Ανάπτυξη Λογισμικού Πληροφορικής
Χειμερινό Εξάμηνο 2016-2017

“Εύρεση συντομότερων μονοπατιών σε γράφο”

Υπεύθυνος Καθηγητής	Συνεργάτες Μαθήματος
Ιωάννης Ιωαννίδης	Μαίλης Θεόφιλος
	Γιαννακόπουλος Αναστάσιος
	Γαβάνας Χρήστος
	Γαλούνη Κωνσταντίνα
	Λιβισιανός Τσαμπίκος
	Μιχαηλίδης Θοδωρής

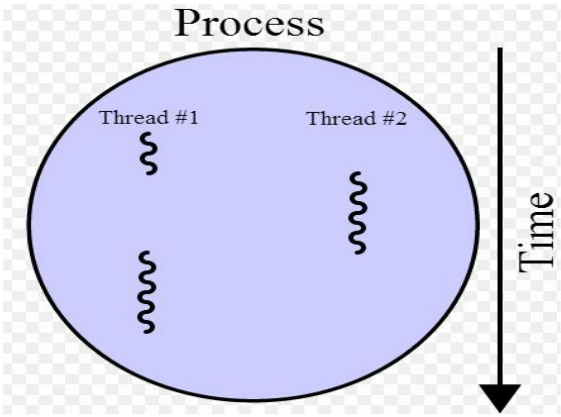
Γ’ Μέρος της Άσκησης

1. Περιγραφή Λειτουργικότητας και Δομών

Στο 3ο επίπεδο της εργασίας, θα εισαχθεί παραλληλία με στόχο τη μείωση του συνολικού χρόνου εκτέλεσης. Η παραλληλία θα εφαρμοστεί μέσω πολυνηματισμού κατά την επεξεργασία των ερωτημάτων εύρεσης συντομότερων μονοπατιών ανά ριπή. Στην επόμενη ενότητα θα γίνει μια σύντομη αναφορά στις βασικές τεχνικές πολυνηματισμού.

2. Πολυνηματισμός

Ένα νήμα είναι μια ελαφριά διεργασία. Η υλοποίηση των νημάτων και των διεργασιών διαφέρει από το ένα λειτουργικό σύστημα στο άλλο. Σε κάθε διεργασία υπάρχει τουλάχιστον ένα νήμα. Αν μέσα σε μια διεργασία υπάρχουν πολλαπλά νήματα, τότε αυτά διαμοιράζονται την ίδια μνήμη και πόρους αρχείων.



Σχήμα 1. Παράδειγμα εκτέλεσης πολυνηματικής διεργασίας

Ένα νήμα διαφέρει από μια διεργασία ενός πολυεπεξεργαστικού λειτουργικού συστήματος στα εξής:

- οι διεργασίες είναι τυπικώς ανεξάρτητες, ενώ τα νήματα αποτελούν υποσύνολα μιας διεργασίας.
- οι διεργασίες περιέχουν σημαντικά περισσότερες πληροφορίες κατάστασης από τα νήματα, ενώ πολλαπλά νήματα μιας διεργασίας μοιράζονται την κατάσταση της διεργασίας, όπως επίσης μνήμη και άλλους πόρους.
- οι διεργασίες έχουν ξεχωριστούς χώρους διευθυνσιοδότησης, ενώ τα νήματα μοιράζονται το σύνολο του χώρου διευθύνσεων που τους παραχωρείται
- η εναλλαγή ανάμεσα στα νήματα μιας διεργασίας είναι πολύ γρηγορότερη από την εναλλαγή ανάμεσα σε διαφορετικές διεργασίες.

Η πολυνημάτωση 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. Το γεγονός ότι επιτρέπει μια διεργασία να εκτελείται παράλληλα σε ένα σύστημα με πολλαπλούς πυρήνες αποτελεί ίσως την πιο ενδιαφέρουσα εφαρμογή της συγκεκριμένης τεχνολογίας.

