Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή

Διδάσκοντες: Ανδρέας Κομνηνός , Σπύρος Σιούτας
Ακαδημαϊκό Έτος: 2025 – 2026
Ημ/νία Παράδοσης: 8/02/2026

Τεχνολογίες Και Αλγόριθμοι Αποκεντρωμένων Δεδομένων

Μάθημα Επιλογής – CEID1411

Εξαμηνιαία Εργασία:

Εφαρμογή Και Πειραματική Αξιολόγηση DHTs Σε Real-Data Sets

Ανδρέας Ζαφειρόπουλος | 1093361 | up1093361@ac.upatras.gr

Δημήτρης Κατσάνος | 1093384 | up1093384@ac.upatras.gr

Σωκράτης Μαντές | 1093421 | up1093421@ac.upatras.gr

Θαλής Χρυσόστομος Τσιωνάς | 1097467 | up1097467@ac.upatras.gr

Περιεχόμενα

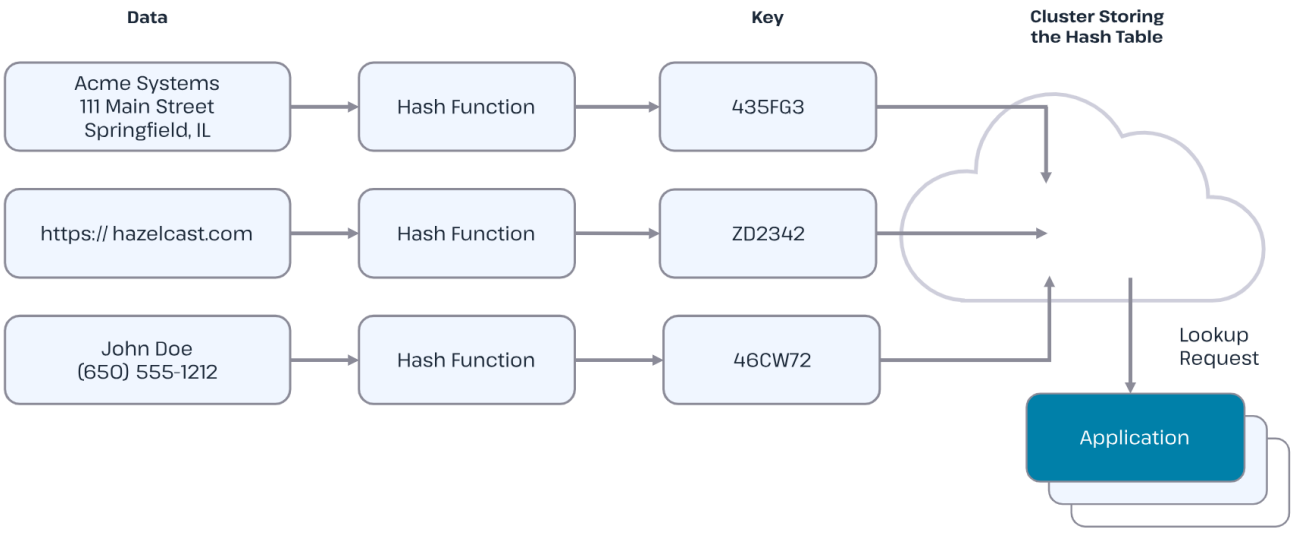
Όλα τα αρχεία μαζί με αποτελέσματα από τα runs βρίσκονται επίσης στο github:

https://github.com/sokratismantes/Decentralized-Data

1. Εισαγωγή

Τα **Distributed Hash Tables (DHT)** είναι μια κατηγορία κατανεμημένων δομών δεδομένων που επιτρέπουν σε ένα δίκτυο πολλών κόμβων να αποθηκεύει και να ανακτά πληροφορία τύπου **key to value** χωρίς να υπάρχει κεντρικός εξυπηρετητής. Η βασική τους λειτουργία είναι ότι κάθε κλειδί (π.χ. ένα όνομα αρχείου, ένας τίτλος ταινίας ή ένα αναγνωριστικό) περνάει από μια συνάρτηση **hash**, η οποία το μετατρέπει σε έναν αριθμό μέσα σε έναν προκαθορισμένο χώρο τιμών (π.χ. 0 έως 2^m−1). Στη συνέχεια, το δίκτυο ορίζει έναν κανόνα αντιστοίχισης που αποφασίζει **ποιος κόμβος είναι υπεύθυνος** για το συγκεκριμένο hash, άρα και για την αποθήκευση της αντίστοιχης τιμής/εγγραφής. Έτσι, όταν κάποιος κόμβος θέλει να κάνει **insert**, **lookup**, **update** ή **delete**, το αίτημα δεν χρειάζεται να ψάξει παντού, αλλά δρομολογείται μέσα στο overlay δίκτυο προς τον κόμβο-ιδιοκτήτη του κλειδιού.

Για να είναι αποδοτικά, τα DHT χρησιμοποιούν ειδικές δομές δρομολόγησης (routing tables) που επιτρέπουν να φτάσουμε στον σωστό κόμβο σε μικρό αριθμό βημάτων, συνήθως **O(log N)** hops, όπου N είναι το πλήθος κόμβων. Αντί να γνωρίζει κάθε κόμβος όλους τους άλλους, κρατάει μόνο κάποιες στρατηγικές επαφές όπως fingers στο Chord ή prefix-based entries στο Pastry, έτσι ώστε κάθε προώθηση να μειώνει σημαντικά την απόσταση από τον στόχο. Ένα σημαντικό πλεονέκτημα των DHT είναι ότι λειτουργούν καλά σε περιβάλλοντα **peer-to-peer (p2p)**, όπου οι κόμβοι μπαίνουν και βγαίνουν συχνά **(churn phenomenon).** Για αυτό υπάρχουν διαδικασίες **node join** και **node leave**, οι οποίες ενημερώνουν τη δομή δρομολόγησης και ανακατανέμουν τα κλειδιά ώστε το σύστημα να παραμένει σωστό και διαθέσιμο.



Εικόνα 1: Μετατροπή δεδομένων σε κλειδιά μέσω hash function και αποθήκευση/αναζήτηση σε κατανεμημένο cluster (hash table) από την εφαρμογή.

Στην δική μας περίπτωση, ο στόχος είναι να **αξιοποιήσουμε δύο διαφορετικά DHTs (Chord και Pastry)** ώστε να υλοποιήσουμε ένα αποκεντρωμένο p2p overlay, δηλαδή ένα δίκτυο που θα αποθηκεύει ζεύγη **(key, value)**, όπου το value είναι ένα σύνολο γνωρισμάτων (attributes)

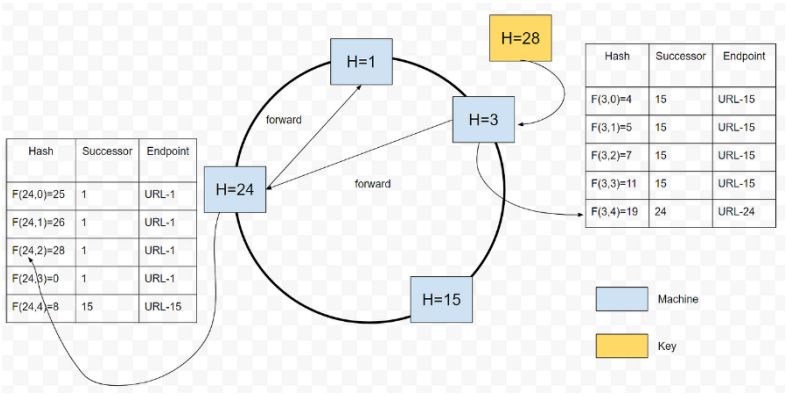
για κάθε κλειδί, και να κάνουμε **πειραματική αξιολόγηση** των βασικών λειτουργιών μετρώντας τον αριθμό των **hops** που απαιτούνται, δηλαδή πόσες προωθήσεις από κόμβο σε κόμβο κάνει ένα αίτημα μέχρι να φτάσει στον σωστό υπεύθυνο κόμβο.Στα DHT συνήθως τα hops αυξάνονται περίπου **λογαριθμικά** με τον αριθμό των κόμβων **N**.

Πρακτικά, επιλέγουμε και χρησιμοποιούμε το dataset data_movies_clean με περίπου 946 χιλιάδες εγγραφές ταινιών, όπου ο τίτλος ταινίας **title** χρησιμοποιείται ως key και τα υπόλοιπα πεδία ως attributes και το μοιράζουμε πάνω σε ένα σύνολο από κόμβους ενός DHT. Κάθε ταινία έχει τίτλο, και ο τίτλος γίνεται hash σε έναν χώρο κλειδιών μεγέθους 2^m, ενώ στη συνέχεια το DHT επιλέγει ποιος κόμβος είναι υπεύθυνος για το συγκεκριμένο hash. Στο Chord ο υπεύθυνος είναι ο successor του key στον δακτύλιο, ενώ στο Pastry η δρομολόγηση βασίζεται σε prefix-matching του nodeID, αλλά και στις δύο περιπτώσεις το κρίσιμο είναι ότι δεν ψάχνουμε γραμμικά σε όλους τους κόμβους, αλλά δρομολογούμε το αίτημα προς τον σωστό κόμβο μέσα από λίγα hops, συνήθως λογαριθμικά ως προς N.

Σκοπός μας είναι να αξιολογήσουμε τις θεμελιώδεις λειτουργίες **Build, Insert key, Delete key, Update key, Lookup(key), Node Join, Node Leave**, άρα αξιοποιούμε τα DHT όχι μόνο για να αποθηκεύσουμε και να ανακτήσουμε δεδομένα, αλλά και για να μετρήσουμε πόσο κοστίζουν οι βασικές λειτουργίες όταν το δίκτυο μεγαλώνει ή όταν οι κόμβοι μπαίνουν/βγαίνουν. Για την πειραματική αξιολόγηση θα εκτελέσουμε σειρές από αυτές τις λειτουργίες για διαφορετικά μεγέθη δικτύου (π.χ. N=16,32,64), με τυχαία σημεία εκκίνησης, και θα **μετρήσουμε hops** ανά operation και ανά πρωτόκολλο. Στη συνέχεια θα συγκεντρώσουμε όλα τα μετρικά αποτελέσματα και θα φτιάξουμε γραφήματα που συγκρίνουν Chord vs Pastry για κάθε query, ώστε να αποτυπώνεται καθαρά πώς κλιμακώνεται το κόστος δρομολόγησης όταν μεγαλώνει το N και πώς επηρεάζεται από churn (joins/leaves).

2. Περιγραφή Των DHTs Που Υλοποιούνται

➢ Chord



Το Chord είναι ένα πρωτόκολλο κατανεμημένου πίνακα κατακερματισμού (DHT) που βασίζεται στον συνεπή κατακερματισμό (consistent hashing) για να κατανέμει ομοιόμορφα τόσο τους κόμβους όσο και τα κλειδιά σε έναν κοινό χώρο αναγνωριστικών. Κάθε κόμβος και κάθε κλειδί παίρνει έναν αναγνωριστικό m-bit, με βασική συνάρτηση κατακερματισμού τον SHA-1. Η ομοιόμορφη κατανομή μειώνει στο ελάχιστο την πιθανότητα σύγκρουσης και επιτρέπει σε κόμβους να μπαίνουν ή να αποχωρούν από το δίκτυο χωρίς να προκαλούν σημαντική διαταραχή.

Οι κόμβοι και τα κλειδιά οργανώνονται σε έναν κύκλο αναγνωριστικών με τιμές από 0 έως 2^m − 1. Κάθε κόμβος διατηρεί έναν διάδοχο (successor) και έναν προκάτοχο (predecessor):

ο διάδοχος είναι ο επόμενος κόμβος δεξιόστροφα στον κύκλο, ενώ ο προκάτοχος είναι ο αμέσως προηγούμενος αριστερόστροφα.

