Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2022– 2023

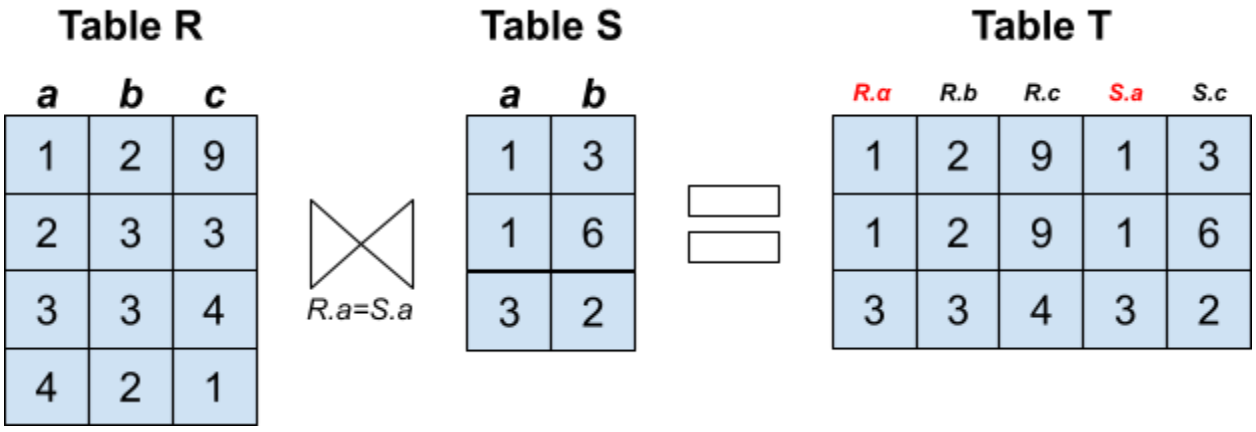
Καθηγητής Ι. Ιωαννίδης

Άσκηση 1 – Παράδοση: Δευτέρα 7 Νοεμβρίου 2022

Με την εξέλιξη των υπολογιστών τα τελευταία χρόνια, η σχέση τιμής ανά GB για μνήμες RAM είναι πολύ μικρή ενώ ταυτόχρονα αυξάνονται ολοένα και περισσότερο ο αριθμός των πυρήνων σε κάθε CPU. Σήμερα είναι σύνηθες να συναντάμε Servers με TB RAM και 20 επεξεργαστικούς πυρήνες. Σε αυτά τα νέα δεδομένα hardware προσπαθούν να προσαρμοστούν οι σύγχρονες σχεσιακές βάσεις δεδομένων. Σε αυτο το εξάμηνο θα προσπαθήσουμε να δημιουργήσουμε ένα υποσύνολο μιας βάσης δεδομένων που διαχειρίζεται δεδομένα που βρίσκονται εξ ολοκλήρου στη RAM. Στο πρώτο μέρος θα ασχοληθούμε με την υλοποίηση του βασικό σχεσιακού τελεστή που θα χρειαστούμε για τα μετέπειτα κομμάτια της άσκησης που είναι ο τελεστής ζεύξης ισότητας.