Το πλεονέκτημα ενός πολυνηματικού προγράμματος είναι ότι του επιτρέπει να εκτελείται γρηγορότερα σε υπολογιστικά συστήματα που έχουν πολλούς επεξεργαστές, επεξεργαστές με πολλούς πυρήνες, ή κατά μήκος μιας συστοιχίας υπολογιστών. Για να μπορούν να χειραγωγηθούν σωστά τα δεδομένα, τα νήματα θα πρέπει ορισμένες φορές να συγκεντρωθούν σε ένα ορισμένο χρονικό διάστημα έτσι ώστε να επεξεργασθούν τα δεδομένα στη σωστή σειρά.

Ένα ακόμα πλεονέκτημα της πολυδιεργασίας (και κατ' επέκταση της πολυνημάτωσης) ακόμα και στους απλούς επεξεργαστές (με έναν πυρήνα επεξεργασίας) είναι η δυνατότητα για μια εφαρμογή να ανταποκρίνεται άμεσα. Έχοντας ένα πρόγραμμα που εκτελείται μόνο σε ένα νήμα, αυτό θα φαίνεται ότι κολλάει ή ότι παγώνει, στις περιπτώσεις που εκτελείται κάποια μεγάλη διεργασία που απαιτεί πολύ χρόνο. Αντίθετα σε ένα πολυνηματικό σύστημα, οι χρονοβόρες διεργασίες μπορούν να εκτελούνται παράλληλα με άλλα νήματα που φέρουν άλλες εντολές για το ίδιο πρόγραμμα, καθιστώντας το άμεσα ανταποκρίσιμο.

Για την υλοποίηση των πολυνηματικών λειτουργιών θα χρησιμοποιήσετε τη Pthreads που είναι μια POSIX API C βιβλιοθήκη. Για τη χρήση της πρέπει να συμπεριλαμβάνεται στα αρχεία του κώδικα σας το <pthread.h> και να χρησιμοποιήσετε κατά το compilation το " -pthread flag " .

Οι βασικές ρουτίνες των PThreads είναι οι εξής:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
  void*(*start_routine) (void), void *arg)
- void pthread_exit(void *value_ptr)
- int pthread_join(pthread_t thread, void **value_ptr)

Ο στόχος της ρουτίνας pthread_create είναι να δημιουργήσει ένα νέο νήμα και αφού αρχικοποιήσει τα χαρακτηριστικά αυτού, το κάνει διαθέσιμο για χρήση. Στην μεταβλητή thread επιστρέφετε 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 ορισθούν οι παράμετροι αυτής.

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

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

Συγχρονισμός

Όταν δυο thread χρησιμοποιούν τις ίδιες δομές θα πρέπει να βρείτε ένα τρόπο να συγχρονίσετε την πρόσβαση σε αυτές, ώστε να εξασφαλίσετε συνέπεια στο αποτέλεσμα των ενεργειών που εκτελούν τα threads. Η POSIX παρέχει τους mutexes.
Οι mutexes έχουν δυο καταστάσεις locked, unlocked. Χρησιμοποιώντας ένα mutex ανά κοινή δομή, μπορούμε να περιορίσουμε την πρόσβαση σε αυτή σε ένα μόνο thread κάθε στιγμή.

Οι βασικές ρουτίνες των POSIX mutexes είναι οι εξής:

- int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *mutexattr)
- int pthread_mutex_lock(pthread_mutex_t *mutex)
- int pthread_mutex_unlock(pthread_mutex_t *mutex)
- int pthread_mutex_destroy(pthread_mutex_t *mutex)

Όσο κάποιο thread έχει κλειδωμένο ένα mutex τότε αν κάποιο άλλο thread προσπαθήσει να κλειδώσει τον ίδιο mutex, το δεύτερο thread 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.

Υλοποίηση

Η χρήση των threads μπορεί να γίνει με δύο τρόπους είτε δημιουργώντας καινούργια threads για κάθε παράλληλο κομμάτι, είτε με την υλοποίηση ενός job scheduler. Αν και η δεύτερη επιλογή είναι 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 υλοποιήσετε ένα job scheduler για την παράλληλη επεξεργασία.