➢ Πως Υλοποιήθηκε Το Chord

Το Chord που υλοποιήσαμε βασίζεται στην κλασική ιδέα ενός δαχτυλιδιού από κόμβους, όπου τόσο οι κόμβοι όσο και τα δεδομένα βρίσκονται στον ίδιο κυκλικό χώρο. Ξεκινάμε ορίζοντας έναν χώρο τιμών μεγέθους 2^m(στα δικό μας runs χρησιμοποιήσαμε συνήθως m=40). Κάθε τιμή (π.χ. τίτλος ταινίας) μετατρέπεται σε αριθμητικό key με τη chord_hash και κάνουμε SHA-1 πάνω στο string και παίρνουμε modulo 2^m. Έτσι, μπορούμε μετά να πούμε με ακρίβεια ποιος κόμβος είναι υπεύθυνος για ποια κλειδιά.

```python
from b_tree import BPlusTree
import hashlib


def chord_hash(value, m=40):
    h = hashlib.sha1(str(value).encode()).hexdigest()
    num = int(h, 16) % (2**m)
    return num
```

Κάθε κόμβος (ChordNode) έχει το node_id, pointers σε successor και predecessor, και ένα finger table μεγέθους m. Το finger table είναι αυτό που δίνει την δρομολόγηση, καθώς αντί να προχωράμε βήμα-βήμα στον επόμενο κόμβο, μπορούμε να κάνουμε άλματα προς κόμβους που είναι πιο κοντά στο key. Ένα ακόμη πρακτικό στοιχείο της υλοποίησης είναι ότι κάθε κόμβος κρατάει και ένα B+ tree (BPlusTree) για την τοπική αποθήκευση των records. Έτσι, αφού ένα key φτάσει στον σωστό κόμβο, η αναζήτηση/εισαγωγή/διαγραφή στο εσωτερικό του κόμβου γίνεται αποδοτικά.

```python
class ChordNode:
    def __init__(self, node_id, m, btree_size=32):
        self.node_id = node_id
        self.m = m
        self.btree = BPlusTree(btree_size)

        self.successor = None
        self.predecessor = None

        # finger table size m
        self.finger = [None] * m

    def __repr__(self):
        return f"Node({self.node_id})"
```

Χρησιμοποιούμε την κλάση ChordRing. Αυτή κρατάει τη λίστα nodes, το m και οτιδήποτε αφορά τη λειτουργία του overlay. Εκεί υπάρχουν πρώτα βοηθητικά methods όπως το _in_interval, που λύνει το κλασικό πρόβλημα των κυκλικών διαστημάτων $(start, end]$στο ring, και το _update_links, που μετά από κάθε αλλαγή στην ταξινομημένη λίστα κόμβων ενημερώνει σωστά successor και predecessor για όλους.

```
28  class ChordRing:
29      def __init__(self, m=160, btree_size=32):
30          self.m = m
31          self.btree_size = btree_size
32          self.nodes = []
```

```
34      # ------------------------ helpers ------------------------
35      def _in_interval(self, key, start, end):
36          """(start, end] in circular space."""
37          if start < end:
38              return start < key <= end
39          else:
40              return key > start or key <= end
41
42      def _update_links(self):
43          n = len(self.nodes)
44          for i, node in enumerate(self.nodes):
45              node.successor = self.nodes[(i + 1) % n]
46              node.predecessor = self.nodes[(i - 1) % n]
```

Η δρομολόγηση υλοποιείται στη find_successor. Η λογική είναι ότι ξεκινάμε από έναν κόμβο (default ο πρώτος), και μέχρι να βρουμε τον κόμβο που καλύπτει το key, προχωράμε. Πρώτα χειριζόμαστε το exact match (if key == curr.node_id) ώστε να επιστρέψουμε τον ίδιο τον κόμβο. Έπειτα ελέγχουμε αν το key πέφτει στο $(curr, succ]$και αν ναι επιστρέφουμε τον successor. Αν όχι, χρησιμοποιούμε closest_preceding_finger, το οποίο σαρώνει το finger table από το μεγαλύτερο i προς το 0 και επιλέγει τον καλύτερο προηγούμενο κόμβο πριν από το key, για να κάνουμε άλματα. Για ασφάλεια υπάρχει max_steps, και αν κάτι πάει στραβά (π.χ. κάποια ασυνέπεια στις δομές), γίνεται fallback σε find_successor_linear, ώστε να μην κολλήσει ποτέ η εκτέλεση. Παράλληλα, το method επιστρέφει και hops, που είναι βασικό για τις μετρήσεις μας στην συνέχεια.

```python
    # ------------------------ routing ------------------------
    def find_successor(self, key, start_node=None):
        """
        Find the successor node for a given key using Chord routing.
        Returns (node, hops).
        """
        if not self.nodes:
            return None, 0

        start_node = start_node or self.nodes[0]
        curr = start_node
        hops = 0

        # safety bound to avoid infinite loops
        max_steps = max(4, len(self.nodes) * 4)
```

```python
        while True:
            if key == curr.node_id:
                return curr, hops

            # normal success condition
            succ = curr.successor
            if self._in_interval(key, curr.node_id, succ.node_id):
                return succ, hops + 1  # count the final hop to successor

            nxt = self.closest_preceding_finger(curr, key)
            if nxt is None or nxt is curr:
                nxt = succ

            curr = nxt
            hops += 1

            if hops > max_steps:
                # fallback to linear scan
                return self.find_successor_linear(key), hops
```

```python
    def find_successor_linear(self, key):
        """Linear successor search."""
        for node in self.nodes:
            if key <= node.node_id:
                return node
        return self.nodes[0]
```

```python
    def closest_preceding_finger(self, node, key):
        for i in reversed(range(self.m)):
            finger = node.finger[i]
            if finger and self._in_interval(finger.node_id, node.node_id, key):
                return finger
        return node
```

Ακριβώς πάνω σε αυτό το routing χτίσαμε τα operations:

Insert: Το insert(key_int, record) υποθέτει ότι έχουμε ήδη hashed key. Καλεί find_successor (με προαιρετικό start node) και όταν βρει τον υπεύθυνο κόμβο, κάνει node.btree.insert(record, key_int). Επιστρέφει τα hops για να μετράμε κόστος overlay. Για ευκολία και ασφάλεια υπάρχει και insert_title(title, record) που κάνει hash εσωτερικά και μετά καλεί το insert.

```
110     # ------------------------ data operations ------------------------
111     def insert(self, key_int, record, start_node=None):
112         """Insert record under an already-hashed key."""
113         node, hops = self.find_successor(key_int, start_node=start_node)
114         node.btree.insert(record, key_int)
115         return hops
116 
117     def insert_title(self, movie_title, record, start_node=None):
118         """Insert by title (hash inside for safety/consistency)."""
119         key_int = chord_hash(movie_title, self.m)
120         return self.insert(key_int, record, start_node=start_node)
```

Lookup: Το lookup(title) κάνει chord_hash(title) και βρίσκει successor με find_successor. Μετά ψάχνει το key στο B+ tree του κόμβου με search_key. Επιστρέφει (records, hops), ώστε να βλέπουμε τόσο τι βρέθηκε όσο και πόσο ακριβά δρομολογήθηκε.

```
122     def lookup(self, movie_title, start_node=None):
123         key_int = chord_hash(movie_title, self.m)
124         node, hops = self.find_successor(key_int, start_node=start_node)
125         records = node.btree.search_key(key_int)
126         return records, hops
```

Delete: Υπάρχουν δύο παραλλαγές, η πρώτη είναι delete_key(key_int) αν έχουμε ήδη hash και η δεύτερη delete_title(title) αν θέλουμε να δουλεύουμε με τίτλο). Και στις δύο περιπτώσεις η λογική είναι ίδια, μεταβαίνουμε από την find_successor στο node.btree.delete(key_int), εκτελείται διαγραφή και επιστρέφουν τα hops.

```
128     def delete_key(self, key_int, start_node=None):
129         node, hops = self.find_successor(key_int, start_node=start_node)
130         node.btree.delete(key_int)
131         return hops
132 
133     def delete_title(self, movie_title, start_node=None):
134         key_int = chord_hash(movie_title, self.m)
135         return self.delete_key(key_int, start_node=start_node)
```

Update: Το update_movie_field(title, field, new_value) κάνει hash τον τίτλο, βρίσκει τον κόμβο που κρατάει το key, διαβάζει το record με search_key και, αν υπάρχει, αλλάζει το αντίστοιχο πεδίο (π.χ. popularity). Επιστρέφει (ok, hops)και συγκεκριμένα ok=False αν δεν βρέθηκε record, αλλιώς True.

```python
    def update_movie_field(self, title, field, new_value, start_node=None):
        key_int = chord_hash(title, self.m)
        node, hops = self.find_successor(key_int, start_node=start_node)
        records = node.btree.search_key(key_int)
        if not records:
            return False, hops
        record = records[0]
        record[field] = new_value
        return True, hops
```

Join: Το join_node(node_id) προσθέτει έναν νέο κόμβο στο ring. Πρώτα, αν το ring δεν είναι άδειο μετράει κόστος routing για το που θα έμπαινε ο κόμβος καλώντας find_successor(node_id). Μετά δημιουργεί ChordNode, τον βάζει στη λίστα, κάνει sort, και ενημερώνει successor/predecessor με _update_links. Το κρίσιμο μέρος είναι η ανακατανομή, όπου η _redistribute_keys(new_node) μεταφέρει από τον successor προς τον νέο κόμβο όλα τα keys που πλέον ανήκουν στο διάστημα $(pred, new]$. Μετράμε και πόσα records μετακινήθηκαν (moved_cnt) και, προαιρετικά, hops που καταναλώθηκαν κατά τη μεταφορά. Στο τέλος καλούμε fix_all_fingers για να ενημερωθούν τα finger tables.

```python
    def join_node(self, node_id, start_node=None):
        """
        Node join.
        Returns:
        - routing hops to locate successor of node_id
        - moved_cnt
        - migrate_hops: routing hops used to move keys (sum over moved keys)
        """
        locate_hops = 0

        # routing cost to find where the node would attach before insertion
        if self.nodes:
            _, locate_hops = self.find_successor(node_id, start_node=start_node or self.nodes[0])

        new_node = ChordNode(node_id, self.m, btree_size=self.btree_size)
        self.nodes.append(new_node)
        self.nodes.sort(key=lambda n: n.node_id)
        self._update_links()

        migrate_hops = 0
        moved_cnt = 0
        if len(self.nodes) > 1:
            migrate_hops, moved_cnt = self._redistribute_keys(new_node)

        self.fix_all_fingers()

        return new_node, locate_hops, moved_cnt, migrate_hops
```

```python
    def _redistribute_keys(self, new_node):
        """
        Move keys from new_node.successor to new_node when they fall in (pred, new_node].
        Returns (migrate_hops, moved_count).
        """
        pred = new_node.predecessor
        succ = new_node.successor

        items = succ.btree.get_all_items()
        total_hops = 0
        moved = 0

        for key_int, record in items:
            if self._in_interval(key_int, pred.node_id, new_node.node_id):
                # count routing cost
                _, hops = self.find_successor(key_int, start_node=succ)
                total_hops += hops
                new_node.btree.insert(record, key_int)
                succ.btree.delete(key_int)
                moved += 1

        return total_hops, moved
```