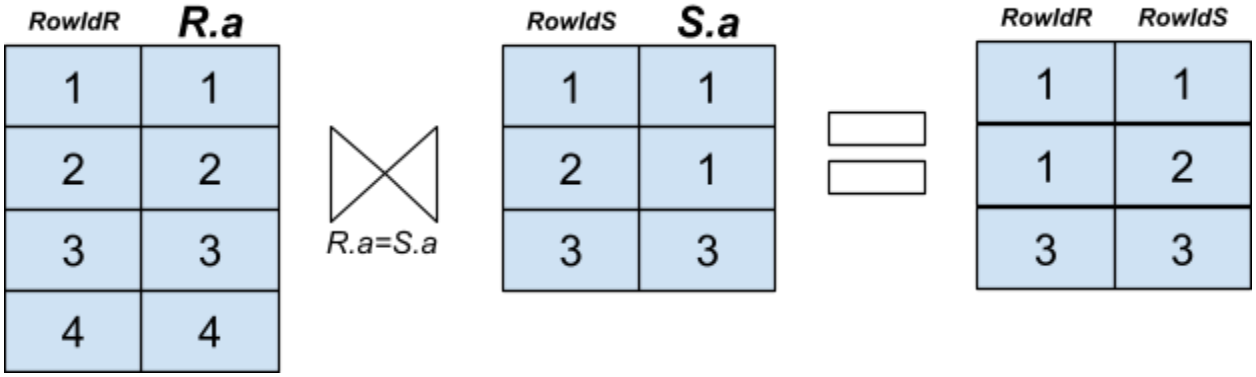
Τελεστής ζεύξης

Πρίν προχωρήσουμε στις δομές που θα χρειαστείτε να φτιάξετε για την πρώτη άσκηση θα δούμε ένα παράδειγμα με το πώς δουλεύει ένας τελεστής ζεύξης ισότητας. Στην πιο κάτω εικόνα, φαίνεται το αποτέλεσμα της πράξεις $R.\alpha = S.\alpha$ μεταξύ δύο σχεσιακών πινάκων (R, S):



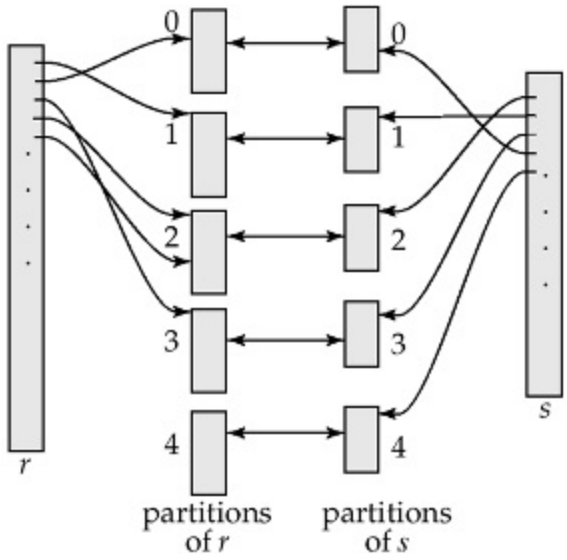
Η πιο πάνω απεικόνιση θεωρεί ότι οι πίνακες αποθηκεύονται στην μνήμη κατα σειρές (row store). Αυτό πρακτικά σημαίνει, ότι το κάθε στοιχείο μιας στήλης θα απέχει από το επόμενο του, τόσα στοιχεία μακριά όσες είναι και ο αριθμός των στηλών της αντίστοιχης σχέσης. Αυτο σε αρκετες περιπτωσεις είναι χρονοβόρο, γιατί καλούμαστε να διαχειριζόμαστε πολύ περισσότερη μνήμη, από όση στην πράξη χρειαζόμαστε. Μια άλλη τεχνική, αποθηκεύει τις σχέσεις ανά στήλες. Αυτό σημαίνει, ότι στη μνήμη τα στοιχεία μιας στήλης είναι σειριακά το ένα δίπλα στο άλλο. Με αυτόν τον τρόπο, αν θέλουμε να διαβάσουμε μια στήλη, “ακουμπάμε” πολύ λιγότερη μνήμη.