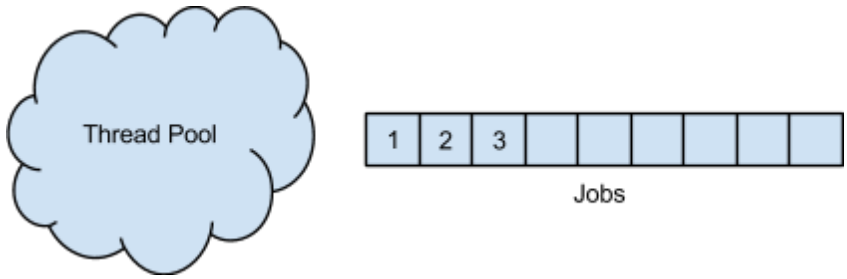
Job Scheduler

Εργασία (Jobs)

Εργασία (Job) είναι μια ρουτίνα κώδικα η οποία θέλουμε να εκτελεστεί από κάποιο thread πιθανότατα παράλληλα με κάποια άλλη. Μπορούμε να ορίσουμε οτιδήποτε θέλουμε ως job και να το αναθέσουμε στον χρονοπρογραμματιστή.

Χρονοπρογραμματιστής Εργασιών (Scheduler) και Δεξαμενή Νημάτων (Thread Pool)

Ο χρονοπρογραμματιστής ουσιαστικά δέχεται δουλειές και αναλαμβάνει την ανάθεση τους σε νήματα, για προσωρινή αποθήκευση των εργασιών χρησιμοποιεί μια ουρά προτεραιότητας. Έστω ότι έχουμε μια Δεξαμενή νημάτων (thread pool) και μια συνεχόμενη ροή από ανεξάρτητες εργασίες (jobs). Όταν δημιουργείται μια εργασία, μπαίνει στην ουρά προτεραιότητας του χρονοπρογραμματιστή και περιμένει να εκτελεστεί. Οι εργασίες εκτελούνται με την σειρά που δημιουργήθηκαν (first-in-first-out - FIFO). Κάθε νήμα περιμένει στην ουρά μέχρι να του ανατεθεί μια εργασία και, αφού την εκτελέσει, επιστρέφει στην ουρά για να αναλάβει νέα εργασία. Για την ορθή λειτουργία ενός χρονοπρογραμματιστή είναι απαραίτητη η χρήση σεμαφόρων στην ουρά, ώστε να μπλοκάρουν εκεί τα νήματα, και κάποια κρίσιμη περιοχή (critical section), ώστε να γίνεται σωστά εισαγωγή και απολαβή εργασιών από την ουρά.



Σχήμα 2. Αναπαράσταση δομής JobScheduler

Condition Variables

Ένα ακόμα εργαλείο συγχρονισμού που μπορεί να σας 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 thread που την κάλεσε και η signal που ξυπνά ένα από τα thread που είναι απενεργοποιημένα πάνω στο ίδιο condition variable. Ένα condition variable χρησιμοποιείται ζευγάρι 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)
- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
- int pthread_cond_signal(pthread_cond_t *cond)
- int pthread_cond_broadcast(pthread_cond_t *cond)
- int pthread_cond_destroy(pthread_cond_t *cond)

Οι condition variables χρησιμοποιούνται μέσα σε while loops:

```
mutex_lock(mtx1)
while( canRun ){
   cond_var_wait(condvar1, mtx1)
}
//do some work
cond_var_signal(condvar1)
mutex_unlock(mtx1)
```

Πριν τη χρήση του condvar1 κλειδώνουμε το mtx1. Μετά ελέγχουμε σε μια while μια συνθήκη που αν αποτιμηθεί ως αληθής σημαίνει ότι το thread δεν μπορεί να κάνει τη δουλειά του και πρέπει να αδρανοποιηθεί με τη χρήση της cond_var_wait. Αλλιώς το thread δεν εισέρχεται στο εσωτερικό της while και εκτελεί τη δουλειά του. Χρησιμοποιεί τη signal για να ξυπνήσει ένα ακόμα thread, αν υπάρχει, που είναι αδρανοποιημένο στο ίδιο condition variable και ξεκλειδώνει το mtx1.

