρηθεί ότι είναι μια γενικώς αποδεκτή λύση για την εύρεση μοναδικών τιμών.

Εκτίμηση Πληθικότητας

Τα στατιστικά που κρατήσαμε προηγουμένως, θα τα χρησιμοποιήσουμε για την εύρεση του πλήθους των αποτελεσμάτων, των βασικών τελεστών μας. Για να το πετύχουμε αυτό παίρνουμε τις πιο κάτω βασικές παραδοχές:

1. Όλες οι τιμές είναι ομοιόμορφα κατανεμημένες, και
2. Οι τιμές όλων των στηλών έχουν διαμοιραστεί ανεξάρτητα

Έχοντας αυτες τις δύο υποθέσεις στο μυαλό μας, μπορούμε να κάνουμε εκτιμήσεις πληθικότητας για τους βασικούς τελεστές μας.

Φίλτρα της μορφής $\sigma_{A=k}$

Αν έχουμε φίλτρα ισότητας (πχ R.A = k), τότε τα στατιστικά της στήλης που προκύπτει μετα την εφαρμογη του φίλτρου είναι:

$$l'_A = k$$
$$u'_A = k$$
$$d'_A = 1 \text{αν } k \text{ ανήκει στο } d_A,\ 0 \text{ αλλιώς}$$

$f'_A = f_A/d_A$ αν k ανήκει στο d_A, 0 αλλιώς

Τα στατιστικά οποιασδήποτε άλλης στήλης (C), του πίνακα που εφαρμόστηκε το φίλτρο είναι:

$$l'_C = l_C$$
$$u'_C = u_C$$
$$d'_C = d_C * (1 - (1 - \frac{f'_A}{f_A})^{f_C/d_C})$$
$$f'_C = f'_A$$

Φίλτρα της μορφής $\sigma_{k1 <= A <= k2}$

Αν έχουμε φίλτρα ευρους (R.A >= k1 and R.A <= k2) τότε τα στατιστικά για τη στήλη που προκύπτει μετα την εφαρμογη του φίλτρου είναι:

$$l'_A = k_1$$
$$u'_A = k_2$$
$$d'_A = \frac{k_2 - k_1}{u_A - l_A} * d_A$$
$$f'_A = \frac{k_2 - k_1}{u_A - l_A} * f_A$$

Για να ισχύουν οι πιό πάνω τύποι έχουμε κάνει τις κατάλληλες μετατροπές ώστε τα k_1, k_2 να είναι εντός ορίων. Συγκεκριμένα:

Αν $k_1 < l_A$ τότε $k_1 = l_A$.
Αν $k_2 > u_A$ τότε $k_2 = u_A$.

Τα στατιστικά οποιασδήποτε άλλης στήλης (C) του πίνακα που εφαρμόστηκε το φίλτρο είναι:

$$l'_C = l_C$$
$$u'_C = u_C$$
$$d'_C = d_C * (1 - (1 - \frac{f'_A}{f_A})^{f_C/d_C})$$
$$f'_C = f'_A$$

Δεν είναι δύσκολο να εφαρμόσουμε τους πιο πάνω τύπους, για φίλτρα που έχουν μόνο μικρότερο η μεγαλύτερο. Στην πρώτη περίπτωση θα ορίσουμε το κάτω όριο l_A, και στη δεύτερη το άνω όριο u_A.

Φίλτρα της μορφής $\sigma_{A=B}$

Αν έχουμε φίλτρο μεταξύ δύο στηλών του ίδιου πίνακα (R.A = R.B) τότε τα στατιστικά για τις στήλες που προκύπτουν μετά την εφαρμογή του φίλτρου είναι:

$$l'_A = l'_B = max(l_A, l_B)$$
$$u'_A = u'_B = min(u_A, u_B)$$
$$f'_A = f'_B = \frac{f}{n}$$
$$d'_A = d'_B = d_A * (1 - (1 - \frac{f'_A}{f_A})^{f_A/d_A})$$

, όπου $n = u'_A - l'_A + 1$

Για οποιαδηποτε άλλη στήλη του πίνακα, ισχύει:

$$l'_C = l_C$$
$$u'_C = u_C$$
$$d'_C = d_C * (1 - (1 - \frac{f'_A}{f_A})^{f_C/d_C})$$
$$f'_C = f'_A$$

