Παραδοτέο 1

Μάθημα: Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Μέλη Ομάδας:

Ραμαντάν Κονόμι - ΑΜ: 1115201800281

Θεμιστοκλής Παπαθεοφάνους - ΑΜ: 1115202100227

Μάριος Γιαννόπουλος - ΑΜ: 1115202000032

Πίνακας Περιεχομένων

Ανάλυση της Κλάσης **HopscotchTable**

HopscotchTable

Η κλάση `HopscotchHashTable` υλοποιεί τον αλγόριθμο Hopscotch Hashing, μια υβριδική τεχνική που συνδυάζει τα πλεονεκτήματα της ανοικτής διευθυνσιοδότησης (open addressing) και της αλυσιδωτής κατακερμάτισης (chaining). Ο βασικός στόχος είναι η διατήρηση κάθε κλειδιού σε μια **γειτονιά (neighborhood)** σταθερού μεγέθους H κοντά στην αρχική του θέση κατακερματισμού. Αυτό εγγυάται σταθερό χρόνο αναζήτησης $O(1)$ στη χειρότερη περίπτωση, καθώς η αναζήτηση περιορίζεται αυστηρά εντός της γειτονιάς.

Στην υλοποίησή μας, το μέγεθος της γειτονιάς ορίζεται ως $H = 64$, εκμεταλλευόμενοι πλήρως τους 64-bit επεξεργαστές μέσω bitwise πράξεων.

Δομές Δεδομένων

Η θεμελιώδης δομή είναι το `Bucket`, το οποίο περιέχει πληροφορίες για τον έλεγχο της γειτονιάς και την αποθήκευση των δεδομένων.

Bucket & Hop-Information Bitmap

Κάθε θέση στον πίνακα (`table`) είναι ένα `Bucket` που περιέχει:

```
template <typename Key>
struct Bucket {
    uint64_t bitmask; // Hop-information bitmap (H=64)
    Key key;
    uint32_t first_segment;
    uint32_t last_segment;
    uint16_t count;
    bool occupied;
};
```

`bitmask` (Hop Bitmap): Ένας ακέραιος 64-bit που λειτουργεί ως bitmap. Αν το i-οστό bit του `bitmask` στη θέση B είναι '1', σημαίνει ότι το κλειδί που αντιστοιχεί στη θέση B (βάσει hash) βρίσκεται φυσικά αποθηκευμένο στη θέση $B + i$. **`key, occupied`**: Το αποθηκευμένο κλειδί και η κατάσταση της θέσης. **`first/last_segment`**: Δείκτες προς τα `value_store` και `segments` για την υποστήριξη διπλότυπων κλειδιών (παρόμοια λογική με τα Robin Hood και Cuckoo).

Συναρτήσεις Κατακερματισμού & Hardware Acceleration

Για την μεγιστοποίηση της απόδοσης, η υλοποίηση επιλέγει δυναμικά τη συνάρτηση κατακερματισμού. Για κλειδιά τύπου `int32_t`, χρησιμοποιούνται **hardware intrinsic εντολές** (CRC32), οι οποίες εκτελούνται απευθείας από τον επεξεργαστή:

```
static inline size_t hash_int32(int32_t key) noexcept {
    #if defined(__aarch64__)
        return __builtin_arm_crc32w(key, 0); // ARM NEON/CRC
    #elif defined(__x86_64__)
        return __builtin_ia32_crc32si(key, 0); // Intel SSE4.2
    #endif
}
```

Αυτό προσφέρει εξαιρετικά γρήγορο κατακερματισμό και καλή κατανομή, μειώνοντας τις συγκρούσεις.

Μηχανισμός Εισαγωγής (Insertion & Relocation)

Η εισαγωγή είναι η πιο πολύπλοκη διαδικασία του Hopscotch Hashing, καθώς πρέπει να διατηρηθεί η ιδιότητα της γειτονιάς.