Ενσωμάτωση δομής Job Scheduler

Στην αρχή του προγράμματος θα δημιουργείται η δομή του JobScheduler, η οποία αποθηκεύει όλα τα jobs , διαχειρίζεται την εκτέλεσή τους θα διατηρείται καθόλη τη διάρκεια εκτέλεσης του προγράμματος και θα καταστρέφεται στο τέλος.

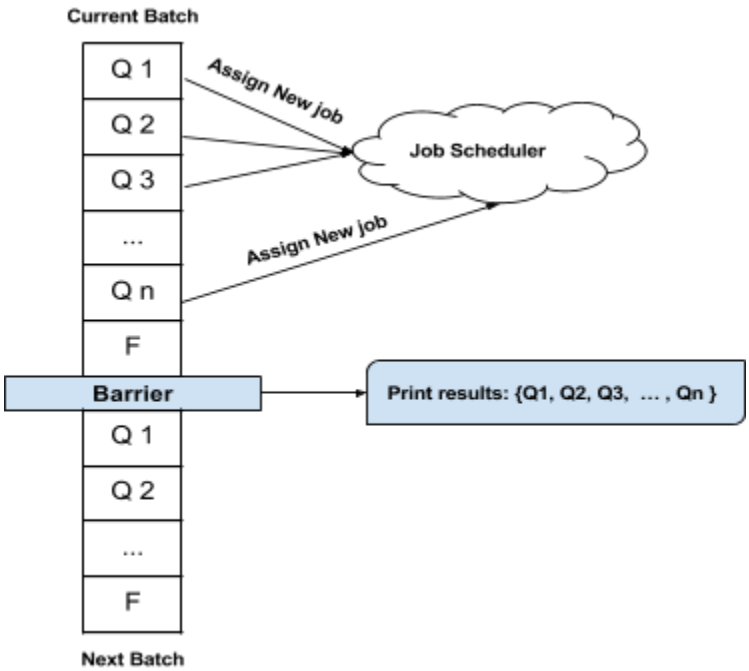
```
struct JobScheduler{
        int execution_threads; // number of execution threads
        Queue* q; // a queue that holds submitted jobs / tasks
        p_thread_t* tids; // execution threads
        ….
         // mutex, condition variable, ...
};
```

Εκτός της δομής του scheduler απαιτείται και μια δομή Job, η οποία καθορίζει ουσιαστικά το κώδικα της εργασίας που εκτελείται παράλληλα. Η δομή αυτή πρέπει να μπορεί να υποστηρίζει διαφορετικούς τύπους εργασιών, τις οποίες εκτελεί ο scheduler παράλληλα, χωρίς να γνωρίζει τη λειτουργικότητά τους.
Οι βασικές λειτουργίες που πρέπει να υλοποιεί η δομή του scheduler είναι οι εξής:

```
JobScheduler* initialize_scheduler( int execution_threads);

void submit_job( JobScheduler* sch, Job* j);

void execute_all_jobs( JobScheduler* sch);

void wait_all_tasks_finish( JobScheduler* sch); //waits all submitted tasks to finish

OK_SUCCESS destroy_scheduler( JobScheduler* sch);
```

