Πολυδιάστατες Δομές Δεδομένων

Project 2021-2022

DHTs (Chord)

Στυλιανός Στυλιανάκης	1059713
Χάρης Καπελετιώτης	1057772
Κωνσταντίνος Κωστόπουλος	1067482

Link to GitHub

Λίγη θεωρία

Distributed Hash Tables

Οι κατανεμημένοι πίνακες κατακερματισμού είναι μία δομή **αποκεντροποιημένης** διαμοίρασης αρχείων, δομημένων σε ζευγάρια κλειδιού-τιμής.

Κάθε **κόμβος** ενός DHT είναι υπεύθυνος για την αποθήκευση ενός υποσυνόλου κλειδιών και των τιμών τους, οι οποίες μπορεί να είναι οποιουδήποτε τύπου δεδομένων. Τα κλειδιά είναι μοναδικά αναγνωριστικά που δημιουργούνται μετά από **κατακερματισμό** των δεδομένων τιμών, ενώ οι κόμβοι που συμμετέχουν στη δομή δρουν ως **ισάξιες** οντότητες για τον διαμοιρασμό των αρχείων.

Ένα από τα κύρια πλεονεκτήματα των DHTs είναι ότι οι κόμβοι μπορούν να εισέρχονται και να απομακρύνονται από το σύστημα με συνοπτικές διαδικασίες ανανέωσης κλειδιών. Για αυτόν το λόγο, τέτοια συστήματα έχουν τη δυνατότητα να **κλιμακώνονται** μέχρι και έναν πολύ μεγάλο αριθμό κόμβων, ενώ η **ανοχή** τους **στις βλάβες** τα καθιστά ικανά να διαχειρίζονται **ταυτόχρονες** αφίξεις και αναχωρήσεις κόμβων.

Chord

Το Chord είναι ένα **πρωτόκολλο** κατανεμημένων πινάκων κατακερματισμού, το οποίο παρουσιάστηκε από το MIT το 2001 και βελτιώνει δραματικά την **πολυπλοκότητα** (και άρα το χρόνο) των αναζητήσεων κλειδιών.

Για την ισοκατανομή των κλειδιών στους κόμβους, το Chord χρησιμοποιεί **consistent hashing** αλγόριθμους για τον κατακερματισμό των κλειδιών. Συγκεκριμένα, ο βασικός αλγόριθμος που χρησιμοποιείται είναι ο **SHA-1**, ο οποίος χρησιμοποιήθηκε και στην εργασία, ύστερα από τροποποίησή του για να περιοριστούν οι τιμές σε μικρότερους **χώρους κατακερματισμού**.

Για την επιτάχυνση της αναζήτησης κλειδιών, οι κόμβοι αποθηκεύουν έναν πίνακα δρομολόγησης, γνωστό ως **finger table**, ο οποίος περιέχει KS εγγραφές. Κάθε εγγραφή αποτελείται από μία θέση και έναν κόμβο, ο οποίος ευθύνεται (πιθανώς μεταξύ άλλων και) για τα κλειδιά τα οποία ανήκουν στο διάστημα [θέση, κόμβος εγγραφής]. Λόγω του πίνακα αυτού, το κόστος της αναζήτησης ενός κλειδιού είναι O(log(n)).

Υλοποίηση

Κλάση Node

Η βασική κλάση της υλοποίησης, αναπαριστά έναν **κόμβο** δικτύου ο οποίος μπορεί να τρέξει όλες τις απαραίτητες διαδικασίες, όπως παρουσιάζονται παρακάτω.

`node.id`	Το id του κόμβου.
`node.items`	Dictionary που περιέχει τα στοιχεία του κόμβου. Τα κλειδιά του λεκτικού είναι τα unhashed κλειδιά των ζεύγων (κλειδιών, δεδομένων) και οι τιμές τους τα δεδομένα.
`node.f_table`	Το finger table του κόμβου (λίστα). Κάθε εγγραφή της λίστας αποτελείται από μία λίστα 2 στοιχείων της μορφής [position, node].
`node.pred`	Ο αμέσως προηγούμενος κόμβος.
`node.succ_list`	Λίστα με τους άμεσους successors μεγέθους SLS, μία global σταθερά που εμείς έχουμε ορίσει να είναι 3.