1. **Εύρεση Θέσης:** Υπολογίζεται η αρχική θέση `base_index`. Αν η θέση είναι κατειλημμένη από το ίδιο κλειδί, γίνεται ενημέρωση (duplicate handling).
2. **Γραμμική Αναζήτηση:** Αναζητείται η πλησιέστερη κενή θέση (`free_index`) ξεκινώντας από το `base_index`.
3. **Έλεγχος Απόστασης:**
   - Αν `free_index - base_index < H`, το κλειδί τοποθετείται και το αντίστοιχο bit στο `bitmask` του `base_index` ενεργοποιείται.
   - Αν `free_index - base_index >= H`, απαιτείται **μετακίνηση (relocation)**.

Relocation (Μετακίνηση της Κενής Θέσης)

Όταν η κενή θέση είναι πολύ μακριά, ο αλγόριθμος προσπαθεί να τη «φέρει» πιο κοντά μέσω διαδοχικών ανταλλαγών (`relocate`). Αναζητά ένα κατειλημμένο κελί `swap_index` στην περιοχή $[\texttt{free\_index} - H + 1, \texttt{free\_index}]$ τέτοιο ώστε το κλειδί που περιέχει να μπορεί να μετακινηθεί στο `free_index` χωρίς να παραβιάσει τη δική του γειτονιά. Αν βρεθεί, τα στοιχεία ανταλλάσσονται και η κενή θέση μετακινείται πιο κοντά στο στόχο. Η διαδικασία επαναλαμβάνεται μέχρι η κενή θέση να εισέλθει στη γειτονιά του νέου κλειδιού.

Αναζήτηση (Search Optimization)

Η αναζήτηση είναι εξαιρετικά αποδοτική και Cache-friendly.

1. Μεταβαίνουμε στο `table[hash(key)]`.
2. Διαβάζουμε το `bitmask`. Αν είναι 0, το κλειδί δεν υπάρχει (άμεσος τερματισμός).
3. Αν το bitmask δεν είναι 0, χρησιμοποιούμε την εντολή **`__builtin_ctzll`** (Count Trailing Zeros) για να βρούμε αμέσως τις υποψήφιες θέσεις μέσα στη γειτονιά, χωρίς να σαρώνουμε περιττά κελιά.

```
while (temp_mask) {
    int i =  __builtin_ctzll(temp_mask); // Εύρεση επόμενου πιθανού index σε O(1)
    // ... έλεγχος κλειδιού ...
    temp_mask &= (temp_mask - 1); // Αφαίρεση του bit που ελέγχθηκε
}
```

Ανάλυση της Κλάσης **RobinHoodTable**

RobinHoodTable

Η κλάση `RobinHoodTable` υλοποιεί τον Κατακερματισμό Robin Hood, μια στρατηγική επίλυσης συγκρούσεων ανοικτής διευθυνσιοδότησης που εμπνέεται από το ρητό «κλέβει από τους πλούσιους και δίνει στους φτωχούς». Ο κύριος στόχος της είναι η **ελαχιστοποίηση της διακύμανσης** των μηκών των αλυσίδων διερεύνησης (Probe Sequence Length - PSL) για όλα τα κλειδιά, βελτιώνοντας δραστικά τον χρόνο αναζήτησης στην **χειρότερη περίπτωση** και την απόδοση της κρυφής μνήμης (cache locality).

Δομές Δεδομένων

Κάθε καταχώρηση στον πίνακα αποθηκεύει, εκτός από το κλειδί, το Μήκος Ακολουθίας Διερεύνησης (PSL).

Probe Sequence Length (PSL)

Το PSL μετρά την απόσταση του κλειδιού από την **ιδανική (primal) θέση** του, όπως αυτή ορίζεται από τη συνάρτηση κατακερματισμού.

```
PSL = (i - p) mod m
```

- i: τρέχουσα θέση του κλειδιού.
- p: ιδανική θέση (αρχικό hash index).
- m: μέγεθος πίνακα.
- **PSL = 0**: Το κλειδί βρίσκεται στην ιδανική του θέση (χωρίς συγκρούσεις).
- **PSL = +1**: Το PSL αυξάνεται κατά 1 για κάθε σύγκρουση κατά την εισαγωγή.