Ζεύξη μεταξύ δύο διαφορετικών πινάκων

Αν έχουμε μια ζεύξη, μεταξύ δύο στηλών διαφορετικών πινάκων, τότε θα πρέπει πρώτα να δημιουργήσουμε τις κατάλληλες εκτιμήσεις, εφαρμόζοντας τα κατάλληλα φίλτρα, έτσι ώστε οι l και οι u τιμές του κάθε πίνακα, να είναι ίδιες. Για παράδειγμα, αν έχουμε για τη στήλη A.a την τετράδα (0, 100, 10000, 10), και για τη στήλη B.a την τετράδα (20, 110, 1000, 10), τότε θα πρέπει να βγάλουμε δύο νέες τετράδες για κάθε μια από τις 2 στήλες, με $l_{A.a} = l_{B.a} = 20$ και $u_{A.a} = u_{B.a} = 100$. Για να το πετύχουμε αυτο, θα πρέπει να πάρουμε τις κατάλληλες εκτιμήσεις πληθικότητας εφαρμόζοντας τα κατάλληλα φίλτρα.

Εφόσον κάνουμε το παραπάνω, είμαστε έτοιμοι να εκτιμήσουμε το αποτέλεσμα του τελεστή ζεύξης. Τα στατιστικά για τις στήλες που συμμετέχουν στον τελεστή ζεύξης είναι:

$$l'_A = l'_B = l_A = l_B$$
$$u'_A = u'_B = u_A = u_B$$
$$f'_A = f'_B = \frac{f_A * f_b}{n}$$
$$d'_A = d'_B = \frac{d_A * d_B}{n}$$

, όπου n = $u_{A,B} - l_{A,B} + 1$.

Για οποιαδήποτε άλλη στήλη των δύο πινάκων ισχύει ότι:

$$l'_C = l_C$$

$$u'_C = u_C$$
$$f'_C = f'_A$$
$$d'_C = d_C * (1 - (1 - \frac{d'_A}{d_A})^{f_C/d_C}) \text{ αν } C \text{ είναι } attribute \text{ που ανήκει στο A}$$
$$d'_C = d_C * (1 - (1 - \frac{d'_B}{d_B})^{f_C/d_C}) \text{ αν } C \text{ είναι } attribute \text{ που ανήκει στο B}$$

Αυτοσυσχέτιση

Για τις ζεύξεις που πραγματοποιούνται στην ίδια στήλη του ίδιου πίνακα, ισχύει ότι:

$$l'_A = l_A$$
$$u'_A = u_A$$
$$f'_A = \frac{f_A f_A}{n}$$
$$d'_A = d_A$$

Για όλες τις άλλες στήλες ισχύει ότι:

$$l'_C = l_C$$
$$u'_C = u_C$$
$$f'_C = f'A$$
$$d'_C = d_C$$

Join Enumeration

Εφόσον μπορούμε να πάρουμε προβλέψεις πληθικότητας, για καθένα από τους τελεστές που χρησιμοποιούμε, μπορούμε να βρούμε το συνολικό κόστος ενός συνδυασμού εκτέλεσης. Στα πλαίσια της εργασίας προσπαθούμε να ελαχιστοποιήσουμε το πλήθος των ενδιάμεσων αποτελεσμάτων. Αυτή θα είναι και η συνάρτηση κόστους μας. Δηλαδή για δύο συνδυασμούς που παράγουν το ίδιο αποτέλεσμα θα θεωρείται καλύτερος εκείνος που παράγει τα λιγότερα ενδιάμεσα αποτελέσματα.

Θεωρούμε πως τα φίλτρα εκτελούνται πάντα πρώτα. Η σειρά που θα εκτελεστούν τα φίλτρα δεν έχει ιδιαίτερη σημασία. Αυτό που έχει σημασία είναι η σειρά που εκτελούνται οι ζεύξεις. Αν στο ερώτημά μας έχουμε N σχέσεις, τότε όλοι οι δυνατοί συνδυασμοί που μπορούν να προκύψουν είναι N!. Γρήγορα ένας τέτοιος αλγόριθμος φτάνει στα όρια του.