Χρήση JobScheduler σε στατικούς γράφους

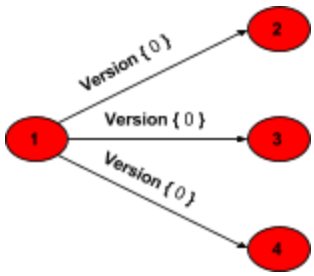
Τα στατικά workloads, δεν περιλαμβάνουν μεταβολές στο γράφο με αποτέλεσμα ο παράλληλος υπολογισμός των ερωτημάτων να είναι απλός στην εφαρμογή του. Πιο συγκεκριμένα, μόλις ξεκινάει κάθε ριπή, θα τη διαβάζετε ολόκληρη σειριακά μέχρι την επομένη εντολή “F” και θα δημιουργείτε jobs τα οποία θα εισάγονται στην ουρά του scheduler. Μόλις ολοκληρωθεί η δημιουργία των εργασιών τότε, πρέπει ο job scheduler να εκκινήσει την παράλληλη εκτέλεση των εργασιών και το κυρίως πρόγραμμα να περιμένει να τελειώσουν όλες οι εργασίες προτού συνεχίσει. Στη συνέχεια, το κυρίως πρόγραμμα θα τυπώσει όλα τα αποτελέσματα της τρέχουσας ριπής και θα συνεχίσει με την επόμενη.
Ένα σημείο που απαιτεί προσοχή στην παραπάνω διαδικασία είναι η εκτύπωση των αποτελεσμάτων με τη σωστή σειρά, ώστε να γίνεται σωστά ο έλεγχος των αποτελεσμάτων. Για να επιτευχθεί το παραπάνω κατά την ανάθεση εργασιών, κάθε εργασία θα παίρνει ένα αύξων αναγνωριστικό το οποίο δηλώνει πρακτικά τη σειρά του εκάστοτε ερωτήματος. Το αναγνωριστικό αυτό, θα το χρησιμοποιήσει κάθε εργασία μόλις υπολογίσει την απάντηση του ερωτήματος για να γράψει το αποτέλεσμα στην κατάλληλη θέση. Με αυτό το τρόπο το κυρίως πρόγραμμα θα αναγνώσει μετά όλα τα αποτελέσματα με τη σειρά που ανατέθηκαν οι εργασίες και θα τα τυπώσει στην πρότυπη έξοδο. Το παρακάτω σχήμα δείχνει τη ροή εκτέλεσης των ερωτημάτων στους στατικούς γράφους.



Σχήμα 3. Περιγραφή παράλληλης εκτέλεσης ερωτημάτων σε στατικό workload

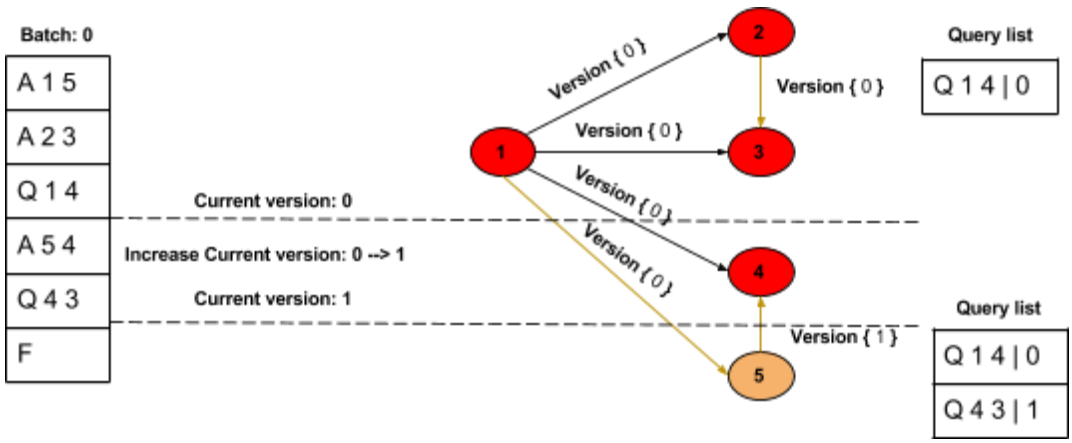
Χρήση JobScheduler σε δυναμικούς γράφους

Στα δυναμικά workloads η προσθήκη ακμών στο γραφό, επηρεάζει την απάντηση των ερωτημάτων, ανάλογα με τη σειρά που εκτελούνται οι εντολές “A” και “Q”. Επομένως, είναι απαραίτητο να διασφαλιστεί ότι κατά την παράλληλη επεξεργασία το κάθε ερώτημα θα “βλέπει” την κατάσταση του γράφου, όπως διαμορφώνεται αυτή κατά την ανάγνωση του workload. Με βάση τα παραπάνω, κατά την εισαγωγή ακμών στο γράφο, κάθε ακμή θα κρατάει στο πεδίο “property” της, μια ακέραιη τιμή το “versioning” που αναπαριστά ανάλογα με τη τιμή της, την κατάσταση του γράφου μια δεδομένη χρονική στιγμή.
Κατά τη διάρκεια της εκτέλεσης το πρόγραμμα διατηρεί μια μεταβλητή “current_version” , η οποία αρχικοποιείται με “0” . Επομένως, κατά το φόρτωμα του αρχικού γράφου κάθε ακμή θα έχει ως ιδιότητα τη τιμή “0” . Παρακάτω παρουσιάζεται ένας γράφος στην αρχική του κατάσταση.