Εδώ να σημειωθεί πως το μέτρο κατά των **massive node failures** δεν καταφέραμε να το υλοποιήσουμε λόγω έλλειψης χρόνου και δυναμικού. Παρόλα αυτά, έχουν υλοποιηθεί τα θεμέλια για τη λειτουργία αυτή, αφού αποθηκεύουμε τη λίστα λίστα `succ_list` σε κάθε κόμβο, η οποία ενημερώνεται. Αυτό που λείπει λοιπόν είναι η **σύνδεση** της λίστας αυτής με τη διαδικασία **αναζήτησης κλειδιού** και η σωστή **ενημέρωση** των **finger tables** σε περίπτωση που κάποιος successor δεν ανταπεξέρχεται.

Βασικές συναρτήσεις της κλάσης Node

`find_successor`	Επιστρέφει τον αμέσως επόμενο (clockwise) successor του κόμβου, με την βοήθεια των μεθόδων `closest_pre_node` και `comp_cw_dist`.
insert_new_pred	Εισάγει στον εκάστοτε κόμβο, έναν συγκεκριμένο predecessor. Εισάγεται ουσιαστικά (join) ένας καινούριος κόμβος στο δίκτυο.
insert_item_to_node & delete_item_from_node,	Εισάγουν και διαγράφουν αντίστοιχα ένα item στον κόμβο.

initialize_finger_table	Αρχικοποιεί το finger table του κόμβου σύμφωνα με τον τύπο 〖n+2〗^(i-1) (mod 2^m), όπου το n είναι το id του κόμβου το m ο αριθμός των bit στο hash key.
leave	Διαγράφει έναν κόμβο απο το δίκτυο. Τα δεδομένα αυτού του κόμβου περνούν στον άμεσο successor του και έπειτα ανανεώνονται τα finger tables των κόμβων που απαιτείται (και μέσω της μεθόδου update_necessary_fingers()).

Κλάση Interface

Η κλάση αυτή αποτελεί, όπως λέει και το όνομά της, ένα interface από το οποίο **προσομοιάζεται** η πρόσβαση στους διαφορετικούς κόμβους του δικτύου, ώστε να μπορούμε να ξεκινάμε διαδικασίες ως οποιοσδήποτε κόμβος θέλουμε.

Βασικές συναρτήσεις της κλάσης Interface

build_network	Δημιουργεί το δίκτυο είτε με συγκεκριμένα node ids που έχουν περαστεί ως λίστα είτε με τυχαία, εφόσον ορίσουμε τον αριθμό κόμβων που θέλουμε να δημιουργηθούν.
insert_all_data	Διαβάζει τα δεδομένα του csv και τα περνάει στους κατάλληλους κόμβους.
insert_item	Βρίσκει τον υπεύθυνο κόμβο για ένα item και το εισάγει σε εκείνον.
node_join	Δεδομένου ενός id, δημιουργεί και προσθέτει έναν κόμβο στο δίκτυο.
node_leave	Δεδομένου ενός node id, βρίσκει (αν υπάρχει στο δίκτυο) και διαγράφει έναν κόμβο.
get_node	Δεδομένου ενός id, βρίσκει (αν υπάρχει) και επιστρέφει τον κόμβο. Εάν δεν υπάρχει στο δίκτυο, επιστρέφει τον παλαιότερο κόμβο του δικτύου.
range_query	Δεδομένου ενός εύρους, επιστρέφει τους κόμβους που βρίσκονται σε αυτό (inclusive on both ends).
knn	Δεδομένου ενός αριθμού k κι ενός id κόμβου επιστρέφει τους k πιο κοντινούς κόμβους του κόμβου με id = id.
exact match	Δεδομένου ενός id, επιστρέφει (αν υπάρχει) τον κόμβο με αυτό το id.