Συναρτήσεις Κατακερματισμού και Μεγέθη Πινάκων

Συνάρτηση Κατακερματισμού - Fibonacci Hashing

Η υλοποίηση χρησιμοποιεί **πολλαπλασιαστικό κατακερματισμό** (multiplicative hashing) με σταθερές που βασίζονται στον χρυσό λόγο (Fibonacci Hashing) για κλειδιά τύπου `int32_t`. Η τεχνική αυτή χρησιμοποιεί δύο διαφορετικά «κομμάτια» του κατακερματισμού συνδυασμένα με bitwise OR για να επιτύχει:

- **Φαινόμενο Χιονοστιβάδας (Avalanche Effect)**.
- Ομοιόμορφη κατανομή.
- Μειωμένο clustering.

Μέγεθος Πίνακα και Συντελεστής Φόρτου

- **Μέγεθος Πίνακα**: Είναι πάντα **δύναμη του δύο**, το οποίο επιτρέπει τον γρήγορο υπολογισμό του δείκτη μέσω bitwise AND: `index = h(k) & (m - 1)`, αποφεύγοντας δαπανηρές πράξεις modulo.
- **Συντελεστής Φόρτου (Load Factor)**: Διατηρείται στο βέλτιστο επίπεδο του **≈ 60%** (με εύρος 60-75%) για να εξασφαλίσει χαμηλό μέσο και φραγμένο χειρότερο μήκος διερεύνησης.

Στρατηγική Επίλυσης Συγκρούσεων (Εισαγωγή)

Ο αλγόριθμος βασίζεται στον κανόνα εκτόπισης: «Ο πλούσιος δίνει τη θέση του στον φτωχό».

Διαδικασία Εκτόπισης (The Swap)

Όταν εισάγεται ένα νέο κλειδί με PSL P_{new} σε μια θέση κατειλημμένη από κλειδί με PSL P_{old}:

- **Κανόνας Robin Hood:** Αν $P_{\text{new}} > P_{\text{old}}$, το εισερχόμενο κλειδί (*ο «φτωχός»* - δηλαδή αυτό που βρίσκεται πιο μακριά από την ιδανική του θέση) εκτοπίζει το υπάρχον κλειδί (*ο «πλούσιος»*).
- **Συνέχεια Διερεύνησης:** Το εκτοπισμένο κλειδί (το οποίο τώρα είναι «στον αέρα») συνεχίζει τη γραμμική διερεύνηση (linear probing) με το PSL του **αυξημένο κατά 1**.

Η διαδικασία συνεχίζεται έως ότου βρεθεί μια κενή θέση.

Βελτιστοποίηση Πρόωρου Τερματισμού (Αναζήτηση)

Η σημαντικότερη βελτίωση είναι ο **πρόωρος τερματισμός** κατά την αναζήτηση, ο οποίος εκμεταλλεύεται το Robin Hood invariant.

Συνθήκη Τερματισμού

Κατά την αναζήτηση ενός κλειδιού, αν το τρέχον μήκος διερεύνησης (`vpsl`) υπερβεί το PSL (`table[p].psl`) του κλειδιού που βρίσκεται στην τρέχουσα θέση (p), η αναζήτηση **τερματίζεται άμεσα** με αποτέλεσμα «δεν βρέθηκε».

```
if (vpsl > table[p].psl) break; // Key not found
```

Χαρακτηριστικά Απόδοσης

Χρονική Πολυπλοκότητα

- **Μέση Περίπτωση:** $O(1)$ για εισαγωγή και αναζήτηση.
- **Χειρότερη Περίπτωση:** $O(n)$, αλλά με πολύ μικρότερους σταθερούς παράγοντες σε σχέση με τη συνήθη γραμμική διερεύνηση (linear probing).
- **Αναμενόμενο Μήκος Διερεύνησης:** ≈ 2.5 σε συντελεστή φόρτου 60%.