Για να εκμεταλλευτούμε την αποθήκευση ανά στήλες θα θέλαμε ο τελεστής ζεύξης να χρησιμοποιεί μόνο τις στήλες που γίνεται η πράξη και το αποτέλεσμα να είναι σε τέτοια μορφή ώστε να μπορούμε να ανακτήσουμε όλες τις στήλες από τους πίνακες που παίρνουν μέρος. Για τον λόγο αυτό εισάγουμε την έννοια του rowId. Το rowId είναι μια ένδειξη του αριθμού της γραμμής που ανήκει ένα συγκεκριμένο στοιχείο μιας στήλης. Με αυτόν τον τρόπο, μπορούμε να αποθηκεύσουμε κάθε στήλη μιας σχέσης, σαν μια ξεχωριστή σχέση με δύο στήλες, όπου η πρώτη στήλη είναι το rowId και η δεύτερη οι πραγματικές τιμές της στήλης. Το αποτέλεσμα της ζεύξης δύο τέτοιων σχέσεων, είναι μια σχέση με δύο στήλες όπου αναφέρονται στα ζευγάρια από rowIds τις πρώτης και της δεύτερης σχέσης που ταιριαζουν με βάση την συνθήκη της ισότητας. Η πιο κάτω εικόνα, απεικονίζει την εκτέλεση του τελεστή της ζεύξης του προηγούμενου παραδείγματος, χρησιμοποιώντας αποθήκευση ανα στήλες.



Partitioned Hash Join

Στο πρώτο μέρος της άσκησης καλείστε να υλοποιήσετε τον τελεστή join υποθέτοντας ότι τα δεδομένα σας είναι αποθηκευμένα ανά στήλες. Πιο συγκεκριμένα καλείστε να υλοποιήσετε τον Partitioned Hash Join (PHJ) αλγόριθμο. Η ιδέα του αλγορίθμου PHJ είναι να κομματιάσει τα δεδομένα από τις δύο σχέσεις σε τόσους κάδους, έτσι ώστε ο μεγαλύτερος σε μέγεθος κάδος να μπορεί να χωράει στην L2 Cache του επεξεργαστή. Οι κάδοι προκύπτουν εφαρμόζοντας στα δεδομένα των δύο σχέσεων την ίδια συνάρτηση κατακερματισμού hash1(). Εφόσον έχουμε κομματιάσει τα δεδομένα μας σε κάδους εφαρμόζοντας την ίδια συνάρτηση κατακερματισμού μπορούμε να συγκρίνουμε μόνο τα δεδομένα που ανήκουν στους ίδιους κάδους.