Σχήμα 4. Αρχική κατάσταση γράφου

Στη συνέχεια ακολουθεί η ανάγνωση της 1ης ριπής από το workload. Στόχος είναι να γίνει η προετοιμασία και η δημιουργία των jobs της ριπής τα οποία θα εκτελεστούν παράλληλα. Αρχικά, δημιουργούμε μια λίστες μια για τις εντολές ερωτημάτων. Αν η τρέχουσα εντολή είναι “Q” , τότε την εισάγουμε στη λίστα των ερωτημάτων και αποθηκεύουμε επιπλέον στην εντολή τη τρέχουσα τιμή της μεταβλητής “current_version”. Σε περίπτωση που η τρέχουσα εντολή είναι “A”, τότε γίνεται η προσθήκη της νέας ακμής στο γράφο. Η νέα ακμή θα έχει ως ιδιότητα ακμής τη τιμή της μεταβλητής “current_version”, αν η προηγούμενη εντολή ήταν και αυτή τύπου “A”. Ωστόσο, αν η προηγούμενη εντολή ήταν τύπου “Q”, τότε αυξάνεται η μεταβλητή “current_version” κατά ένα και προσαρτάται η ενημερωμένη τιμή ως ιδιότητα στη νέα ακμή.” Ένα παράδειγμα της παραπάνω διαδικασίας παρουσιάζεται στο επόμενο σχήμα.



Σχήμα 5. Παράδειγμα δημιουργίας tasks σε δυναμικό workload

Σε αυτό το σημείο πρέπει για όλα τα ερωτήματα να δημιουργηθούν jobs και να ανατεθούν στον job scheduler. Επιπλέον, όπως στους στατικούς γράφους έτσι και στους δυναμικούς, πρέπει να καθοριστεί η σωστή σειρά που θα γραφούν τα αποτελέσματα με τον ίδιο τρόπο.

Υπολογισμός ερωτημάτων συντομότερου μονοπατιού με versioning

Δεδομένου ότι κατά τη διάρκεια υπολογισμών σε μια ριπή, πρώτα πραγματοποιούνται όλες οι εισαγωγές σειριακά και έπειτα εκτελούνται παράλληλα τα ερωτήματα, κατά τη διάρκεια αποτίμησης ενός ερωτήματος είναι πιθανό ο γράφος να έχει παραπάνω ακμές από αυτές που μετέχουν πραγματικά στο ερώτημα. Ο καθορισμός των ακμών που πραγματικά αφορούν το ερώτημα ή όχι πραγματοποιείται με βάση τη τιμή “version” του ερωτήματος σε συνδυασμό με τις αντίστοιχες τιμές “version” στις ακμές του γράφου. Επομένως, κατά την αναζήτηση για παράδειγμα μιας BFS διάσχισης σε ερώτημα ***query*** πρέπει για κάθε ακμή ***edge*** που αναπτύσσεται να ισχύει ότι: ***edge.version ≤ query.version.*** Τέλος, η τιμή του version πρέπει να λαμβάνεται υπόψιν και κατά την αναζήτηση με χρήση των ευρετηρίων connected components και cc index. Στους στατικούς γράφους η ύπαρξη ευρετηρίων δεν επηρέαζει τη διαδικασία αναζήτησης.

3. Τελική αναφορά εργασίας

Στη τελική αναφορά θα παρουσιάσετε μια σύνοψη ολόκληρη της εφαρμογής που υλοποιήσατε. 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 με ή χωρίς ευρετήρια, πολυνηματισμό κ.α. . Τέλος, είναι απαραίτητο να παρουσιάσετε τις δομές που σας έδωσαν τους καλύτερους χρόνους εκτέλεσης για τα δύο datasets και το τελικό χρόνο/μνήμη, μαζί με τις προδιαγραφές του μηχανήματος που τρέξατε τα πειράματα. Η αναφορά δεν πρέπει να ξεπερνά τις 3 σελίδες.