Leave: Το leave_node(node_id) αφαιρεί κάποιο κόμβο. Αν ο κόμβος υπάρχει και δεν είναι ο μοναδικός, μετράμε routing hops για το leave event με find_successor πάνω στο node_id. Έπειτα μεταφέρουμε όλα τα δεδομένα του κόμβου στον successor για να μη χαθούν και μετράμε πόσα records μετακινήθηκαν. Αφαιρούμε τον κόμβο από τη λίστα, ξαναφτιάχνουμε links και finger tables. Στη δική μας σχεδίαση δεν χρεώνουμε hops ως ποινή για τη μεταφορά δεδομένων στο leave, δηλαδή τα επιστρέφουμε ως 0, για να ξεχωρίζει καθαρά το κόστος του leave event.

```python
    def leave_node(self, node_id, start_node=None):
        """
        Node leave.
        Returns:
        - routing_hops (event): overlay routing cost for the LEAVE request
        - moved_cnt
        """
        node = next((n for n in self.nodes if n.node_id == node_id), None)
        if not node:
            return False, 0, 0

        if len(self.nodes) == 1:
            self.nodes.remove(node)
            return True, 0, 0

        routing_hops = 0
        if self.nodes:
            start = start_node or self.nodes[0]
            _, routing_hops = self.find_successor(node_id, start_node=start)

        succ = node.successor
```

```python
def main():
    project_root = Path(".").resolve()
    data_path = project_root / "data_movies_clean.csv"

    print(f"Loading dataset from: {data_path}")
    df = load_and_preprocess_csv(str(data_path), max_rows=946_460, seed=1)
    print(f"Loaded {len(df)} rows.\n")

    # params
    num_nodes = 32
    updates_n = 2_000
    deletes_n = 2_000
    joins_n = 10
    leaves_n = 10

    ring = ChordRing(m=40)

    # -------- initial nodes with random unique IDs --------
    N0 = num_nodes
    space = 2 ** ring.m
    random.seed(42)
    node_ids = random.sample(range(space), k=num_nodes)
```

```python
            # move data for correctness
            moved = 0
            items = node.btree.get_all_items()
            for key_int, record in items:
                succ.btree.insert(record, key_int)
                moved += 1

            self.nodes.remove(node)
            self._update_links()
            self.fix_all_fingers()

            return True, routing_hops, moved
```

Για να τρέξουμε το Chord, ξεκινήσαμε από ένα πλήρες σενάριο εκτέλεσης που προσομοιώνει ένα πραγματικό DHT, όπου πρώτα στήνουμε το δίκτυο, μετά φορτώνουμε όλα τα δεδομένα, και έπειτα κάνουμε αλλαγές στο membership και στις λειτουργίες πάνω στα keys. Στο main_chord.py φορτώνουμε το CSV (data_movies_clean.csv) με τη load_and_preprocess_csv, κρατώντας seed ώστε το dataset να είναι ίδιο σε κάθε run. Έπειτα δημιουργούμε ένα ChordRing(m=40) και ορίζουμε πόσους κόμβους θέλουμε αρχικά , π.χ. στο δικό μας παράδειγμα επιλέξαμε 32. Για να έχουν οι κόμβοι μοναδικά IDs στον χώρο 2^{40}, χρησιμοποιούμε random.sample με σταθερό seed που επιλέξαμε το 42.

```python
def main():
    project_root = Path(".").resolve()
    data_path = project_root / "data_movies_clean.csv"

    print(f"Loading dataset from: {data_path}")
    df = load_and_preprocess_csv(str(data_path), max_rows=946_460, seed=1)
    print(f"Loaded {len(df)} rows.\n")

    # params
    num_nodes = 32
    updates_n = 2_000
    deletes_n = 2_000
    joins_n = 10
    leaves_n = 10

    ring = ChordRing(m=40)

    # -------- initial nodes with random unique IDs --------
    N0 = num_nodes
    space = 2 ** ring.m
    random.seed(42)
    node_ids = random.sample(range(space), k=num_nodes)
```

Αφού έχουμε τα node IDs, κάνουμε join έναν-έναν τους κόμβους με ring.join_node(nid) και αμέσως μετά εισάγουμε όλες τις εγγραφές του dataset. Η εισαγωγή γίνεται με hash του τίτλου (chord_hash(title, m=ring.m)) και μετά ring.insert(key, row.to_dict()), ώστε κάθε record να πάει στον σωστό successor κόμβο.

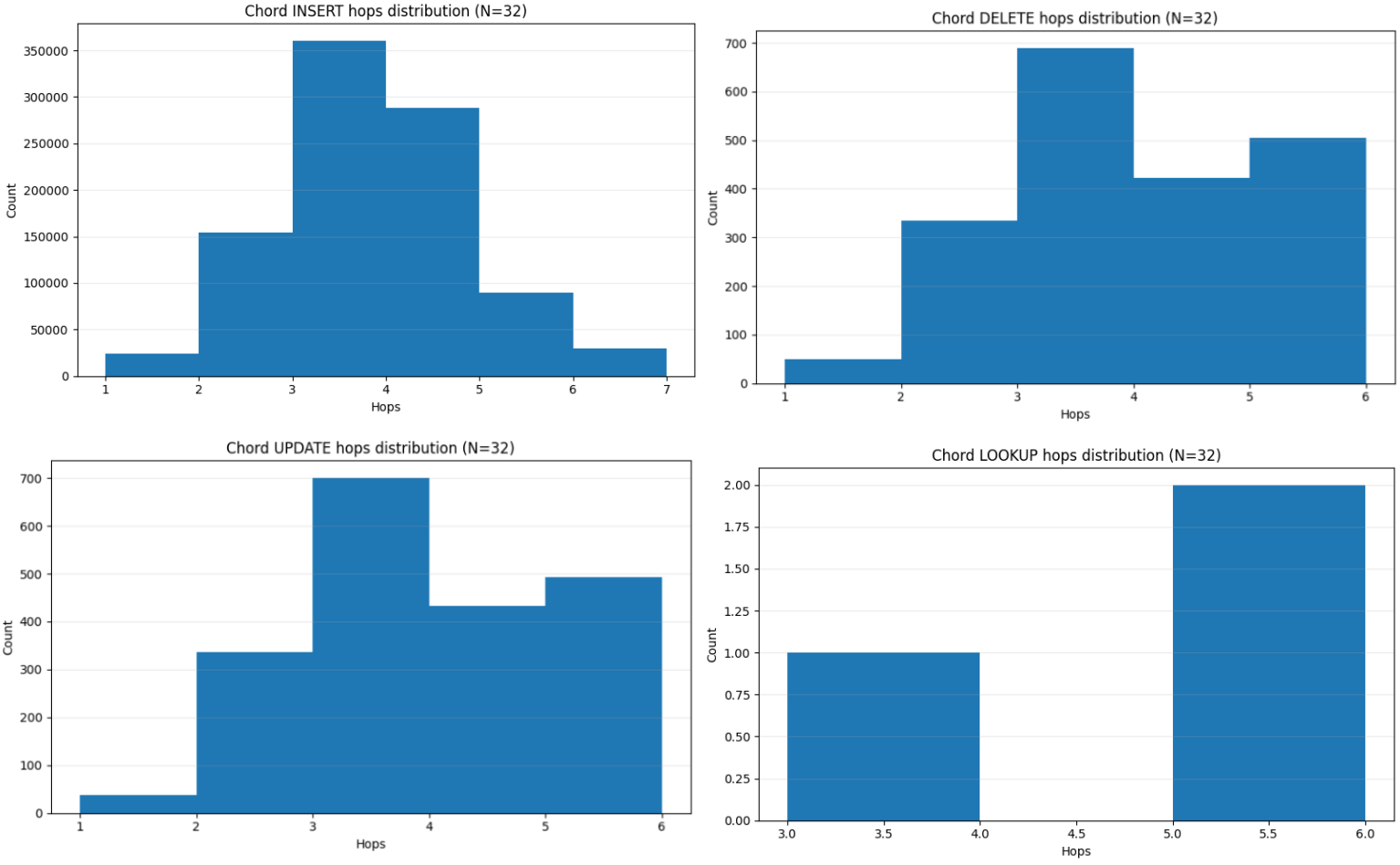
```python
    print("=== Joining initial Chord nodes ===")
    join_hops_list = []
    moved_list = []
    for nid in node_ids:
        ret = ring.join_node(nid)
        if isinstance(ret, tuple) and len(ret) >= 3:
            hops, moved = ret[1], ret[2]
        elif isinstance(ret, tuple) and len(ret) == 2:
            hops, moved = ret[1], 0
        else:
            hops, moved = int(ret), 0
        join_hops_list.append(int(hops))
        moved_list.append(int(moved) if moved is not None else 0)
```

Στη συνέχεια εκτελούμε 10 επιπλέον joins με νέα τυχαία IDs που δεν υπάρχουν ήδη και 10 leaves επιλέγοντας τυχαίους κόμβους, αποφεύγοντας να πέσει το δίκτυο κάτω από 2 κόμβους.

Μετά περνάμε στα operations στα δεδομένα. Συγκεκριμένα, επιλέξαμε να κάνουμε 2000 updates (π.χ. ενημέρωση του πεδίου popularity) και 2000 deletes σε τυχαίους τίτλους από το dataset. Τέλος, κάνουμε lookups για Κ τίτλους, όπου ο χρήστης μπορεί να δώσει το πλήθος των ταινιών Κ αλλά και συγκεκριμένους τίτλους ή να αφήσει κενό για τυχαία επιλογή, και το script κάνει αναζητήσεις, ακόμη και με threads για να δείξει παράλληλα lookups. Στο τέλος αποθηκεύονται και βασικά plots για να έχουμε μια γρήγορη εικόνα των αποτελεσμάτων.

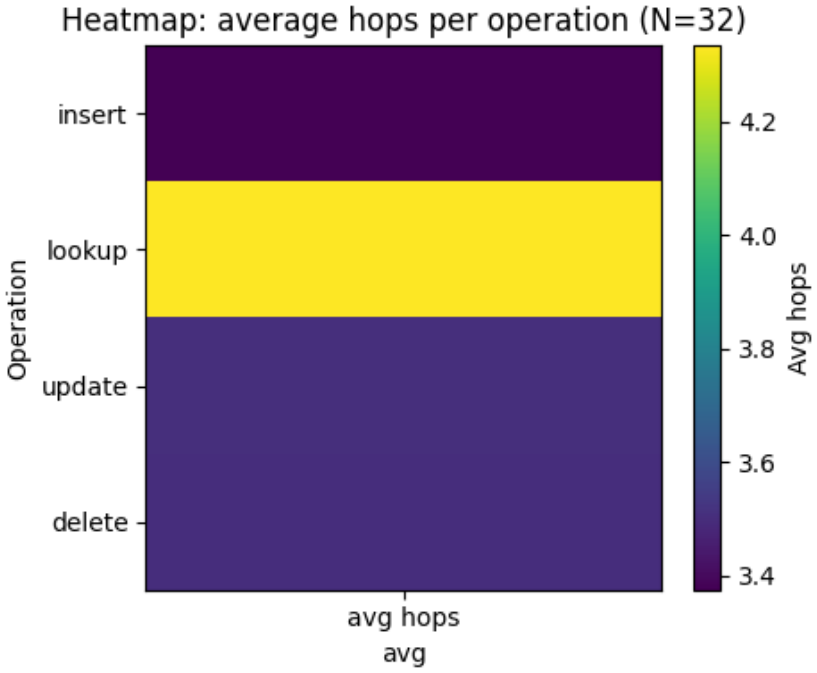
Στην έξοδο εκτυπώσαμε κυρίως στατιστικά για **hops** και για **μετακινήσεις εγγραφών**. Για κάθε κατηγορία (initial join, insert, dynamic join, dynamic leave, update, delete, lookup) κρατήσαμε λίστες με τις τιμές των hops και τυπώσαμε n, avg, median, p95, min, max. Τα hops δείχνουν πόσους κόμβους διασχίζει μια λειτουργία στο overlay μέχρι να φτάσει στον σωστό successor, άρα είναι βασικός δείκτης απόδοσης του Chord routing που πρέπει να συλλέξουμε.

Παράλληλα, στα join/leave μετρήσαμε **moved records**, δηλαδή πόσα keys χρειάστηκε να μεταφερθούν, γιατί αυτό είναι το πρακτικό κόστος συνέπειας όταν αλλάζει το δίκτυο. Επειδή το Chord πρέπει να ξαναμοιράσει δεδομένα για να παραμένει σωστή η αντιστοίχιση key με κόμβος Για τα lookups εκτυπώσαμε και το popularity που βρέθηκε (ή NOT FOUND) μαζί με τα hops, ώστε να φαίνεται ότι το routing δεν είναι απλώς θεωρητικό αλλά οδηγεί και σε πραγματική ανάκτηση δεδομένων.