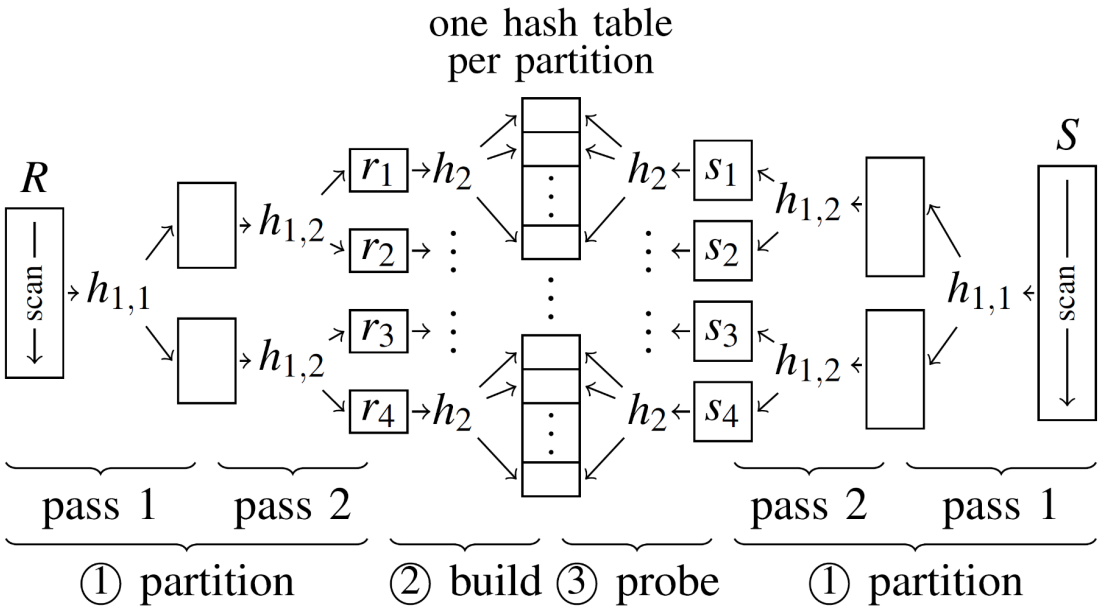


Πιο αναλυτικά, ο PHJ χωρίζεται σε τρεις φάσεις, τη φάση της τμηματοποίησης (partitioning), τη φάση του χτισίματος ευρετηρίων σε κάθε κάδο (building) και τη φάση της σύγκρισης (probing). Η φάση της τμηματοποίησης δημιουργεί τμήματα τόσο μεγάλα όσο να χωράνε στη L2 cache της CPU. Ανάλογα με το μέγεθος της σχέσης στην οποία δημιουργείται το ευρετήριο υπάρχουν οι εξής περιπτώσεις

(α) Δεν απαιτείται τμηματοποίηση. Η σχέση χωράει στη L2 cache.

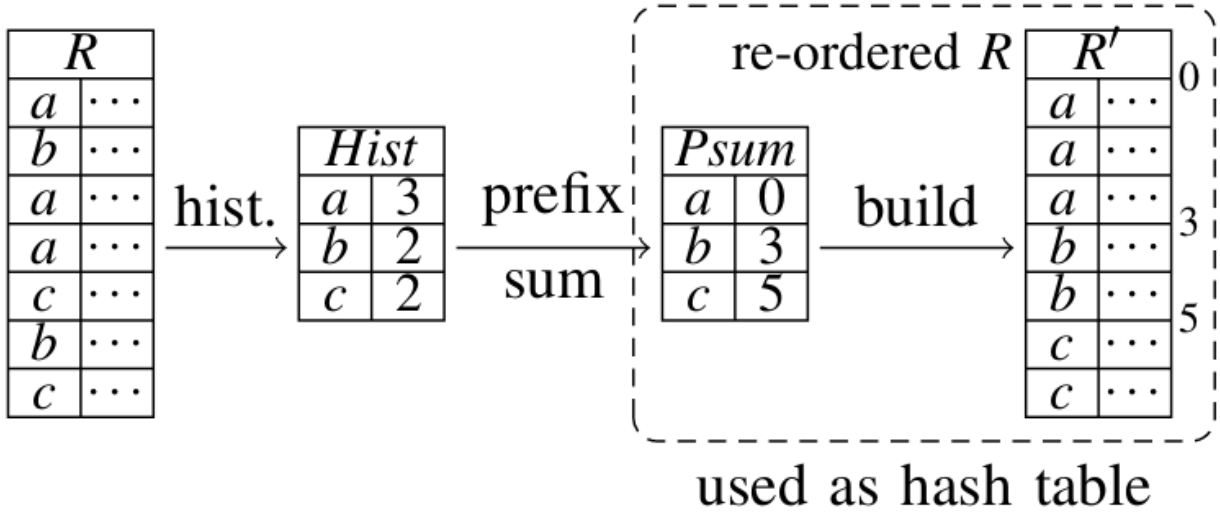
(β) Απαιτείται τμηματοποίηση με ένα πέρασμα και n bits που θα χρησιμοποιηθούν για το partitioning

(γ) Απαιτείται τμηματοποίηση με περισσότερα περάσματα.



