# Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

## Χειμερινό Εξάμηνο 2025-2026

### Άσκηση 1 – Παράδοση: Δευτέρα 3 Νοεμβρίου 2025

## Join Pipeline Optimization

Η εργασία βασίζεται στον προγραμματιστικό διαγωνισμό SIGMOD 2025 (https://sigmod-contest-2025.github.io), όπου δίνεται το join pipeline και τα φιλτραρισμένα δεδομένα εισόδου και καλείστε να υλοποιήσετε ένα αποδοτικό αλγόριθμο ζεύξης (join) για να μειωθεί ο χρόνος εκτέλεσης των ζεύξεων. Συγκεκριμένα θα αναπτύξετε κώδικα χρησιμοποιώντας την παρεχόμενη βασική λύση και θα πρέπει να υλοποιηθεί η παρακάτω συνάρτηση στο `src/execute.cpp`

```
ColumnarTable execute(const Plan& plan, void* context);
```

## Αλγόριθμοι ζεύξης

Στο πρώτο μέρος της άσκησης καλείστε να υλοποιήσετε τον τελεστή join υλοποιώντας 3 εκδοχές του Hash Join αλγορίθμου. Στη βασική λύση, ο πίνακας κατακερματισμού είναι το std::unordered_map της C++/STL. Καλείστε να υλοποιήσετε 3 διαφορετικές εκδοχές αλγορίθμων κατακερματισμού αντικαθιστώντας τον αλγόριθμο στη βασική λύση.

### Αλγόριθμοι κατακερματισμού

Ο κατακερματισμός (hashing) είναι μια τεχνική που αντιστοιχίζει δεδομένα σε έναν πίνακα σταθερού μεγέθους χρησιμοποιώντας μια συνάρτηση κατακερματισμού. Η συνάρτηση κατακερματισμού (hash function) λαμβάνει μια είσοδο (ή κλειδί) και επιστρέφει ένα ευρετήριο (index) στον πίνακα κατακερματισμού, όπου αποθηκεύεται η αντίστοιχη τιμή. Ένας πίνακας κατακερματισμού (hashtable) χρησιμοποιεί αυτό το ευρετήριο για την αποθήκευση των δεδομένων, καθιστώντας τον πολύ αποτελεσματικό για την αναζήτηση και την πρόσβαση σε στοιχεία. Είναι μια προηγμένη τεχνική που βασίζεται στην ανοιχτή διευθυνσιοδότηση. Η βασική ιδέα είναι να ελαχιστοποιηθεί η διακύμανση στην απόσταση μεταξύ κάθε κλειδιού και της αρχικής του θέσης υποδοχής.

Χειρισμός συγκρούσεων στον κατακερματισμό: Όταν πολλά κλειδιά κατακερματίζονται στον ίδιο δείκτη, συμβαίνει μια σύγκρουση.
Γραμμική διερεύνηση (linear probing): Αναζητείται η επόμενη διαθέσιμη θέση. Τα κλειδιά μπορεί να τοποθετηθούν μακριά από την αρχική τους θέση, επιβραδύνοντας τις αναζητήσεις.

## Αλγόριθμος κατακερματισμού Robin Hood

Στο Robin Hood Hashing, όταν συμβαίνει μια σύγκρουση, ο αλγόριθμος διασφαλίζει ότι τα κλειδιά τοποθετούνται όσο το δυνατόν πιο κοντά στην ιδανική τους θέση. Εάν ένα κλειδί εισαχθεί πιο μακριά από την αρχική του θέση από ένα υπάρχον κλειδί, μετατοπίζει το τρέχον κλειδί. Αυτή η διαδικασία "ληστεύει" τα κλειδιά που βρίσκονται πιο κοντά στις ιδανικές τους θέσεις (τα "πλούσια") για να δώσει χώρο στα κλειδιά που βρίσκονται πιο μακριά (τα "φτωχά"), ελαχιστοποιώντας τη συνολική απόσταση και βελτιώνοντας την αποτελεσματικότητα του πίνακα κατακερματισμού.