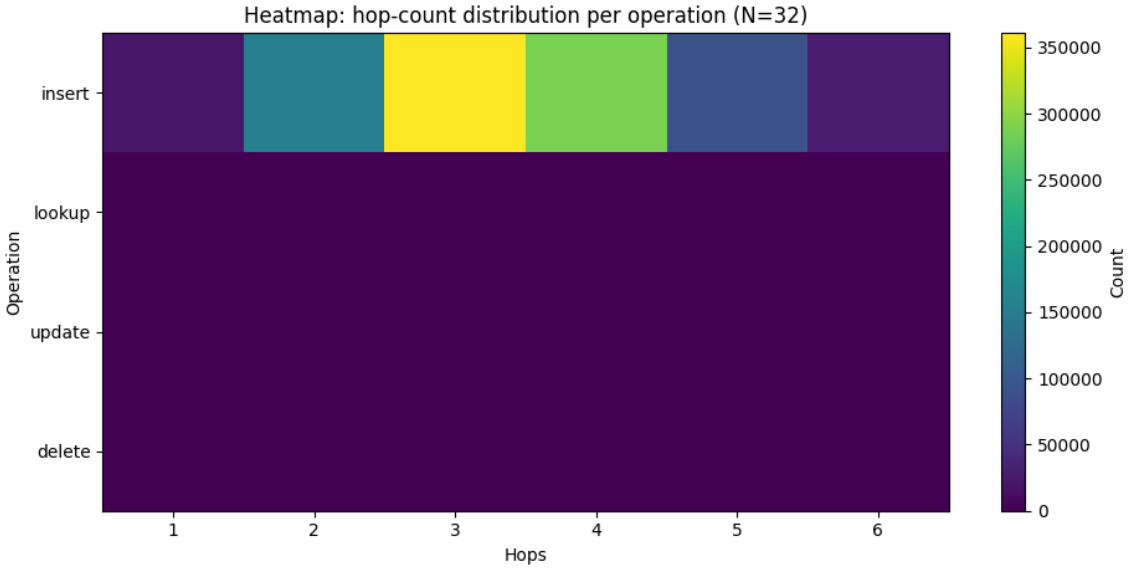


Εικόνα 2: Histograms (hops) για Chord: Κατανομή των routing hops ανά operation — δείχνει πόσο συχνά εμφανίζεται κάθε αριθμός hops ανά operation.

Τα histograms του Chord ουσιαστικά μας δείχνουν την κατανομή του κόστους δρομολόγησης (hops) για κάθε λειτουργία. Οι περισσότερες κλήσεις συγκεντρώνονται γύρω στα 3–5 hops, ενώ παράλληλα υπάρχει ένα μικρό ποσοστό πολύ εύκολων περιπτώσεων (1–2 hops) όταν ξεκινάμε τυχαία κοντά στον σωστό κόμβο, και μια μικρή ουρά προς 6–7 hops όταν η διαδρομή δεν βρίσκει αμέσως ιδανικό finger και χρειάζεται παραπάνω βήματα. Επίσης φαίνεται ξεκάθαρα ότι κάποια histograms (π.χ. insert) είναι πολύ πιο γεμάτα, επειδή έχουν τεράστιο πλήθος δειγμάτων από το bulk loading του dataset, ενώ το lookup έχει λίγα δείγματα και συγκεκριμένα μόνο τις K αναζητήσεις, τις οποίες στο παράδειγμά μας ορίσαμε ως 3 και άρα η κατανομή του είναι πιο αραιή και θορυβώδης οπτικά.



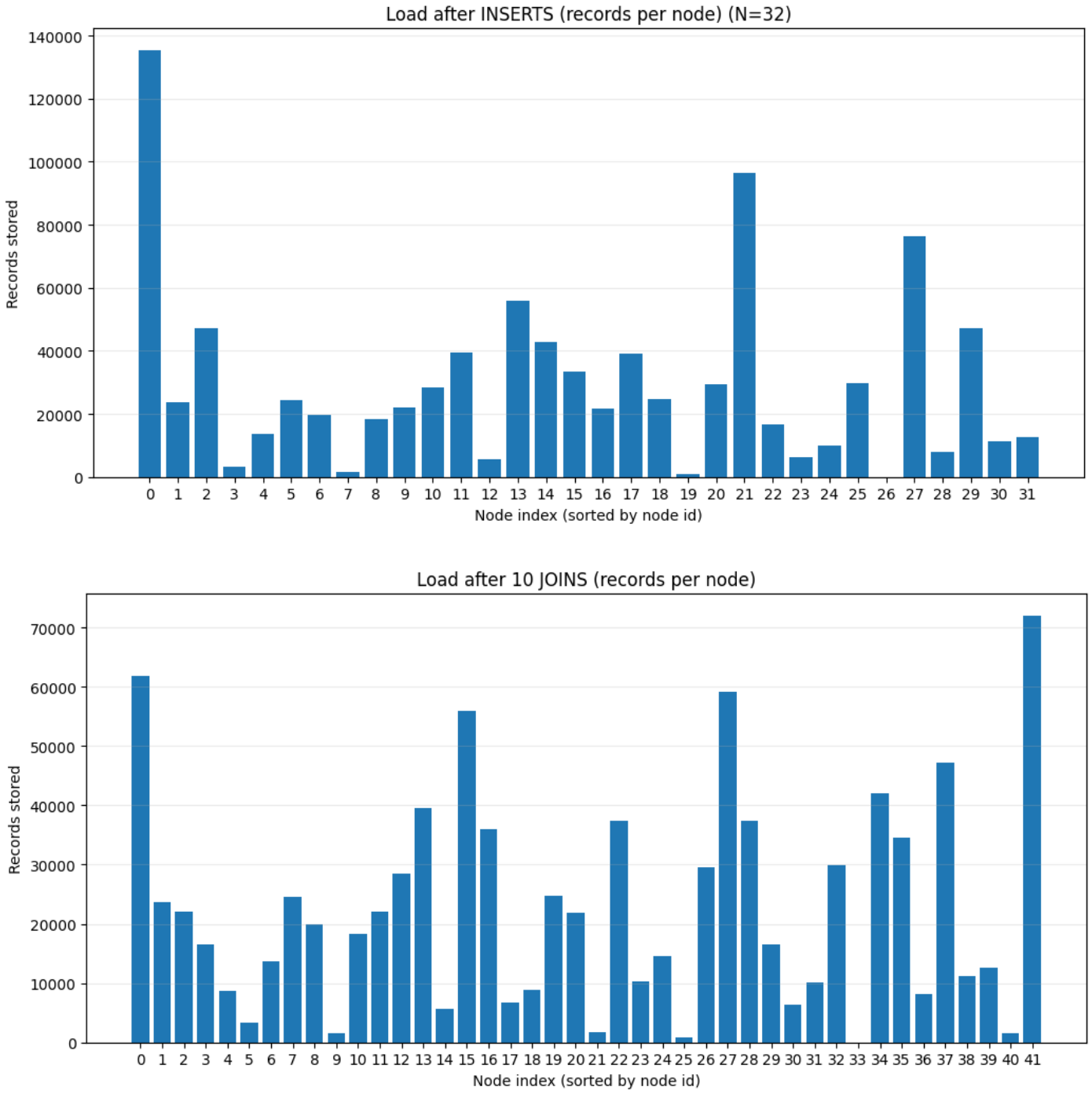
Εικόνα 3: Heatmap (avg hops)για Chord: Μέσος αριθμός hops ανά operation — συνοπτική σύγκριση του μέσου κόστους δρομολόγησης



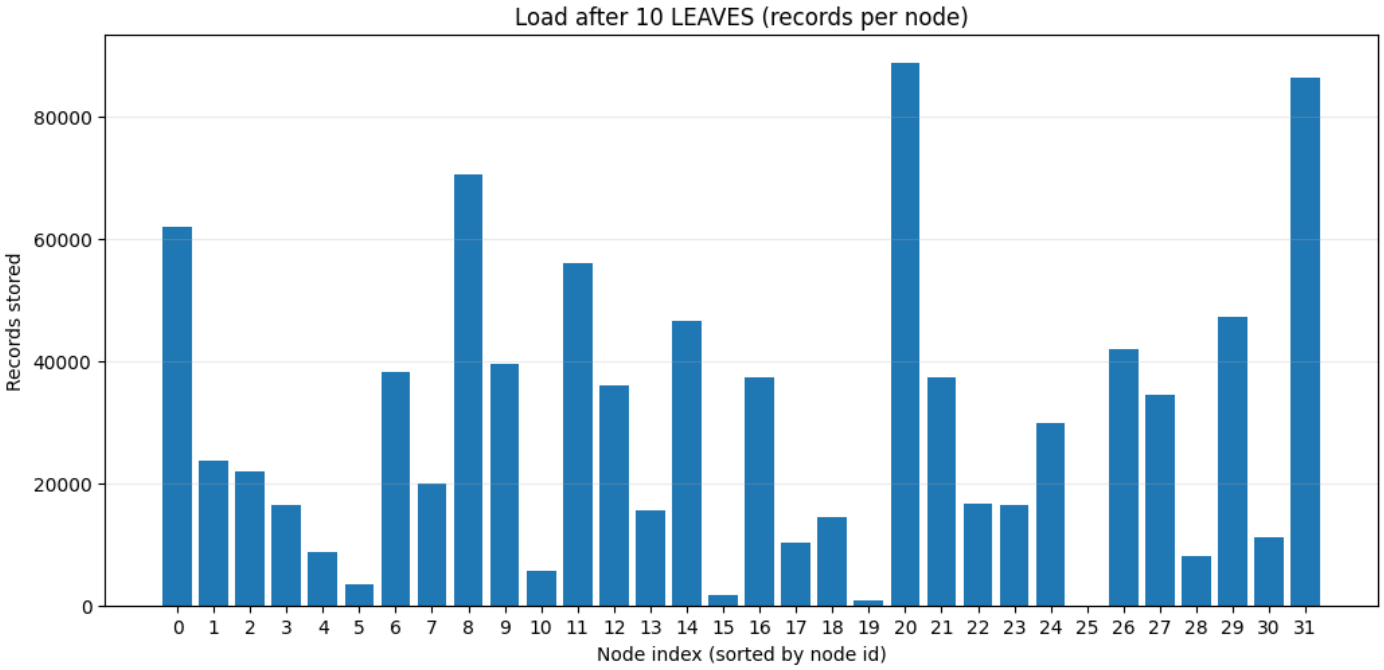
Εικόνα 4: Heatmap (hop-count distribution) για Chord: Πυκνότητα/πλήθος μετρήσεων hops ανά operation — δείχνει πού συγκεντρώνονται τα περισσότερα δείγματα

Στα **heatmaps** του Chord βλέπουμε τα **average hops**. Η εικόνα που έχουμε δείχνει ότι το σύστημα κινείται γενικά σε σταθερό κόστος, καθώς τα περισσότερα operations

(insert/update/delete) μένουν γύρω στα 3–4 hops, ενώ το lookup ανεβαίνει λίγο, γιατί εκεί πληρώνουμε με κόστος καθαρά τη διαδρομή μέχρι τον υπεύθυνο κόμβο. Το δεύτερο heatmap, με τις **μετρήσεις/πυκνότητα** μας αποκαλύπτει πού υπάρχει συγκεντρωμένος ο όγκος των δεδομένων. Συγκεκριμένα, στο insert είναι πολύ πιο έντονο επειδή έγιναν πάρα πολλές εισαγωγές, καθώς εισαγάγαμε όλο το dataset με περίπου 950 χιλιάδες γραμμές, άρα η κατανομή εκεί είναι πιο αξιόπιστη, ενώ στα άλλα operations είναι πιο αραιό γιατί έχουμε αρκετά λιγότερα δείγματα. Συνολικά, το πρώτο heatmap μας οπτικοποιεί πόσο κοστίζει κατά μέσο όρο ένα operation, και το δεύτερο μας οπτικοποίησε πόσο πολύ το είδαμε αυτό και στην πράξη, δηλαδή πόσα δείγματα το στηρίζουν.