Κατά τη φάση της τμηματοποίησης χρησιμοποιούμε σαν συνάρτηση κατακερματισμού hash1(), τον αριθμό που δίνουν τα n λιγότερο σημαντικά bits κάθε στοιχείου. Για παράδειγμα, αν έχουμε τον δυαδικό αριθμό 10010100, και $n = 3$, τοτε hash1(10010100) = 100 δηλαδή 4. Κατα αυτόν τον τρόπο

μπορούμε να κοματιάσουμε ένα πίνακα σε 2^n κάδους. Διαλέγοντας κάποιο n αρκετά μεγάλο μπορούμε να σπάσουμε τον πίνακα σε αρκετα μεγάλο πλήθος κάδων έτσι ώστε τα δεδομενα κάθε κάδου να χωράνε στην Cache του επεξεργαστή. Για να γίνει αυτή η διαδικασία αποδοτικά χρειάζεται πρώτα να δεσμεύσουμε ένα καινούργιο πίνακα με μέγεθος όσο ο αρχικός. Στον πίνακα αυτό θα αποθηκεύσουμε σε σειρα τα δεδομένα του κάθε κάδου. Για να το κάνουμε αυτό χρειάζεται να ξέρουμε σε ποια θέση του νέου πίνακα ξεκινάνε τα δεδομένα του κάθε κάδου. Για τον λόγο αυτό δημιουργούμε ένα πίνακα (ιστόγραμμα) 2^n θέσεων όπου στην κάθε θέση του κρατάμε το πλήθος των στοιχείων που υπάρχουν σε αυτόν τον κάδο. Στη συνέχεια θα δημιουργήσουμε το αθροιστικό του ιστόγραμμα (prefix sum) που θα δείχνει στη θέση από όπου θα ξεκινάει ο κάθε κάδος. Έχοντας αυτό το ιστόγραμμα μπορούμε με ένα πέρασμα του αρχικού πίνακα να γράψουμε τα δεδομένα στη σωστή θέση του νέου πίνακα. Η φάση της τμηματοποίησης φαίνεται στο πιο κάτω περιληπτικό σχήμα:



Εφόσον έχουμε δημιουργήσει τους κατάλληλους κάδους, και στις δύο σχέσεις, με την ίδια συνάρτηση κατακερματισμού, αυτό που μένει, είναι να συνδυάσουμε τους κάδους που κρατάνε δεδομένα, στα οποία η συνάρτηση κατακερματισμού έδωσε την ίδια τιμή. Αυτο δηλαδή που μένει, είναι να επιλέξουμε τα ζευγάρια κάδων που ταιριάζουν από την σχέση R και S, να χτίσουμε τα κατάλληλα ευρετήρια σε έναν από τους δύο κάδους και να δούμε ποια δεδομένα της μιας σχέσης ταιριάζουν με τα δεδομένα της άλλης.

Το ευρετήριο που θα χτίσουμε θα έχει τη μορφή ενός πίνακα κατακερματισμού που θα ακολουθεί την τεχνική hopscotch hashing [2]. Ο αλγόριθμος χρησιμοποιεί ένα μοναδικό πίνακα από n θέσεις. Για κάθε θέση, η *γειτονιά* του είναι ένας μικρός αριθμός από Η συνεχόμενες θέσεις (θέσεις δηλαδή με δείκτες κοντά στο αρχική θέση). Η επιθυμητή ιδιότητα της γειτονιάς είναι ότι το κόστος της εύρεσης ενός στοιχείου στις θέσεις της γειτονιάς είναι παρεμφερές με το κόστος της εύρεσης του στην ίδια τη θέση (επειδη΄θα βρίσκονται στην ίδια cache line). Το μέγεθος της γειτονιάς πρέπει να είναι αρκετό για να επιτρέπει πολυπλοκότητα αναζήτησης O(log(n)) στη χειρότερη περίπτωση, αλλά O(1) κατά μέσο όρο. Αν κάποια γειτονιά γεμίσει, ο πίνακας θα πρέπει να ανασχηματιστεί (rehash).