Γιατί Λειτουργεί

Λόγω της στρατηγικής Robin Hood, κάθε κλειδί έχει εγγυημένα PSL μικρότερο ή ίσο με οποιοδήποτε κλειδί που βρίσκεται μπροστά του στην αλυσίδα διερεύνησης. Εάν το αναζητούμενο κλειδί υπήρχε, **θα είχε εκτοπίσει** το κλειδί με το μικρότερο PSL που συναντήθηκε. Επομένως, ο πρόωρος τερματισμός μειώνει το μήκος διερεύνησης κατά 50% ή περισσότερο σε αποτυχημένες αναζητήσεις.

Ανάλυση της Κλάσης **CuckooTable**

CuckooTable

Η κλάση `CuckooTable<Key>` υλοποιεί τον αλγόριθμο Κατακερματισμού Cuckoo (Cuckoo Hashing), μια προηγμένη τεχνική κατακερματισμού που εγγυάται σταθερό χρόνο αναζήτησης $O(1)$ στην **χειρότερη** περίπτωση. Η υλοποίηση είναι βελτιστοποιημένη για αποδοτικότητα μνήμης και κρυφής μνήμης (Cache Efficiency), χρησιμοποιώντας έναν **μηχανισμό κοινόχρηστης αποθήκευσης** για πολλαπλές τιμές και **inline value optimization** για μοναδικές τιμές. Σε αντίθεση με τις μεθόδους ανοικτής διευθυνσιοδότησης που βασίζονται σε chaining ή γραμμική διερεύνηση (linear probing), το Cuckoo Hashing χρησιμοποιεί δύο πίνακες και δύο συναρτήσεις κατακερματισμού για να εξασφαλίσει ότι κάθε στοιχείο βρίσκεται ακριβώς σε μία από τις δύο πιθανές θέσεις του.

Δομές Δεδομένων

Ο πίνακας Cuckoo αποτελείται από δύο πίνακες, `table1` και `table2`, ίσου `capacity`. Επιπλέον, χρησιμοποιεί κοινόχρηστη αποθήκευση για τις τιμές:

`value_store`: Αποθηκεύει τις τιμές (indices/items) που σχετίζονται με τα κλειδιά. **`segments`**: Αποθηκεύει τους δείκτες για την πρόσβαση σε πολλαπλές τιμές εντός του `value_store`.

CuckooBucket

Κάθε θέση στους πίνακες περιέχει ένα `CuckooBucket`, το οποίο αντικαθιστά την ανάγκη για `std::optional` μέσω του πεδίου `occupied`. Η δομή αυτή υποστηρίζει την inline βελτιστοποίηση:

```
template<typename Key>
struct CuckooBucket {
    Key key;
    uint32_t first_segment; // Δείκτης για την αλυσίδα στοιχείων στο value_store (αν
count > 1)
    uint32_t last_segment;  // Αποθηκεύει την τιμή (item) αν count = 1 (inline
optimization)
    uint16_t count;         // Πλήθος τιμών
    bool occupied;          // Αντικαθιστά το std::optional
};
```

Πίνακες και Χωρητικότητα

Ο πίνακας διαχειρίζεται δύο εσωτερικούς πίνακες table1 και table2, καθένας με χωρητικότητα capacity.

```
std::vector<CuckooBucket<Key>> table1; // Χρήση CuckooBucket, όχι std::optional
std::vector<CuckooBucket<Key>> table2;
size_t capacity;
```

Συναρτήσεις Κατακερματισμού

Χρησιμοποιούνται δύο ανεξάρτητες συναρτήσεις κατακερματισμού, h1 και h2, για την αντιστοίχιση ενός κλειδιού σε μια θέση στους table1 και table2 αντίστοιχα.

Συνάρτηση h1

Η h1 είναι η τυπική συνάρτηση κατακερματισμού, χρησιμοποιώντας την std::hash.

```
size_t h1(const Key& key) const {
    return key_hasher(key) % capacity;
}
```

Συνάρτηση h2

Η h2 προκύπτει από μια απλή κυκλική μετατόπιση (rotation) του αρχικού hash value, εξασφαλίζοντας μια δεύτερη, ανεξάρτητη διεύθυνση.

```
size_t h2(const Key& key) const {
    size_t h = key_hasher(key);
    // Κυκλική μετατόπιση αριστερά (e.g., κατά 1 bit)
    return ((h << 1) | (h >> (sizeof(size_t) * 8 - 1))) % capacity;
}
```

Μηχανισμός Εισαγωγής (The Kick Process)

Η εισαγωγή ενός νέου στοιχείου γίνεται μέσω της διαδικασίας «εκτόπισης» (kicking) που υλοποιείται στη μέθοδο insert_internal.