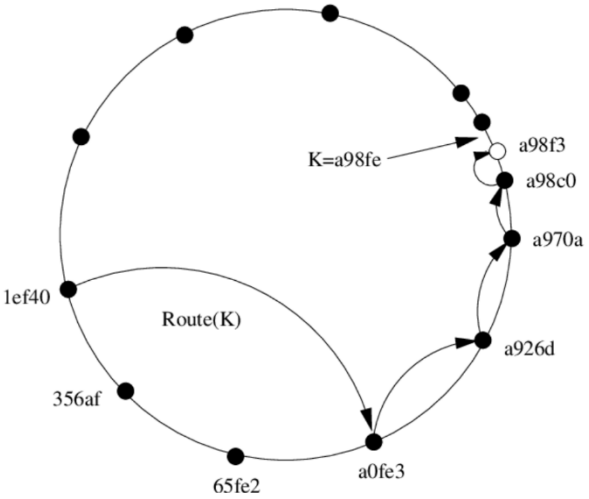


Εικόνα 5: Loads (records per node) για Chord: Κατανομή records ανά κόμβο μετά από inserts/joins/leaves — απεικονίζει πώς μοιράζεται το φορτίο στο δίκτυο



Στα **τρία load plots** ουσιαστικά βλέπουμε πώς μοιράζονται τα records στους κόμβους καθώς το δίκτυο αλλάζει. **Μετά τα inserts**, η κατανομή δεν είναι τέλεια καθώς κάποιοι κόμβοι έχουν πολύ περισσότερα keys, γιατί στο Chord ο κάθε κόμβος παίρνει ό,τι πέφτει στο διάστημα ανάμεσα σε αυτόν και τον predecessor του. Οπότε αν το διάστημα είναι μεγάλο, παίρνει και πολύ φορτίο. **Μετά τα joins**, η εικόνα συνήθως απλώνει εφόσον εισέρχονται νέοι κόμβοι στο ring, κόβουν μεγάλα διαστήματα σε μικρότερα και τραβάνε μέρος των keys, οπότε πέφτουν κάποια μεγάλα peaks και το load κατανέμεται σε περισσότερους κόμβους. **Μετά τα leaves**, συμβαίνει το αντίστροφο. Οι κόμβοι φεύγουν και τα δεδομένα τους μεταφέρονται στον successor, άρα βλέπουμε ξανά πιο έντονες κορυφές σε συγκεκριμένους κόμβους. Με απλά λόγια, τα loads απέδειξαν ξεκάθαρα ότι τα joins τείνουν να εξομαλύνουν, ενώ τα leaves μπορούν να υπερφορτώσουν προσωρινά κάποιους κόμβους.

- **Pastry**



Το Pastry είναι ένα πρωτόκολλο κατανεμημένου πίνακα κατακερματισμού (DHT), παρόμοιο με το Chord, που ξεχωρίζει κυρίως λόγω του overlay δικτύου δρομολόγησης που «χτίζει» πάνω από τη βασική ιδέα του DHT. Τα ζεύγη κλειδί–τιμή αποθηκεύονται σε ένα πλεονάζον (redundant) peer-to-peer δίκτυο, ώστε να μειώνεται η πιθανότητα απώλειας δεδομένων όταν κόμβοι αποχωρούν ή αποτυγχάνουν. Η είσοδος ενός νέου κόμβου ξεκινά με bootstrap, δίνοντας τη διεύθυνση IP ενός peer που βρίσκεται ήδη στο δίκτυο, και στη συνέχεια ο κόμβος «μαθαίνει» το υπόλοιπο σύστημα μέσω δομών δρομολόγησης που δημιουργούνται και επιδιορθώνονται δυναμικά. Λόγω αποκεντρωμένης και πλεονάζουσας αρχιτεκτονικής, δεν υπάρχει μοναδικό σημείο αποτυχίας (single point of failure), και ένας

κόμβος μπορεί να φύγει ανά πάσα στιγμή χωρίς προειδοποίηση, συνήθως με μικρή ή μηδενική απώλεια δεδομένων.

➢ Πως Υλοποιήθηκε Το Pastry

Το Pastry που υλοποιήσαμε ακολουθεί την ίδια φιλοσοφία υλοποίησης με το Chord. Έχουμε έναν κοινό χώρο IDs μεγέθους 2^m (και επιλέγουμε $m = 40$) και κάθε τίτλος ταινίας μετατρέπεται σε αριθμητικό key με hashing. Στο pastry_hash κάνουμε SHA-1 πάνω στο string και παίρνουμε modulo 2^m, ώστε όλοι οι κόμβοι και όλα τα δεδομένα να βρίσκονται στον ίδιο κυκλικό χώρο αναγνωριστικών. Αυτό είναι το πρώτο βήμα για να μπορέσουμε να πραγματοποιήσουμε αναζήτηση για κοντινότερο κόμβο και να κάνουμε δρομολόγηση που συγκλίνει.

Η βασική μονάδα είναι η κλάση PastryNode. Κάθε node έχει id, το ίδιο m, και για την αποθήκευση δεδομένων κρατάει πάλι ένα BPlusTree (self.btree), ώστε όταν τελικά φτάσουμε στον σωστό κόμβο, η αναζήτηση/εισαγωγή/διαγραφή του record να γίνεται τοπικά γρήγορα. Επίσης ο κόμβος κρατάει αναπαράσταση του id σε δυαδικό (id_bits), γιατί στο Pastry η δρομολόγηση βασίζεται σε κοινό prefix bits. Δύο ακόμη δομές είναι κρίσιμες, πρώτον ο route_table, ένας πίνακας δρομολόγησης ανά επίπεδο prefix και ανά bit (0/1), και δεύτερον το leaf_set, μια λίστα με έως leaf_size αριθμητικά κοντινούς γείτονες. Τέλος, το distance_to υπολογίζει την κυκλική απόσταση στο χώρο 2^m. Αυτό το χρησιμοποιούμε παντού για να δούμε ποιος κόμβος είναι πιο κοντά σε key.

```python
class PastryNode:
    def __init__(self, node_id: int, m: int, btree_size: int = 32, leaf_size: int = 8):
        self.id = node_id
        self.m = m
        self.btree = BPlusTree(btree_size)
        self.leaf_size = leaf_size

        # represent node id as binary string
        self.bit_len = m
        self.id_bits = f"{self.id:0{self.bit_len}b}"

        # route_table
        self.route_table: List[Dict[int, Optional[PastryNode]]] = [
            {0: None, 1: None} for _ in range(self.bit_len)
        ]

        # leaf set - up to leaf_size numerically closest neighbors
        self.leaf_set: List[PastryNode] = []

    def __repr__(self):
        return f"PNode({self.id})"

    def distance_to(self, key: int) -> int:
        """Return ring distance between this node id and key (modulo space)."""
        max_id = 2 ** self.m
        diff = abs(self.id - key)
        return min(diff, max_id - diff)
```

Όπως και στο Chord, έτσι και εδώ υλοποιείται η κλάση PastryRing . Αυτή κρατάει τη λίστα nodes, τις παραμέτρους leaf_size, m, και παρέχει όλα τα operations. Στα helper methods έχει δύο πολύ πρακτικά κομμάτια. Αρχικά, με _id_to_bits κάνει μετατροπή int σε δυαδικό string σταθερού μήκους και με _prefix_len μετράει πόσα αρχικά bits έχουν κοινά δύο IDs. Αυτό είναι το κλειδί της λειτουργίας του Pastry, γιατί το routing προσπαθεί να αυξήσει το κοινό prefix με το key σε κάθε hop. Υπάρχει και το _best_leaf_candidate, που κοιτάει τον τρέχοντα κόμβο και το leaf_set του και επιστρέφει ποιος είναι αριθμητικά πιο κοντά στο key, έτσι το Pastry εκμεταλλεύεται πρώτα τη γεωμετρία του χώρου IDs πριν πάει στο routing table.

```
42  class PastryRing:
43      def __init__(self, m: int = 40, leaf_size: int = 8, btree_size: int = 32):
44          self.m = m
45          self.nodes: List[PastryNode] = []
46          self.leaf_size = leaf_size
47          self.bit_len = m
48          self.btree_size = btree_size
49
50      # ------------------------ helpers ------------------------
51      def _id_to_bits(self, id_int: int) -> str:
52          return f"{id_int:0{self.bit_len}b}"
53
54      def _prefix_len(self, a_bits: str, b_bits: str) -> int:
55          i = 0
56          for ca, cb in zip(a_bits, b_bits):
57              if ca != cb:
58                  break
59              i += 1
60          return i
```

```
62      def _best_leaf_candidate(self, curr: PastryNode, key_int: int) -> PastryNode:
63          """Return numerically closest node among curr + leaf_set."""
64          best = curr
65          best_dist = curr.distance_to(key_int)
66          for ln in curr.leaf_set:
67              d = ln.distance_to(key_int)
68              if d < best_dist:
69                  best = ln
70                  best_dist = d
71          return best
```

Για τη συντήρηση της topology, η δική μας υλοποίηση κάνει _rebuild_all. Αντί να υλοποιήσουμε πλήρως τα αποκεντρωμένα maintenance μηνύματα του Pastry, κάνουμε κεντρικό rebuild των route tables και leaf sets μετά από τα join/leave. Υπολογίζουμε από την αρχή τα route tables και leaf και οι μετρήσεις hops εξακολουθούν να μετράνε μόνο overlay forwards, δηλαδή πόσες φορές αλλάζει κόμβο το routing για να υπολογισθεί το κόστος rebuild. Μέσα στο _rebuild_all ταξινομούμε τους κόμβους, ενημερώνουμε τα id_bits, χτίζουμε το leaf_set παίρνοντας τους κοντινούς γείτονες κυκλικά γύρω από τον κόμβο, και μετά γεμίζουμε το route_table με βάση το πρώτο bit που διαφέρει και συγκεκριμένα το level του

prefix. Για κάθε θέση κρατάμε τον πιο καλό αντιπρόσωπο, προτιμώντας κάποιον κόμβο που είναι πιο κοντά αριθμητικά στον τρέχοντα.

```
74      def _rebuild_all(self):
75          """
76          Recompute route tables and leaf sets for all nodes.
77          """
78          if not self.nodes:
79              return
80
81          self.nodes.sort(key=lambda x: x.id)
82
83          for node in self.nodes:
84              node.id_bits = self._id_to_bits(node.id)
85
86          for node in self.nodes:
87              idx = self.nodes.index(node)
88
89              # collect up to leaf_size neighbors around the node circularly
90              neighbors: List[PastryNode] = []
91              left = 1
92              right = 1
```

```
93      while len(neighbors) < self.leaf_size and (left <= len(self.nodes) or right <= len(self.nodes)):
94          if left <= len(self.nodes) - 1:
95              nleft = self.nodes[(idx - left) % len(self.nodes)]
96              if nleft is not node and nleft not in neighbors:
97                  neighbors.append(nleft)
98          if len(neighbors) < self.leaf_size and right <= len(self.nodes) - 1:
99              nright = self.nodes[(idx + right) % len(self.nodes)]
100             if nright is not node and nright not in neighbors:
101                 neighbors.append(nright)
102         left += 1
103         right += 1
104     node.leaf_set = neighbors[: self.leaf_size]
```

```
106     # routing table
107     node.route_table = [{0: None, 1: None} for _ in range(node.bit_len)]
108     for other in self.nodes:
109         if other is node:
110             continue
111         pref = self._prefix_len(node.id_bits, other.id_bits)
112         if pref < node.bit_len:
113             next_bit = int(other.id_bits[pref])
114             existing = node.route_table[pref].get(next_bit)
115             if existing is None:
116                 node.route_table[pref][next_bit] = other
117             else:
118                 # keep a better representative closer to node
119                 if other.distance_to(node.id) < existing.distance_to(node.id):
120                     node.route_table[pref][next_bit] = other
```