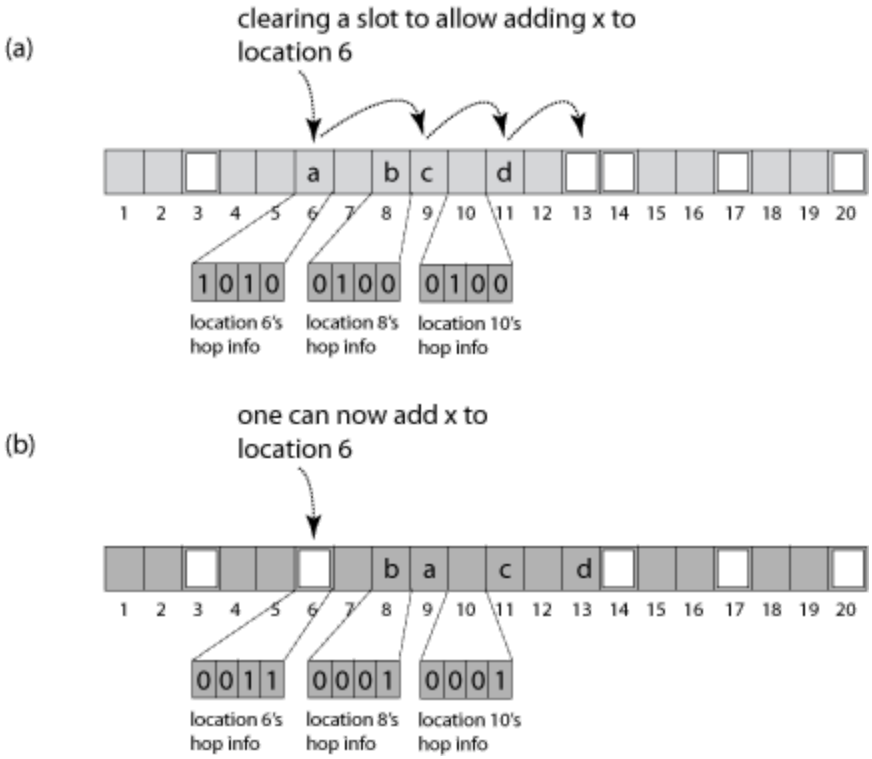
Στο hopscotch hashing, ένα στοιχείο θα εισαχθεί πάντα και θα ευρεθεί πάντα στη γειτονιά της θέσης που προκύπτει από τη συνάρτηση κατακερματισμού. Με άλλα λόγια, θα είναι πάντα είτε στην

αρχική θέση (original hash value) είτε σε μία από τις επόμενες $H-1$ γειτονικές θέσεις. Προκειμένου να επιταχυνθεί η αναζήτηση, κάθε θέση περιλαμβάνει ένα hop-information bitmap που υποδεικνύει ποιες από τις επόμενες $H-1$ θέσεις περιέχουν στοιχεία που αρχικά ήταν να εισαχθούν στην τρέχουσα θέση. Με αυτό τον τρόπο, ένα στοιχείο μπορεί να βρεθεί γρήγορα, ψάχνοντας μόνο σε αυτές τις θέσεις και αγνοώντας τις υπόλοιπες.

Ένας αλγόριθμός προσθήκης ενός στοιχείου x με hash value i μέσω hopscotch hashing παρουσιάζεται παρακάτω:

1. Αν το hop-information bitmap δείχνει ότι και στις H θέσεις βρίσκονται ήδη στοιχεία που αντιστοιχούν στη θέση i, τότε ο πίνακας έχει γεμίσει και χρειάζεται rehashing.
2. Ξεκινώντας από τη θέση i, βρείτε μέσω γραμμικής αναζήτησης μία κενή θέση j (αν δεν υπάρχει κενή θέση, ο πίνακας έχει γεμίσει).
3. Όσο $j - i \ mod \ n \geq H$, μετακινήστε την κενή θέση προς το i ως εξής:
    a. Αναζητήστε τις -1θέσεις που προηγούνται του j για ένα στοιχείο y, του οποίου το hash value k βρίσκεται το πολύ σε απόσταση $H-1$ από το j, δηλαδή $j - k \ mod \ n < H$. Η αναζήτηση διευκολύνεται αρκετά χρησιμοποιώντας τα hop-information bitmaps.
    b. Αν δεν υπάρχει τέτοιο στοιχείο y, ο πίνακας έχει γεμίσει.
    c. Μετακινήστε το y στο j, δημιουργώντας μία νέα κενή θέση πιο κοντά στο i.
    d. Δώστε στο j την τιμή της κενής θέσης που άδειασε από το y και επαναλάβετε.
4. Αποθηκεύσετε το x στη θέση j και επιστρέψτε.