Παρόλα αυτά, μπορούμε να παρατηρήσουμε ότι αν έχουμε τέσσερις σχέσεις A, B, C, D και η σειρά εκτέλεσης (A⋈B)⋈C είναι η βέλτιστη για τις σχέσεις A, B, C, τότε δεν χρειάζεται να ελέγξουμε αν ο συνδυασμός ((C⋈A)⋈B)⋈D είναι καλύτερος από τον ((A⋈B)⋈C)⋈D, αφού ο δεύτερος είναι σίγουρα καλύτερος. Σε αυτή την παρατήρηση βασίζεται ένα εύρος αλγορίθμων δυναμικού προγραμματισμού για Join Enumeration. Πιο κάτω δίνεται ο ψευδοκώδικας ενός τέτοιου αλγορίθμου.

```
for (i = 1; i <= n; ++i) {
  BestTree({R_i}) = R_i;
}
for (i = 1; i < n; ++i) {
  for all S ⊆ {R_1, ..., R_n}, |S| = i do {
    for all R_j ∈ {R_1, ..., R_n}, R_j ∉ S do {
      if (NoCrossProducts && !connected({R_j}, S)) {
          continue;
      }
      CurrTree = CreateJoinTree(BestTree(S), R_j);
      S' = S ∪ {R_j};
      if (BestTree(S') == NULL || cost(BestTree(S')) > cost(CurrTree)) {
          BestTree(S') = CurrTree;
      }
    }
  }
}
return BestTree({R_1, ..., R_n});
```

Ο αλγόριθμος θεωρεί ότι έχει ένα hashTable με όνομα BestTree που κρατάει για κάθε υποσύνολο σχέσεων τον καλύτερο συνδυασμό. Στο παράδειγμά μας, για το υποσύνολο σχέσεων {A, B, C} θα κράταγε τον συνδυασμό (A⋈B)⋈C. Αρχικά ο αλγόριθμος, αρχικοποιεί το BestTree με όλους τους συνδυασμούς που χρησιμοποιούν μόνο μία σχέση. Έπειτα παράγει όλους τους συνδυασμούς που χρησιμοποιούν 2 σχέσεις κοκ. Στο τέλος βρίσκει τον καλύτερο συνδυασμό που χρησιμοποιεί όλες τις σχεσεις. Προσέξτε ότι ο συγκεκριμένος αλγόριθμος δεν υπολογίζει συνδυασμούς που χρειάζεται καρτεσιανό γινόμενο για να εκτελεστούν.

Τελική Αναφορά

Στη τελική αναφορά θα παρουσιάσετε μια σύνοψη ολόκληρη της εφαρμογής που υλοποιήθηκε. Μπορείτε να αναφέρετε πράγματα που παρατηρήσατε κατά την μοντελοποίηση/υλοποίηση της εφαρμογής σας, με αποτέλεσμα να σας οδηγήσουν σε συγκεκριμένες σχεδιαστικές επιλογές που βελτίωσαν την εφαρμογή σας σε επίπεδο χρόνου, μνήμης, κτλ.
Στην αναφορά πρέπει ακόμη να παρουσιάσετε ένα σύνολο από πειράματα, τα οποία θα δείχνουν το χρόνο εκτέλεσης για όλες τις επιλογές των δομών που αναπτύξατε στα τρία επίπεδα για τα δοθέντα datasets. Για παράδειγμα μπορείτε να αναφέρετε (π.χ. διαγράμματα) το χρόνο εκτέλεσης, κατανάλωση μνήμης, πολυνηματισμό κ.α. . Τέλος, είναι απαραίτητο να παρουσιάσετε τις δομές που σας έδωσαν τους καλύτερους χρόνους εκτέλεσης για τα datasets και τον τελικό χρόνο/μνήμη, μαζί με τις προδιαγραφές του μηχανήματος που τρέξατε τα πειράματα. Η αναφορά δεν πρέπει να ξεπερνά τις 30 σελίδες.

Προθεσμία παράδοσης: 19/01/2020.

Χρήσιμοι Σύνδεσμοι: http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

Περιβάλλον υλοποίησης: Linux (gcc 5.4+).

Παραδοτέα: Η παράδοση της εργασίας θα γίνει με βάση το τελευταιο commit πριν την προθεσμία υποβολής στο git repository σας. Η τελικη αναφορά σε μορφή pdf θα βρίσκεται μέσα στο repository. **Η χρήση git είναι υποχρεωτική**.