Το join στο PastryRing.join_node δουλεύει σε δύο φάσεις. Πρώτα μετράμε locate_hops, δηλαδή αν υπάρχουν ήδη κόμβοι, ξεκινάμε από έναν start node και κάνουμε routing προς το node_id με _route, μόνο και μόνο για να μετρήσουμε πόσα overlay hops θα χρειαζόταν. Έπειτα προσθέτουμε τον νέο PastryNode, κάνουμε _rebuild_all, και μετά υλοποιούμε μια πρακτική ανακατανομή όπου κοιτάμε τους κόμβους στο leaf_set του νέου και μεταφέρουμε όσα keys είναι πλέον πιο κοντά στον νέο κόμβο (με βάση distance_to). Για κάθε τέτοια μεταφορά μετράμε και ένα routing κόστος _route(n, k) ως moved_hops_total και στο τέλος επιστρέφουμε avg_move_hops και moved (πόσα records μετακινήθηκαν). Έτσι για join έχουμε και routing hops του event και κόστος ανακατανομής.

```python
    def join_node(self, node_id: int):
        """
        Returns:
        (new_node, locate_hops, avg_move_hops, moved_count)
        """
        locate_hops = 0
        if self.nodes:
            start = self.nodes[0]
            _, locate_hops = self._route(start, node_id)

        new_node = PastryNode(node_id, self.m, btree_size=self.btree_size, leaf_size=self.leaf_size)
        self.nodes.append(new_node)
        self._rebuild_all()

        moved = 0
        moved_hops_total = 0
        affected = list(new_node.leaf_set)
```

```python
            for n in affected:
                items = n.btree.get_all_items()
                for k, rec in items:
                    if new_node.distance_to(k) < n.distance_to(k):
                        # routing cost from n to key
                        _, h = self._route(n, k)
                        moved_hops_total += h
                        new_node.btree.insert(rec, k)
                        n.btree.delete(k)
                        moved += 1

            avg_move_hops = (moved_hops_total / moved) if moved > 0 else 0.0
            return new_node, locate_hops, avg_move_hops, moved
```

Στο leave (leave_node) κρατάμε την ίδια λογική που είχαμε συζητήσει και για Chord. Μετράμε μόνο το routing κόστος του κάθε event (routing_hops με _route(start, node_id)), αλλά δεν χρεώνουμε hops για τη μεταφορά/επανεισαγωγή δεδομένων. Πρακτικά, παίρνουμε όλα τα items του κόμβου που φεύγει, τον αφαιρούμε, κάνουμε _rebuild_all για να φτιαχτούν οι δομές, και μετά για κάθε record βρίσκουμε τον νέο owner με _route(start,k) (αγνοώντας τα hops) και το βάζουμε στο B+ tree του. Επιστρέφουμε (ok, routing_hops, moved_count) για να έχουμε αφενός το overlay κόστος και αφετέρου πόσο data movement προκάλεσε η αποχώρηση.

```python
    def leave_node(self, node_id: int):
        """
        Leave event (routing-only hops).
        routing_hops counts the overlay routing to reach the leaving node id.
        """
        node = next((n for n in self.nodes if n.id == node_id), None)
        if node is None:
            return False, 0, 0

        routing_hops = 0
        if self.nodes:
            start = self.nodes[0]
            _, routing_hops = self._route(start, node_id)

        if len(self.nodes) == 1:
            self.nodes.remove(node)
            return True, int(routing_hops), 0

        items = node.btree.get_all_items()
        self.nodes.remove(node)
        self._rebuild_all()

        start = self.nodes[0]
```

```python
        moved = 0
        for k, rec in items:
            owner, _ = self._route(start, k)
            owner.btree.insert(rec, k)
            moved += 1

        return True, int(routing_hops), moved
```

Η βάση της δρομολόγησης είναι η _route. Η υλοποίηση ακολουθεί τον βασικό χαρακτήρα του Pastry όπου πρώτα δοκιμάζουμε leaf-set βελτίωση, δηλαδή αν κάποιος στο leaf_set είναι πιο κοντά αριθμητικά στο key, πάμε εκεί (με κόστος ένα hop). Αν όχι, κάνουμε prefix routing, δηλαδή βρίσκουμε το πρώτο bit που διαφέρει (με _prefix_len) και κοιτάμε στο route_table για κόμβο που ταιριάζει στο επόμενο bit του key. Αν δεν υπάρχει κατάλληλη εγγραφή, κάνουμε fallback επιλέγοντας από όλους τους γνωστούς υποψηφίους (route table entries και leaf set) αυτόν που βελτιώνει περισσότερο το prefix ή, σε ισοπαλία, μικραίνει την απόσταση. Υπάρχει visited και max_hops για να αποφεύγονται κύκλοι και πιθανά κολλήματα. Τα hops που επιστρέφει η _route είναι ακριβώς πόσες φορές προωθήθηκε το αίτημα μεταξύ κόμβων.

Πάνω σε αυτή τη δρομολόγηση χτίζονται τα data operations. Το insert παίρνει ήδη hashed key, κάνει _route από έναν start node (ή τον πρώτο), και εισάγει στο owner.btree. Το insert_title κάνει hashing εσωτερικά και μετά καλεί insert. Το lookup κάνει hash, κάνει _route, και επιστρέφει records από search_key μαζί με hops. Το delete_key/delete_title ακολουθεί την ίδια ιδέα route με btree.delete και επιστροφή hops. Το update_movie_field κάνει route, ψάχνει με search_key, και αν υπάρχει record αλλάζει το πεδίο και επιστρέφει (True, hops)

αλλιώς (False, hops). Έτσι το Pastry είναι πλήρες ως προς insert/delete/update/lookup, και οι μετρήσεις είναι συγκρίσιμες με Chord γιατί πάντα μετράμε hops ως overlay forwarding.

```python
    def _route(self, start_node: PastryNode, key_int: int) -> Tuple[PastryNode, int]:
        """
        - First try to improve numerically using leaf_set.
        - Else use routing table on the first differing prefix bit.
        - Else fallback to best among known candidates (leaf_set,routing entries).
        """
        if not self.nodes:
            raise RuntimeError("No nodes in pastry ring")

        key_bits = self._id_to_bits(key_int)
        curr = start_node
        hops = 0
        visited = set()
        max_hops = max(8, len(self.nodes) * 4)

        while True:
            visited.add(curr.id)

            # leaf-set numerical improvement
            best_leaf = self._best_leaf_candidate(curr, key_int)
            if best_leaf is not curr:
                curr = best_leaf
                hops += 1
                if curr.id in visited or hops > max_hops:
                    return curr, hops
                continue
```

```python
        # prefix routing using routing table
        pref = self._prefix_len(curr.id_bits, key_bits)
        if pref >= self.bit_len:
            return curr, hops

        next_bit = int(key_bits[pref])
        candidate = curr.route_table[pref].get(next_bit)
        if candidate is not None and candidate.id not in visited:
            curr = candidate
            hops += 1
            if hops > max_hops:
                return curr, hops
            continue

        # fallback to any known candidate that improves prefix or distance
        candidates = []
        for row in curr.route_table:
            for n in row.values():
                if n is not None:
                    candidates.append(n)
        candidates += curr.leaf_set
```

```
        # remove visited & None & self
        cand2 = [c for c in candidates if c is not None and c.id not in visited and c is not curr]
        if not cand2:
            return curr, hops

        best = None
        best_pref = -1
        best_dist = None
        for c in cand2:
            p = self._prefix_len(c.id_bits, key_bits)
            d = c.distance_to(key_int)
            if p > best_pref or (p == best_pref and (best_dist is None or d < best_dist)):
                best = c
                best_pref = p
                best_dist = d

        if best is None:
            return curr, hops

        curr = best
        hops += 1
        if hops > max_hops:
            return curr, hops
```

Στο Pastry, το **insert** δουλεύει ως εξής. Παίρνουμε το ήδη hashed key_int (ή κάνουμε hash με insert_title), ξεκινάμε από έναν κόμβο επιλέγοντας συνήθως τον πρώτο και καλούμε _route(start, key_int). Η _route μετρά hops και επιστρέφει τον κόμβο-ιδιοκτήτη. Εκεί κάνουμε owner.btree.insert(record, key_int) και επιστρέφουμε τα hops.

```
    # ------------------------ data operations ------------------------
    def insert(self, key_int: int, record: dict, start_node: Optional[PastryNode] = None) -> int:
        if not self.nodes:
            raise RuntimeError("No nodes in pastry ring")
        if start_node is None:
            start_node = self.nodes[0]
        owner, hops = self._route(start_node, key_int)
        owner.btree.insert(record, key_int)
        return hops

    def insert_title(self, movie_title: str, record: dict, start_node: Optional[PastryNode] = None) -> int:
        key_int = pastry_hash(movie_title, m=self.m)
        return self.insert(key_int, record, start_node=start_node)
```

Lookup

Το **lookup** είναι παρόμοιο. Κάνουμε pastry_hash(title) για να πάρουμε key_int, επιλέγουμε start node και τρέχουμε _route μέχρι τον κόμβο που ταιριάζει καλύτερα στο key (leaf-set και prefix routing). Στο τέλος ψάχνουμε το B+ tree του owner με search_key(key_int) και επιστρέφουμε (records, hops) ώστε να φαίνεται και το αποτέλεσμα και το κόστος δρομολόγησης.

```python
    def lookup(self, movie_title: str, start_node: Optional[PastryNode] = None) -> Tuple[Optional[List[dict]], int]:
        key_int = pastry_hash(movie_title, m=self.m)
        if not self.nodes:
            return None, 0
        if start_node is None:
            start_node = self.nodes[0]
        owner, hops = self._route(start_node, key_int)
        records = owner.btree.search_key(key_int)
        return records, hops
```

Delete

Για **delete**, ακολουθούμε την ίδια λογική όπου βρίσκει τον owner και τον διαγράφει. Το delete_title κάνει hash και καλεί delete_key. Το delete_key κάνει _route(start, key_int) για να φτάσει στον σωστό κόμβο και μετά καλεί owner.btree.delete(key_int). Επιστρέφουμε τα hops της δρομολόγησης, επειδή αυτό είναι που συγκρίνουμε πειραματικά ως κόστος overlay.

```python
    def delete_key(self, key_int: int, start_node: Optional[PastryNode] = None) -> int:
        if not self.nodes:
            return 0
        if start_node is None:
            start_node = self.nodes[0]
        owner, hops = self._route(start_node, key_int)
        owner.btree.delete(key_int)
        return hops

    def delete_title(self, movie_title: str, start_node: Optional[PastryNode] = None) -> int:
        key_int = pastry_hash(movie_title, m=self.m)
        return self.delete_key(key_int, start_node=start_node)
```