Βήματα Εισαγωγής

Η διαδικασία εισαγωγής ακολουθεί τους εξής κανόνες:

- **Έλεγχος Υπάρχοντος:** Πριν την εκτόπιση, ελέγχεται αν το κλειδί υπάρχει ήδη στις δύο πιθανές θέσεις του. Αν ναι, η νέα τιμή **απλώς προστίθεται** στο υπάρχον CuckooBucket (μέσω της insert_duplicate).
- **Ανταλλαγή/Εκτόπιση (Kick):** Εάν η θέση προορισμού είναι κατειλημμένη, το νέο στοιχείο εισάγεται και το υπάρχον στοιχείο εκτοπίζεται. Η ανταλλαγή πραγματοποιείται με

την std::swap(bucket, table[idx]), όπου η μεταβλητή bucket περιέχει πάντα το στοιχείο που είναι «στον αέρα».
- **Μετάβαση:** Το εκτοπισμένο στοιχείο αναζητά την εναλλακτική του θέση στον άλλο πίνακα (από h1 σε h2 και αντίστροφα).

Η διαδικασία αυτή συνεχίζεται έως ότου βρεθεί μια κενή θέση.

Όριο Εκτοπίσεων (MAX_KICKS)

Για να αποφευχθεί ο ατέρμονος βρόχος (cycle) που μπορεί να προκληθεί από την κυκλική εκτόπιση στοιχείων, η υλοποίηση θέτει ένα όριο MAX_KICKS (σταθερά 500). Αν το όριο αυτό ξεπεραστεί, θεωρείται ότι έχει εντοπιστεί ένας κύκλος και απαιτείται ανακατακερματισμός.

Ανακατακερματισμός (Rehash)

Συνθήκη Ανακατακερματισμού

Ο ανακατακερματισμός ενεργοποιείται όταν η εισαγωγή ενός στοιχείου αποτύχει να βρει μια κενή θέση εντός του ορίου MAX_KICKS.

Διαδικασία

Η μέθοδος rehash() εκτελεί τα εξής:
- **Διπλασιασμός Χωρητικότητας:** Το capacity διπλασιάζεται.
- **Αποδοτική Μεταφορά Παλιών Πινάκων:** Οι παλιοί πίνακες μεταφέρονται με **`std::move`** σε τοπικές μεταβλητές, επιτυγχάνοντας $O(1)$ μεταφορά των πόρων (χωρίς αντιγραφή) πριν την εκκαθάριση των κύριων πινάκων.
- **Επαναφορά Πινάκων:** Δημιουργούνται νέοι, κενοί πίνακες table1 και table2 με τη νέα χωρητικότητα.
- **Επανεισαγωγή:** Όλα τα στοιχεία από τους παλιούς πίνακες επανεισάγονται στους νέους πίνακες.

Αναζήτηση (Search)

Η αναζήτηση (find / search) είναι η απλούστερη λειτουργία του Cuckoo Hashing, καθώς το στοιχείο μπορεί να βρίσκεται μόνο σε δύο πιθανές θέσεις, εγγυώμενη $O(1)$ χρόνο αναζήτησης στην χειρότερη περίπτωση: Στη θέση h1(key) του table1. Στη θέση h2(key) του table2. Η συνάρτηση επιστρέφει ένα **`ValueSpan<Key>`**. Εάν βρεθεί το κλειδί: Αν **`count == 1`** (Inline Optimization), η τιμή διαβάζεται απευθείας από το πεδίο `last_segment` του bucket. Αν **`count > 1`**, το span χρησιμοποιεί τους δείκτες `first_segment` και `segments` για να ανακτήσει την αλυσίδα τιμών από το κοινόχρηστο `value_store`.