Στον κατακερματισμό, το Probe Sequence Length (PSL **)** αναφέρεται στον αριθμό των βημάτων που απαιτούνται για την εύρεση ενός κλειδιού στον πίνακα κατακερματισμού, ειδικά όταν συμβαίνουν συγκρούσεις. Ενώ ο μέσος χρόνος αναζήτησης σε έναν πίνακα κατακερματισμού χρησιμοποιώντας linear prbing είναι **O(1)** , ο πραγματικός αριθμός των απαιτούμενων βημάτων μπορεί να διαφέρει σημαντικά ανάλογα με τον τρόπο κατανομής των δεδομένων στον πίνακα. Όταν εισάγουμε τιμές σε έναν πίνακα κατακερματισμού, θέλουμε να διατηρήσουμε το PSL όσο το δυνατό μικρότερο.

Όταν εισάγουμε μια τιμή v σε έναν πίνακα κατακερματισμού χρησιμοποιώντας linear probing, ακολουθούμε τα εξής βήματα:

- Βήμα 1: Ξεκινήστε από τον κάδο όπου κατακερματίζεται το v.
- Βήμα 2: Εάν ο κάδος είναι ήδη κατειλημμένος, προχωρήστε στον επόμενο κάδο.
- Βήμα 3: Συνεχίστε να ελέγχετε τους επόμενους κάδους μέχρι να βρεθεί ένας άδειος (μηδενικός κάδος
- Βήμα 4: Μόλις βρεθεί ένας άδειος κάδος, εισάγετε το v μαζί με το PSL ( του.

Κανόνες κατά τη διάρκεια της διαδικασίας

- Οι τιμές που κατακερματίζονται στον κάδο i προηγούνται πάντα των τιμών που κατακερματίζονται στον κάδο i+1.
- Ξεκινάμε από τον κατακερματισμένο κάδο και διασφαλίζουμε ότι οι διαδοχικοί κάδοι vpsl (μήκος ακολουθίας ανιχνευτή) είναι κατειλημμένοι.
- Ο βρόχος συνεχίζεται μέχρι να βρεθεί ένας άδειος (μηδενικός) κάδος.
- Στον πρώτο άδειο κάδο (b[p]) που θα βρεθεί, εισάγεται το v.

Για να αναφερθείτε σε μια συγκεκριμένη ομάδα:

- **b[p].v** → Τιμή αποθηκευμένη στον κάδο **p** .
- **b[p].psl** → PSL της τιμής στον κάδο **p** .

Στο Robin Hood Hashing, κατά την εισαγωγή κλειδιών στον πίνακα κατακερματισμού, το πρώτο κλειδί κατακερματίζεται και τοποθετείται στην ιδανική του θέση, με μήκος ακολουθίας ανιχνευτή (PSL) 0. Καθώς εισάγονται περισσότερα κλειδιά, συμβαίνουν συγκρούσεις όταν κατακερματίζονται στην ίδια θέση. Αντί να εισάγεται το κλειδί στην επόμενη διαθέσιμη υποδοχή, το Robin Hood Hashing συγκρίνει το

PSL του υπάρχοντος κλειδιού με αυτό του νέου κλειδιού. Εάν το νέο κλειδί έχει υψηλότερο PSL, αντικαθιστά το υπάρχον κλειδί για να διατηρηθεί η δικαιοσύνη. Αυτό διασφαλίζει ότι κανένα κλειδί δεν βρίσκεται πολύ μακριά από την ιδανική του θέση, καθώς τα κλειδιά με μεγαλύτερα PSL τοποθετούνται πιο κοντά στις προβλεπόμενες θέσεις τους.

Καθώς εισάγονται περισσότερα κλειδιά, ο αλγόριθμος συνεχίζει να ανταλλάσσει κλειδιά με βάση τις τιμές PSL τους, αποτρέποντας την ομαδοποίηση και διασφαλίζοντας ότι κανένα κλειδί δεν έχει μια παράλογα μεγάλη ακολουθία βημάτων. Στο τέλος, το Robin Hood Hashing διατηρεί τον πίνακα ισορροπημένο, διασφαλίζοντας αποτελεσματικές αναζητήσεις και ελαχιστοποιώντας τη μέγιστη απόσταση που πρέπει να διανύσει ένα κλειδί από την ιδανική του θέση.