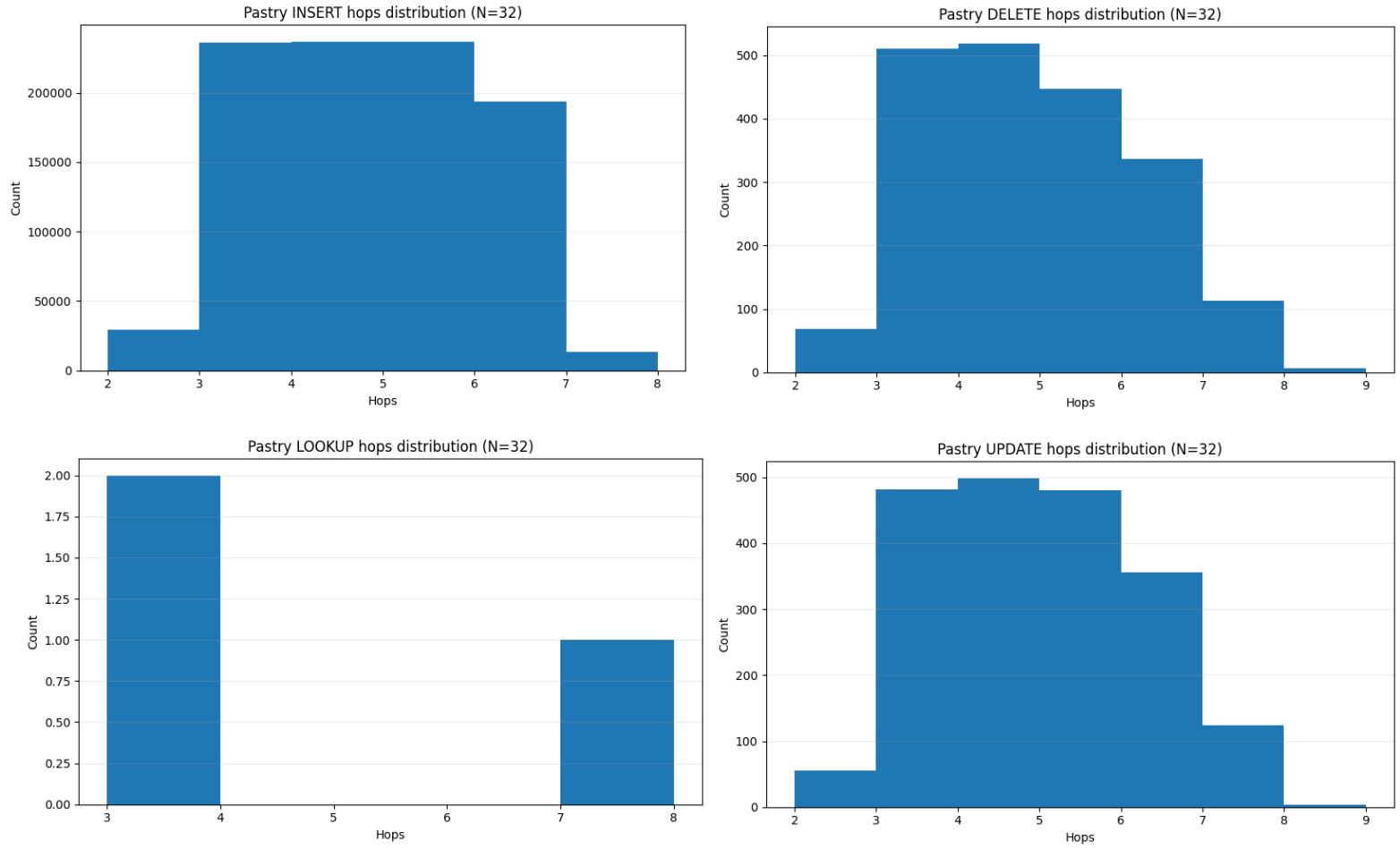
Update

Το **update** (update_movie_field) κάνει hash τον τίτλο, δρομολογεί με _route μέχρι τον owner, και μετά ψάχνει search_key(key_int). Αν δεν βρει record, επιστρέφει (False, hops). Αν βρει, παίρνει το πρώτο record και αλλάζει το πεδίο (rec[field] = new_value). Έτσι μετράμε καθαρά το routing κόστος, όχι την τοπική ενημέρωση.

```python
    def update_movie_field(self, title: str, field: str, new_value, start_node: Optional[PastryNode] = None) -> Tuple[bool, int]:
        key_int = pastry_hash(title, m=self.m)
        if not self.nodes:
            return False, 0
        if start_node is None:
            start_node = self.nodes[0]
        owner, hops = self._route(start_node, key_int)
        records = owner.btree.search_key(key_int)
        if not records:
            return False, hops
        rec = records[0]
        rec[field] = new_value
        return True, hops
```

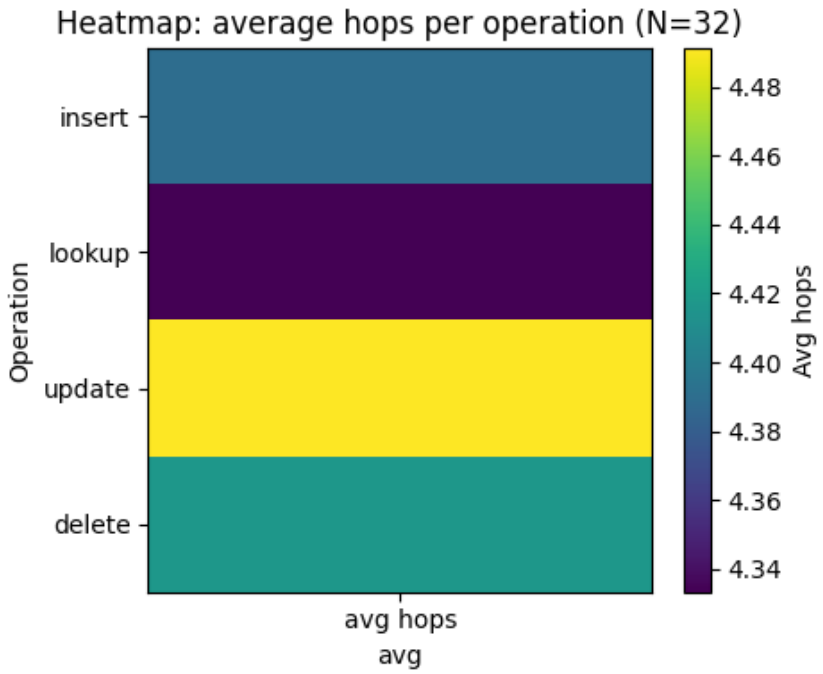
Τέλος, για να τρέξουμε το main_pastry.py η διαδικασία είναι ανάλογη με το main_chord.py, καθώς φορτώνουμε πλήρες dataset, δημιουργούμε PastryRing(m=40, leaf_size=8), κάνουμε initial joins με seed για αναπαραγωγιμότητα, εισάγουμε όλες τις εγγραφές, τρέχουμε churn (joins/leaves), και μετά updates/deletes/lookups. Εκεί οι λίστες hops που τυπώνονται και τα plots βασίζονται στα hops που επιστρέφει η _route, άρα μετράμε καθαρά το overlay routing κόστος.

Καταγράψαμε το ίδιο είδος μετρικών αποτελεσμάτων της εφαρμογής του Pastry με αυτής του Chord για μια άμεση σύγκριση απόδοσης υλοποίησης.

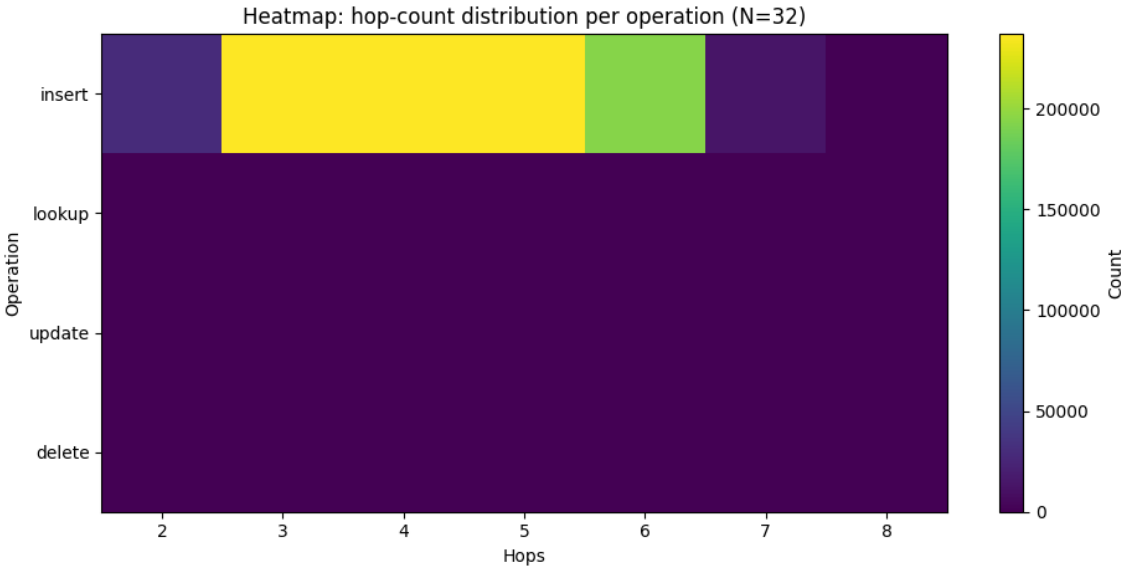


Εικόνα 6: Histograms (hops) για Pastry: Κατανομή των routing hops ανά operation — δείχνει πόσο συχνά εμφανίζεται κάθε αριθμός hops ανά operation.

Βλέπουμε ότι τα histograms του **Pastry** δείχνουν ότι τα hops κινούνται γενικά σε πιο υψηλές σε σχέση με του **Chord**. Στα **insert / update / delete** το πλήθος των τιμών συγκεντρώνεται κυρίως γύρω στα **3–6 hops**, με ουρά που φτάνει μέχρι **8–9**, ενώ στο Chord είχαμε πιο συχνά **3–5** και σπανιότερα τόσο μεγάλα άκρα. Στο **lookup** και στα δύο φαίνονται λίγα δείγματα λόγω του μικρού K που ορίσαμε στο παράδειγμά μας, αλλά πάλι στο Pastry βλέπουμε τιμές που τείνουν να είναι πιο βεβαρημένες από του Chord. Με απλά λόγια, στο δικό μας setup η Pastry δρομολόγηση χρειάζεται συχνότερα 1–2 βήματα παραπάνω, κάτι που προκύπτει κυρίως από την υλοποίηση του routing (leaf-set και prefix table και fallback), όπου όταν δεν υπάρχει καθαρό prefix match, γίνονται επιπλέον hops μέχρι να βρεθεί καλύτερος υποψήφιος.

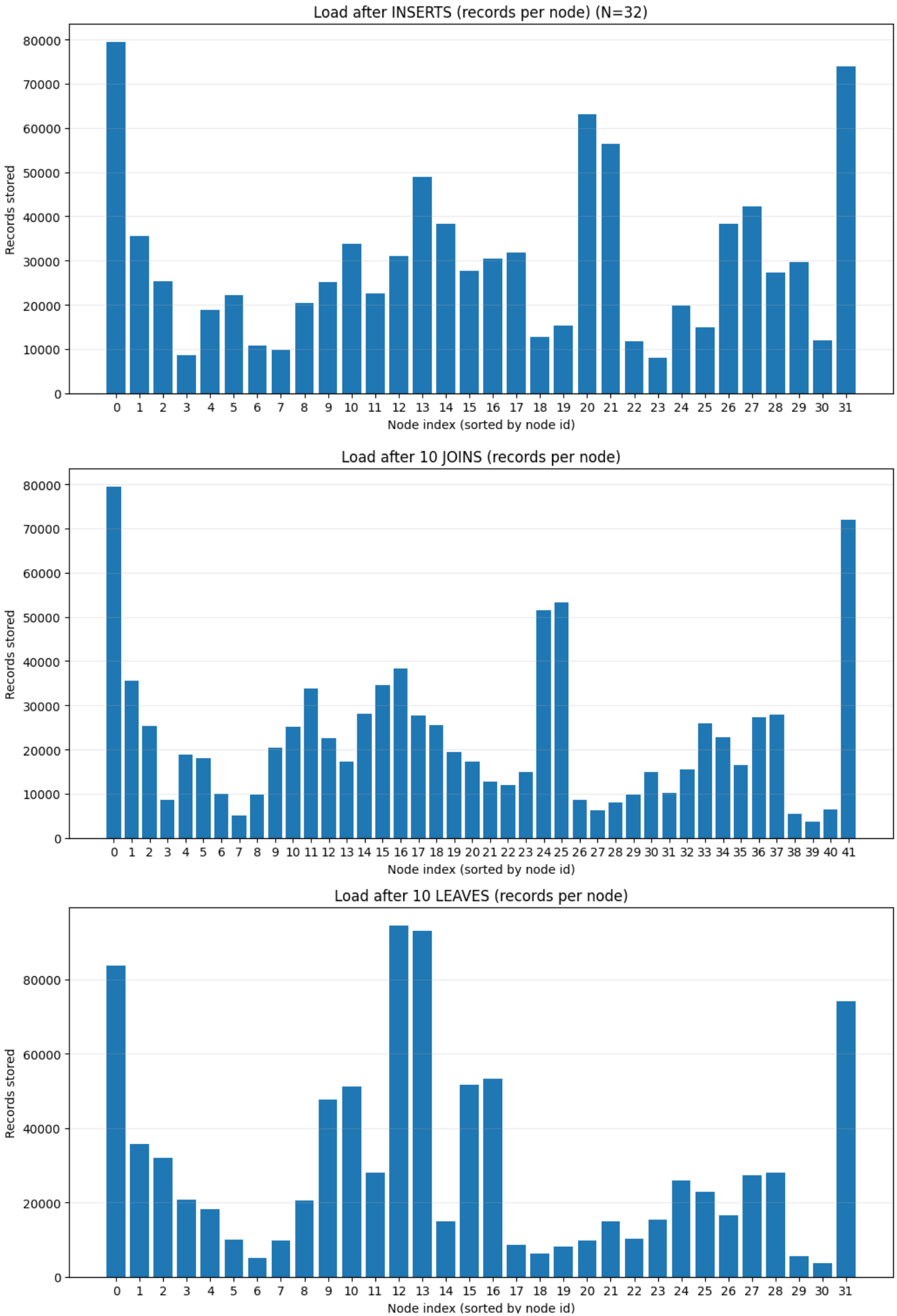


Εικόνα 7: Heatmap (avg hops) για Pastry: Μέσος αριθμός hops ανά operation — συνοπτική σύγκριση του μέσου κόστους δρομολόγησης