Αποτελέσματα και Σύγκριση Επιδόσεων

Στην ενότητα αυτή παραθέτουμε τα συμπεράσματα από την εκτέλεση των αλγορίθμων ζεύξης (Join) χρησιμοποιώντας τις τρεις διαφορετικές υλοποιήσεις πινάκων κατακερματισμού. Η σύγκριση εστιάζει στον χρόνο κατασκευής (Build phase), στον χρόνο διερεύνησης (Probe phase) και στη διαχείριση της μνήμης.

Χρόνος Κατασκευής (Build Phase)

- **Robin Hood:** Επέδειξε την πιο σταθερή συμπεριφορά κατά την κατασκευή. Λόγω της γραμμικής προσπέλασης μνήμης (linear probing) και της καλής τοπικότητας (cache locality), η εισαγωγή είναι γρήγορη, παρόλο που απαιτούνται μετακινήσεις (swaps) για την εξισορρόπηση του PSL.
- **Hopscotch:** Η κατασκευή είναι εξαιρετικά γρήγορη όταν ο πίνακας είναι άδειος. Ωστόσο, καθώς ο συντελεστής φόρτου αυξάνεται (>70%), η διαδικασία `relocate` (εύρεση και μετακίνηση κενής θέσης) καθυστερεί ελαφρώς την εισαγωγή σε σχέση με το Robin Hood.
- **Cuckoo:** Η κατασκευή είναι η ταχύτερη για χαμηλούς συντελεστές φόρτου, αλλά καθίσταται ασταθής όταν ο πίνακας γεμίζει (>50%). Οι πολλαπλές εκτοπίσεις (chain reactions of kicks) και η ανάγκη για `rehash` (όταν εντοπίζονται κύκλοι) αυξάνουν το κόστος κατασκευής στη χειρότερη περίπτωση.

Χρόνος Διερεύνησης (Probe Phase)

Αυτό είναι το κρισιμότερο στάδιο για την απόδοση του Join.

- **Hopscotch:** Πέτυχε τους καλύτερους χρόνους σε workloads με υψηλό ποσοστό επιτυχίας (hits). Η χρήση του `bitmask` και της εντολής `__builtin_ctzll` επιτρέπει την εύρεση της εγγραφής ουσιαστικά σε έναν κύκλο ρολογιού αφού φορτωθεί η `cache line`, αποφεύγοντας περιττές συγκρίσεις.
- **Robin Hood:** Η χρήση του **Bloom Filter** (που ενσωματώθηκε στην υλοποίηση) βελτίωσε δραματικά την απόδοση στα «Not Found» cases, απορρίπτοντας άμεσα κλειδιά που δεν υπάρχουν. Η τεχνική Early Exit (βάσει PSL) βοήθησε επίσης στη μείωση του μέσου μήκους αναζήτησης.
- **Cuckoo:** Εγγυάται σταθερό χρόνο αναζήτησης (το πολύ 2 προσπελάσεις μνήμης). Είναι ιδανικό για περιπτώσεις όπου απαιτείται προβλέψιμη απόδοση (latency guarantees), αν και η έλλειψη τοπικότητας (random access στους δύο πίνακες) μπορεί να προκαλέσει περισσότερα Cache Misses σε σχέση με το Hopscotch.

Συμπέρασμα για το Join Pipeline

Για το συγκεκριμένο πρόβλημα του διαγωνισμού SIGMOD, όπου η ταχύτητα ανάγνωσης (probe) είναι κρίσιμη:

1. Το **Hopscotch Hashing** φάνηκε να είναι η πιο αποδοτική επιλογή για datasets που χωράνε άνετα στην L2/L3 Cache, λόγω των bitwise optimizations.

2. Το **Robin Hood** αποτέλεσε την πιο στιβαρή λύση όταν τα δεδομένα ήταν πολλά και ο πίνακας γέμιζε κοντά στο 90%, χάρη στην εξισορρόπηση του PSL.
3. Η υλοποίηση του **Bloom Filter** στο Robin Hood αποδείχθηκε καθοριστική για την απόδοση σε joins με χαμηλή επιλεκτικότητα (low selectivity), όπου πολλά probes αποτυγχάνουν.