## Αλγόριθμος κατακερματισμού hopscotch

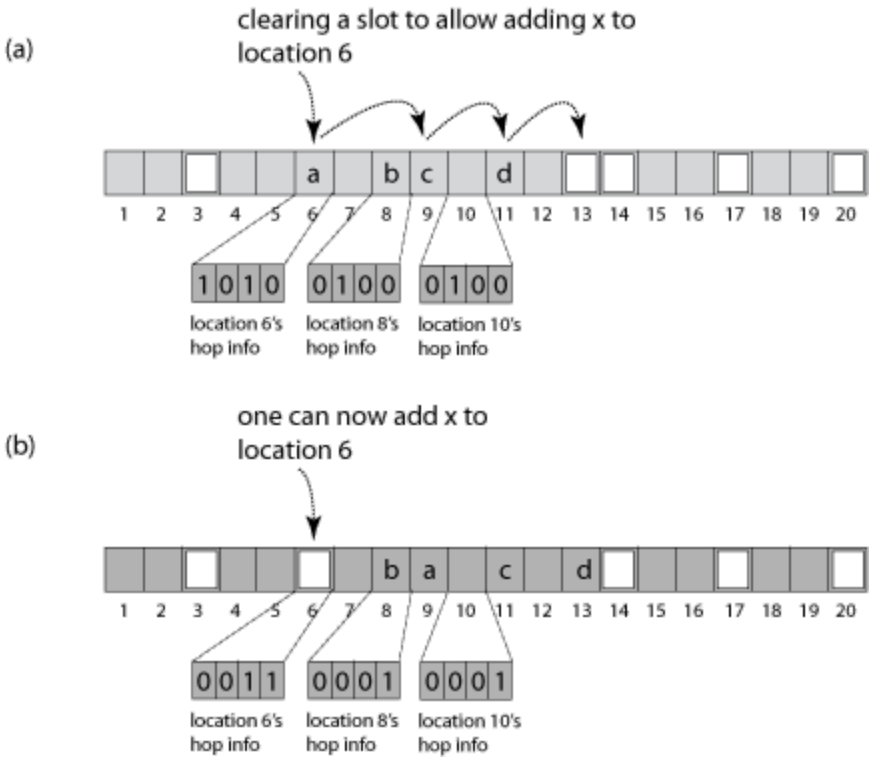
Ο αλγόριθμος hopscotch hashing [2] χρησιμοποιεί ένα μοναδικό πίνακα από $n$ θέσεις. Για κάθε θέση, η *γειτονιά* του είναι ένας μικρός αριθμός από H συνεχόμενες θέσεις (θέσεις δηλαδή με δείκτες κοντά στο αρχική θέση). Η επιθυμητή ιδιότητα της γειτονιάς είναι ότι το κόστος της εύρεσης ενός στοιχείου στις 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 $H-1$ θέσεις 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, H=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.

## Cuckoo hashing

Η βασική ιδέα είναι ότι αντί να έχουμε έναν πίνακα κατακερματισμού T, και μια συνάρτηση κατακερματισμού h (όπως στην κλασική περίπτωση), θα έχουμε δύο πίνακες $T_1$, $T_2$ και δύο συναρτήσεις κατακερματισμού: $h_1$, $h_2$. Ένα κλειδί k, αντί να έχει μία θέση T[h(k)], θα έχει δύο: είτε θα βρίσκεται στη θέση $T_1[h_1(k)]$, είτε στη θέση $T_2(h_2(k)]$..

Ο τρόπος διαχείρισης των συγκρούσεων στον κατακερματισμό κούκου είναι ο εξής: Έστω ότι θέλουμε να εισάγουμε ένα στοιχείο $k_1$. Το στοιχείο αυτό πάντα θα τοποθετείται στη θέση του στον πρώτο πίνακα, δηλαδή στη θέση $T_1(h_1(k_1)]$. Αν η θέση αυτή ήταν κενή τότε δεν χρειάζεται κάτι άλλο.