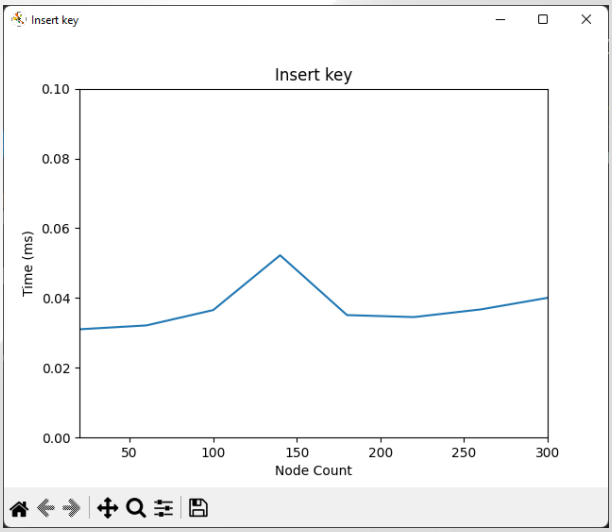
Συνάρτηση Κατακερματισμού

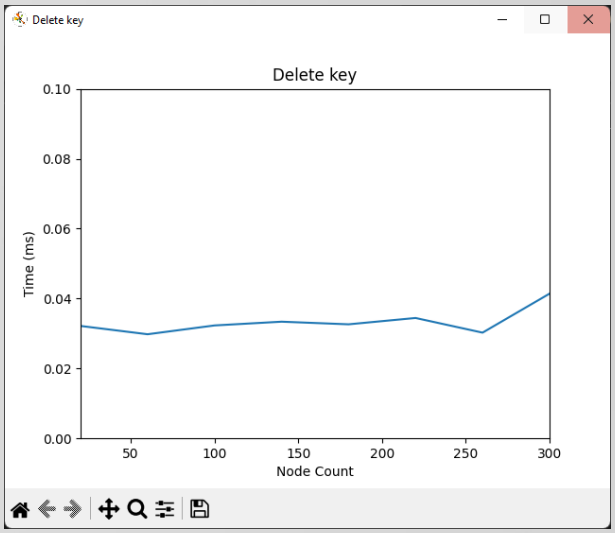
Στο παρόν πόνημα, ο βασικός αλγόριθμος κατακερματισμού που χρησιμοποιείται είναι ο SHA-1, με την συνάρτηση `hash_func(data)`.

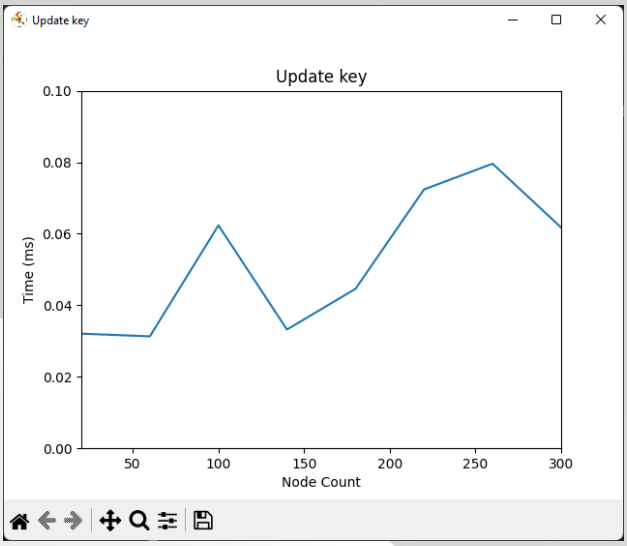
```
def hash_func(data) -> int:
    hash_out = hashlib.sha1()
    hash_out.update(bytes(data.encode("utf-8")))
    return int(hash_out.hexdigest(), 16) % HS
```