Εικόνα 8: Heatmap (hop-count distribution) για Pastry: Πυκνότητα/πλήθος μετρήσεων hops ανά operation — δείχνει πού συγκεντρώνονται τα περισσότερα δείγματα

Στα heatmaps του **Pastry** φαίνεται ότι τα **μέσα hops** κινούνται περίπου γύρω στο **4.3–4.5** για όλα τα operations, δηλαδή το σύστημα έχει σχετικά σταθερό κόστος δρομολόγησης χωρίς τεράστιες διαφορές από operation σε operation. Ενδιαφέρον είναι ότι εδώ το **lookup** δεν βγαίνει το πιο ακριβό όπως στο Chord, αλλά είναι από τα χαμηλότερα, κάτι που ταιριάζει με τη λογική του Pastry, όπου το prefix routing συχνά οδηγεί γρήγορα προς σωστή περιοχή του ID-space. Το δεύτερο heatmap hop-count distribution δείχνει πού υπάρχει όγκος μετρήσεων με το **insert** να είναι πιο γεμάτο γιατί έγινε μαζικό φόρτωμα dataset και οι περισσότερες τιμές πέφτουν στα **3–6 hops**, ενώ τα υπόλοιπα operations φαίνονται πιο άδεια επειδή έχουν πολύ λιγότερα δείγματα. Σε σχέση με το Chord, η γενική εικόνα είναι ότι στο Pastry το μέσο κόστος μένει γύρω στο 4.4, ενώ στο Chord είχατε χαμηλότερα averages για insert/update/delete αλλά πιο έντονη διαφοροποίηση με lookup.



Εικόνα 9: Loads (records per node) για Pastry: Κατανομή records ανά κόμβο μετά από inserts/joins/leaves — απεικονίζει πώς μοιράζεται το φορτίο στο δίκτυο

Στα **loads του Pastry** βλέπουμε πάλι καθαρά πώς αλλάζει η κατανομή των records όταν αλλάζουμε το node index. **Μετά τα inserts**, το φορτίο δεν είναι απόλυτα ομοιόμορφο καθώς μερικοί κόμβοι κρατάνε αισθητά περισσότερα records από άλλους, κάτι φυσιολογικό γιατί η κατανομή εξαρτάται από το πού έπεσαν τα node IDs και από το πώς το routing/ownership αποφασίζει ποιος είναι πιο κοντά σε κάθε key. **Μετά τα 10 joins**, η εικόνα γενικά σπάει σε περισσότερους κόμβους (έχουμε περισσότερα bars γιατί έχουμε περισσότερα nodes) και ένα μέρος του φορτίου μετακινείται, οπότε βλέπουμε ότι κάποια μεγάλα peaks μειώνονται ή μεταφέρονται αλλού, δηλαδή οι νέοι κόμβοι τραβάνε δεδομένα. **Μετά τα 10 leaves**, εμφανίζονται ξανά πιο έντονες κορυφές, γιατί οι κόμβοι που μένουν αναγκαστικά απορροφούν τα δεδομένα αυτών που έφυγαν, έτσι το load συγκεντρώνεται σε λιγότερα σημεία. Γενικά, τα τρία γραφήματα δείχνουν το αναμενόμενο μοτίβο ότι τα **joins τείνουν να απλώνουν** την αποθήκευση, ενώ τα **leaves τείνουν να τη συγκεντρώνουν**.

3.Αποθήκευση Των Recods Στους Κόμβους Με Χρήση Δομής B+ Tree

Η αποθήκευση των records στους κόμβους μας βασίστηκε σε ένα **B+ tree** , γιατί μας βολεύει πρακτικά. Το DHT (Chord/Pastry) λύνει το ζήτημα σε ποιον κόμβο ανήκει το key, αλλά μέσα στον κόμβο θέλουμε μια δομή που να κάνει γρήγορο insert/search/delete και να μπορεί να σαρώνει εύκολα όλα τα keys όταν χρειάζεται μεταφορά δεδομένων (join/leave). Για αυτό φτιάξαμε την κλάση BPlusTree με ρίζα ένα BPlusTreeNode, όπου κάθε node κρατάει values (τα ταξινομημένα keys) και keys (τα αντίστοιχα payloads/records). Στα φύλλα το keys είναι λίστα από λίστες, ώστε **αν δύο records έχουν ίδιο key**, να τα κρατάμε όλα μαζί (δηλαδή keys[i] είναι λίστα records για το values[i]).

```
4   class BPlusTreeNode:
5       def __init__(self, size):
6           self.is_leaf = False
7           self.values = []      # keys
8           self.keys = []        # leaf
9           self.parent = None
10          self.Next = None
11          self.size = size      # max number of keys
12
13      def insert_into_leaf(self, key, value):
14          # key = record payload, value = sortable key
15          if len(self.values) == 0:
16              self.values = [value]
17              self.keys = [[key]]
18              return
19
20          for i, val in enumerate(self.values):
21              if value == val:
22                  self.keys[i].append(key)
23                  return
24              elif value < val:
25                  self.values.insert(i, value)
26                  self.keys.insert(i, [key])
27                  return
28
29          # insert at end
30          self.values.append(value)
31          self.keys.append([key])
```

Η εισαγωγή γίνεται σε δύο βήματα. Πρώτα το search(value) κατεβαίνει από τη ρίζα μέχρι να βρει το σωστό φύλλο, ακολουθώντας τα child indexes στα εσωτερικά nodes. Όταν φτάσει σε φύλλο, το insert_into_leaf(key, value) βάζει το νέο key στη σωστή θέση για να μείνει ταξινομημένο και αν υπάρχει ήδη ίδια τιμή, απλώς κάνει append το record στη λίστα. Μετά το insert ελέγχει overflow και αν το φύλλο ξεπεράσει το επιτρεπτό size, καλείται split_leaf. Εκεί κόβουμε το φύλλο στη μέση (με mid = (size+1)//2), φτιάχνουμε new_leaf και μοιράζουμε τα μισά keys. Σημαντική λεπτομέρεια στο B+ tree είναι ο δείκτης Next καθώς συνδέουμε τα φύλλα σαν αλυσίδα, ώστε να μπορεί να πραγματοποιηθεί γρήγορο range scan και να μαζέψουμε όλα τα items.

```
39      def search(self, value):
40          curr = self.root
41          while not curr.is_leaf:
42              i = 0
43              while i < len(curr.values) and value >= curr.values[i]:
44                  i += 1
45              curr = curr.keys[i]
46          return curr
47
48      def insert(self, key, value):
49          leaf = self.search(value)
50          leaf.insert_into_leaf(key, value)
51
52          if len(leaf.values) > leaf.size:
53              self.split_leaf(leaf)
```

```
55      def split_leaf(self, leaf):
56          mid = (leaf.size + 1) // 2
57
58          new_leaf = BPlusTreeNode(leaf.size)
59          new_leaf.is_leaf = True
60
61          new_leaf.values = leaf.values[mid:]
62          new_leaf.keys = leaf.keys[mid:]
63
64          leaf.values = leaf.values[:mid]
65          leaf.keys = leaf.keys[:mid]
66
67          new_leaf.Next = leaf.Next
68          leaf.Next = new_leaf
69
70          new_leaf.parent = leaf.parent
71
72          promoted_key = new_leaf.values[0]
73          self.insert_in_parent(leaf, promoted_key, new_leaf)
```

Όταν γίνεται split, προωθούμε προς τα πάνω ένα κλειδί (promoted_key = new_leaf.values[0]) με το insert_in_parent. Αν το split έγινε στη ρίζα, δημιουργούμε νέα ρίζα με δύο παιδιά. Αν όχι, βρίσκουμε τη θέση του παιδιού στον parent και βάζουμε εκεί το promoted key και pointer στο νέο node. Αν τώρα overflow κάνει ο parent, σπάμε εσωτερικό node με split_internal και προωθούμε το μεσαίο key προς τα πάνω και μοιράζουμε τα παιδιά στα δύο, ενημερώνοντας και τα parent pointers. Έτσι διατηρείται το δέντρο ισορροπημένο.

```python
    def insert_in_parent(self, node, value, new_node):
        if self.root == node:
            new_root = BPlusTreeNode(node.size)
            new_root.values = [value]
            new_root.keys = [node, new_node]
            new_root.is_leaf = False
            self.root = new_root
            node.parent = new_root
            new_node.parent = new_root
            return

        parent = node.parent

        idx = parent.keys.index(node)
        parent.values.insert(idx, value)
        parent.keys.insert(idx + 1, new_node)
        new_node.parent = parent

        if len(parent.values) > parent.size:
            self.split_internal(parent)
```

```python
    def split_internal(self, node):
        mid = len(node.values) // 2
        promoted = node.values[mid]

        new_internal = BPlusTreeNode(node.size)
        new_internal.is_leaf = False

        new_internal.values = node.values[mid + 1:]
        new_internal.keys = node.keys[mid + 1:]

        for child in new_internal.keys:
            child.parent = new_internal

        node.values = node.values[:mid]
        node.keys = node.keys[:mid + 1]

        new_internal.parent = node.parent

        self.insert_in_parent(node, promoted, new_internal)
```

Για αναζήτηση record, η search_key(sha_key) βρίσκει το φύλλο και μετά κάνει γραμμική αναζήτηση μέσα στις values του φύλλου για exact match, επιστρέφοντας τη λίστα records ή None. Για ευκολία υπάρχει και search_title, που κάνει SHA-1 στον τίτλο και μετά καλεί search_key. Τέλος, το get_all_items πηγαίνει στο αριστερότερο φύλλο και διασχίζει όλα τα φύλλα μέσω Next, επιστρέφοντας όλα τα (key, record). Αυτό είναι ακριβώς που χρησιμοποιούμε στις μεταφορές δεδομένων όταν αλλάζει το δίκτυο (join/leave), ώστε να πάρουμε γρήγορα όλο το περιεχόμενο ενός κόμβου.

```
116     # searching by title ------- SHA-1
117     def search_title(self, title):
118         sha_key = int(hashlib.sha1(title.encode('utf-8')).hexdigest(), 16)
119         return self.search_key(sha_key)
120 
121     def search_key(self, sha_key):
122         leaf = self.search(sha_key)
123 
124         for i, v in enumerate(leaf.values):
125             if v == sha_key:
126                 return leaf.keys[i]   # return list of matching records
```