Αν η θέση ήταν κατειλημμένη από κάποιο άλλο στοιχείο, $k_2$ (για το οποίο προφανώς ισχύει $h_1(k_1)=h_1(k_2)$), εμείς δεν ψάχνουμε άλλη θέση, αλλά εισάγουμε το $k_1$ πάρα ταύτα, διώχνοντας ουσιαστικά το $k_2$

Το $k_2$ θα τοποθετηθεί στον άλλο πίνακα, δηλαδή στη θέση $T_2[h_2(k_2)]$, η οποία πιθανόν να είναι κενή. Αν πάλι δεν είναι, και υπάρχει εκεί ένα στοιχείο $k_3$, τότε διώχνουμε το $k_3$ και το τοποθετούμε στον πρώτο πίνακα. Η ίδια διαδικασία συνεχίζεται, διώχνοντας κάθε φορά το υπάρχον στοιχείο, και τοποθετώντας το στον άλλο πίνακα, μέχρι να βρούμε κάποια θέση που να είναι κενή.

Τέλος, υπάρχουν δύο περιπτώσεις στις οποίες θα χρειαστεί να κάνουμε rehash:

Αν ο βαθμός πληρότητας του πίνακα (συνολικός αριθμός στοιχείων διά το μέγεθος) ξεπεράσει το 0.5, όπως και στους κλασικούς πίνακες κατακερματισμού.

Αν η διαδικασία εισαγωγής πέσει σε κύκλο: αν δηλαδή φτάσουμε σε κάποιο στοιχείο $k_n$, το οποίο στη θέση που προσπαθούμε να το βάλουμε, βρίσκεται κάποιο στοιχείο που έχουμε ήδη μετακινήσει! Ένας απλός τρόπος να εντοπίσουμε τέτοιους κύκλους είναι να μετράμε τον αριθμό στοιχείων που μετακινήθηκαν. Αν φτάσουμε στα n (όσα τα συνολικά στοιχεία που υπάρχουν στον πίνακα), τότε αναγκαστικά υπάρχει κύκλος.

Και στις δύο περιπτώσεις, το rehash γίνεται ως συνήθως: διπλασιάζουμε τον πίνακα, και εισάγουμε τα στοιχεία στις νέες τους θέσεις. Προσοχή όμως: η περίπτωση (2) μπορεί να ξανασυμβεί, οπότε ίσως χρειαστούμε παραπάνω από ένα διαδοχικά rehash!

## Ανάλυση απόδοσης

Υλοποιήστε και τις 3 διαφορετικές μεθόδους και δείτε πώς συμπεριφέρονται (χρονικά) με διαφορετικά datasets.

# Παράδοση εργασίας

Η εργασία είναι ομαδική, **2 ή 3 ατόμων**.
**Γλώσσα υλοποίησης:** C++
**Περιβάλλον υλοποίησης:** Linux (gcc >= 9.4+).

**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**.
Στο αρχείο README.md θα αναφέρονται τα εξής:

- Ονοματεπώνυμα και ΑΜ των μελών της ομάδας
- Αναφορά στο ποιο μέλος της ομάδας ασχολήθηκε με ποιο αντικείμενο

Επιπλέον, εκτός από τον πηγαίο κώδικα, θα παραδώσετε μια σύντομη αναφορά με την ανάλυση της απόδοσης των αλγορίθμων. Θα πρέπει ακόμα να εφαρμόσετε ελέγχους ως προς την ορθότητα του λογισμικού με τη χρήση ανάλογων βιβλιοθηκών (Software testing). Η ορθότητα τυχόν μεταβολών θα ελέγχεται με αυτοματοποιημένο τρόπο σε κάθε commit/push μέσω github actions.

## Αναφορές


1. P. Cellis. Robin Hood Hashing. PhD Thesis 1986, University of Waterloo, Canada, https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf
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. https://doi.org/10.1007/978-3-540-87779-0_24
3. Pagh, R., Rodler, F.F. (2001). Cuckoo Hashing. In: auf der Heide, F.M. (eds) Algorithms — ESA 2001. ESA 2001. Lecture Notes in Computer Science, vol 2161. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44676-1_10