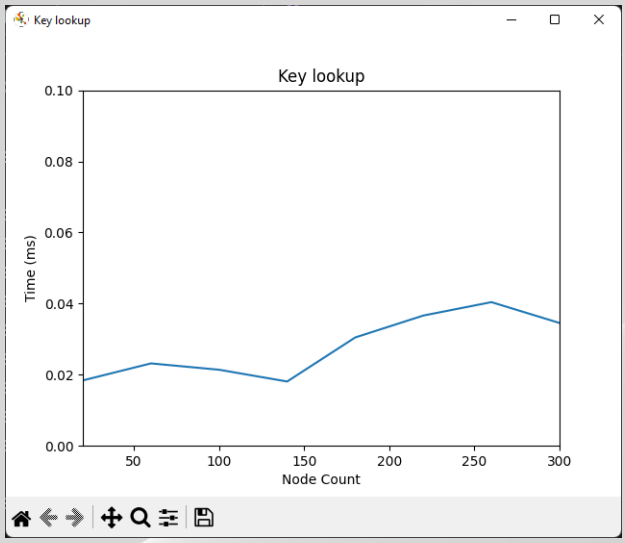
Πειραματική μελέτη

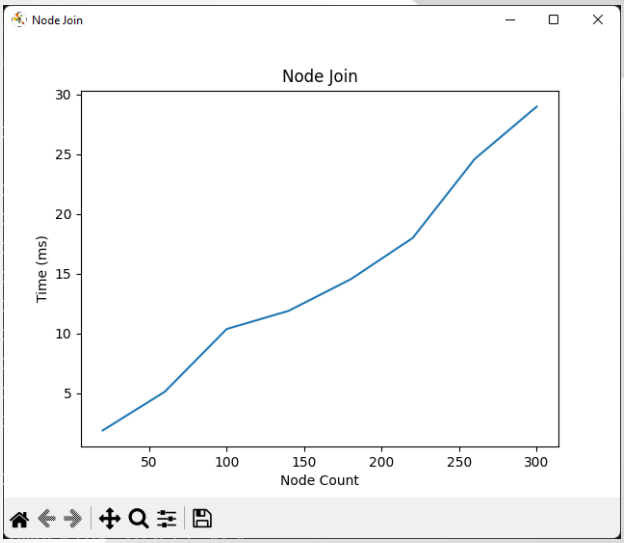
Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης των ζητούμενων διαδικασιών σε δίκτυα με διάφορους αριθμούς κόμβων (από 20 μέχρι 300). Για την εκτέλεση των benchmarks, το **KS** ορίστηκε στο **10** (άρα το hashing space είχε μέγεθος 1024), ενώ κάθε διαδικασία εκτελέστηκε 10 φορές για κάθε δίκτυο και πήραμε τον **μέσο όρο εκτέλεσης**. Τα πειράματα διεκπεραιώθηκαν σε φορητό υπολογιστή με επεξεργαστή Intel Core i7 8550u (15W, 1.8GHz Base Clock, Benchmarking Clocks: 3.1-3.3GHz).