Μέσω του hopscotch hashing, η κενή θέση μετακινείται πιο κοντά στην αρχικά υπολογισμένη θέση.



Στο παράδειγμα που φαίνεται, $= 4$. Οι γκρι θέσεις είναι κατειλημμένες. Στο τμήμα (a), το στοιχείο x προστίθεται με hash value 6. Η γραμμική αναζήτηση βρίσκει ότι η θέση 13 είναι άδεια.

Επειδή η θέση 13 είναι περισσότερο από 4 θέσεις μακριά από τη θέση 6, ο αλγόριθμος αναζητά μία κοντινότερη θέση για ανταλλαγή με τη 13. Η πρώτη θέση βρίσκεται στη θέση 10, που βρίσκεται $H - 1 = 3$ θέσεις μακριά. Το hop information bitmap της θέσης αυτής υποδεικνύει ότι το στοιχείο d που βρίσκεται στη θέση 11 μπορεί να μεταφερθεί στη θέση 13.

Μετά τη μετακίνηση του d, η θέση 11 συνεχίζει να είναι αρκετά μακριά από τη θέση 6. Ο αλγόριθμος εξετάζει τη θέση 8. Το hop information bitmap της θέσης υποδεικνύει ότι το στοιχείο c στη θέση 9 μπορεί να μετακινηθεί στη θέση 11.

Με τη θέση 9 άδεια, το στοιχείο a μπορεί να εισαχθεί στη θέση αυτή. Το μέρος (b) δείχνει την κατάσταση του πίνακα λίγο πριν την προσθήκη του x.

Πρότυπα συναρτησεων και δομών

Πιο κάτω δίνονται οι ορισμοί των συναρτήσεων και των βασικών δομών τους που καλείστε να υλοποιήσετε:

```
/** Type definition for a tuple */
```

```
struct tuple {
     int32_t key;
     int32_t payload;
};

/**
 * Type definition for a relation.
 * It consists of an array of tuples and a size of the relation.
 */
struct relation {
  tuple  *tuples;
  uint32_t num_tuples;
};

/**
 * Type definition for a relation.
 * It consists of an array of tuples and a size of the relation.
 */
struct result {
 ...
};

/** Partitioned Hash Join**/
result* PartitionedHashJoin(relation *relR, relation *relS);
```

Οι πιο πάνω ορισμοί είναι ενδεικτικοί. Αν θεωρείτε ότι σας περιορίζουν μπορείτε να τους αλλάξετε.

Παράδοση εργασίας

Η εργασία είναι ομαδική, **2 ή 3 ατόμων**.
Προθεσμία παράδοσης: 7/11/2022
Γλώσσα υλοποίησης: C / C++ χωρίς χρήση stl.
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 χρήση git είναι υποχρεωτική**.
Επιπλέον, εκτός από τον πηγαίο κώδικα, θα παραδώσετε μια σύντομη αναφορά, με τις σχεδιαστικές σας επιλογές καθώς και να εφαρμόσετε ελέγχους ως προς την ορθότητα του λογισμικού με τη χρήση ανάλογων βιβλιοθηκών (Software testing).

Αναφορές

1. Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. *Proc. of the 29th International Conference on Data Engineering (ICDE 2013)*, Brisbane, Australia, April 2013.
2. M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In DISC ’08: Proceedings of the 22nd international symposium on Distributed Computing, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0