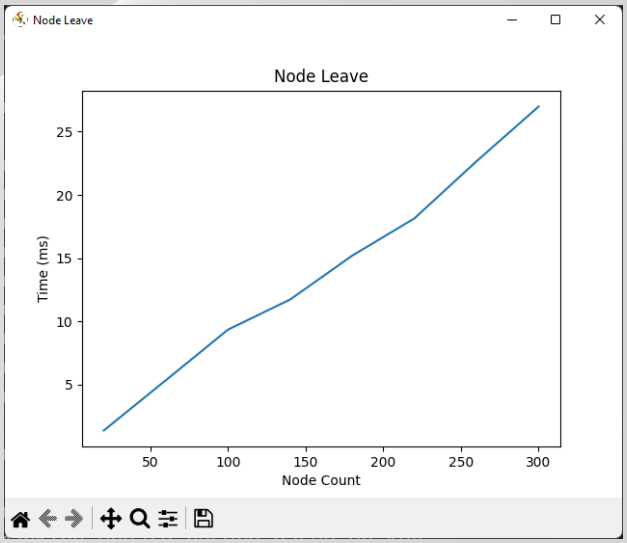


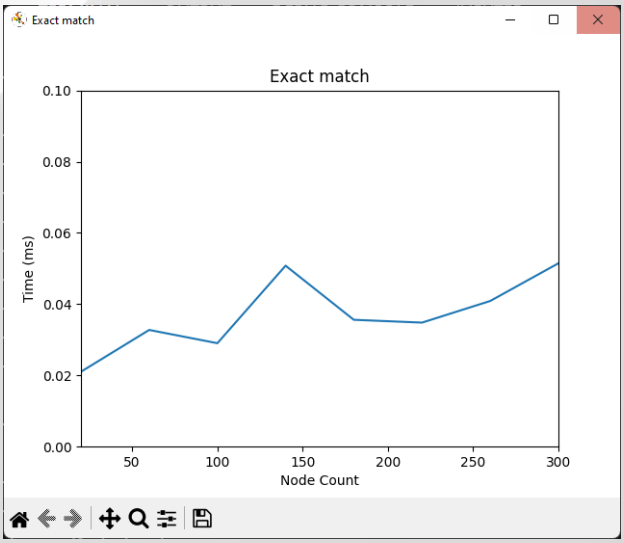


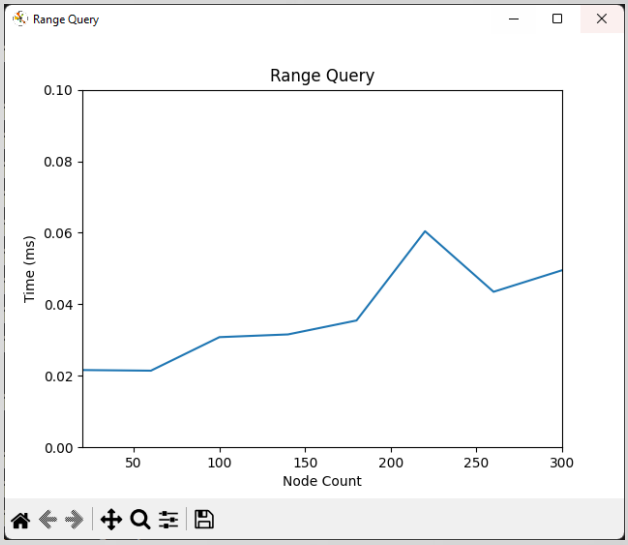

Update key
Update key
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300

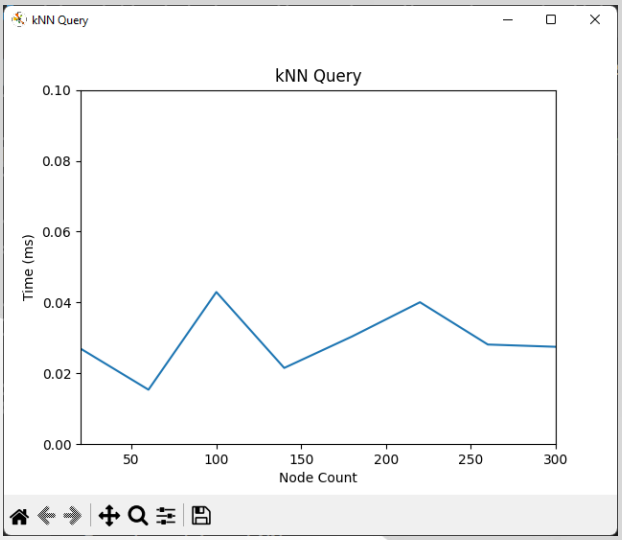

Key lookup
Key lookup
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300


Node Join
Node Join
Time (ms)
Node Count
5
10
15
20
25
30
50
100
150
200
250
300


Node Leave
Node Leave
Time (ms)
Node Count
5
10
15
20
25
50
100
150
200
250
300


Exact match
Exact match
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300


Range Query
Range Query
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300



Για το **range query** χρησιμοποιήθηκαν τυχαίες εμβέλειες μεγέθους 20, ενώ για το **kNN** χρησιμοποιήθηκαν τυχαίοι κόμβοι με k = 5.

Συμπέρασμα

Από τα παραπάνω γραφήματα, παρατηρούμε ότι οι **μόνοι** χρόνοι που επηρεάστηκαν ουσιαστικά από την αύξηση του αριθμού των κόμβων δικτύου είναι οι χρόνοι εισόδου και εξόδου κόμβου στο δίκτυο. Αυτό είναι λογικό, καθώς **δε χρησιμοποιούμε πολυνημάτωση** ώστε ο κάθε κόμβος να μπορεί να τρέχει σε άλλο νήμα, κάτι που όμως στον πραγματικό κόσμο **αποφεύγεται**, καθώς το φορτίο μοιράζεται στους υπολογιστές-κόμβους. Το σημαντικό είναι ότι οι χρόνοι εκτέλεσης των υπόλοιπων βασικών πράξεων **δεν παρατηρούν** κάποια αξιοσημείωτη **αύξηση** και έτσι το σύστημα παραμένει γρήγορο ανεξαρτήτως του αριθμού των κόμβων.