Εξαμηνιαία εργασία στα Κατανεμημένα Συστήματα

*Chordify**

`username: ******, password: ******`

Σπυρίδων Γαλανόπουλος (03120063) - Χαράλαμπος Πλάτανος (03120108)
Ανδρέας Στάμος (03120018)

1 Εισαγωγή

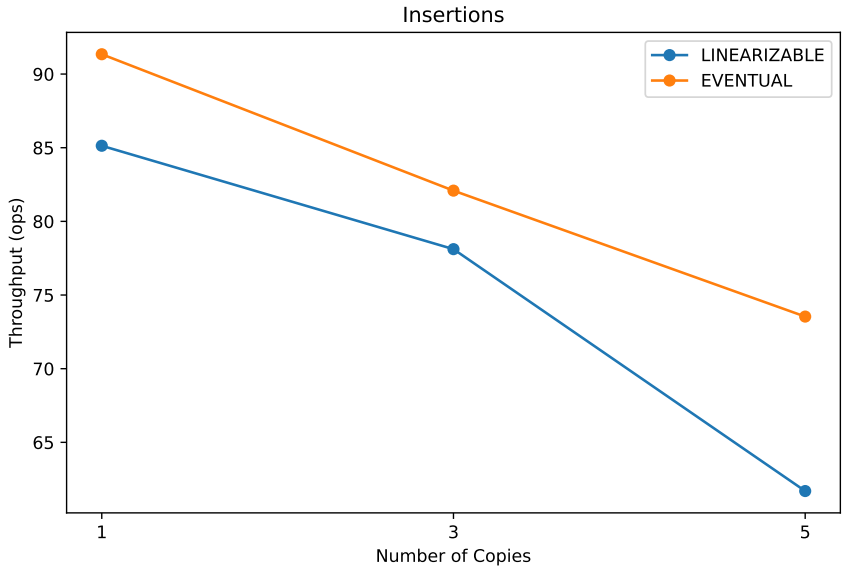
Στην παρούσα εξαμηνιαία εργασία υλοποιήσαμε μια απλουστευμένη εκδοχή του Chord DHT, το Chordify σε Python. Ειδικότερα, δόθηκε έμφαση στην υλοποίηση της δομής του δακτυλίου και της δρομολόγησης σε αυτόν, στην εισαγωγή και αναχώρηση κόμβων και τέλος στο replication των δεδομένων (επιλέξαμε chain replication). Ακόμη, υποθέσαμε ότι δεν υπάρχουν αποτυχίες κόμβων καθώς και ότι τα όποια αιτήματα στο σύστημα έρχονται αφού αυτό σύστημα έρθει σε ισορροπία. Επιπροσθέτως, υλοποιήσαμε και δρομολόγηση μέσω finger tables ενώ τέλος, φτιάξαμε και ένα web app για πιο άμεση και εύκολη αλληλεπίδραση με το σύστημα.

2 Πειράματα

Με τα scripts `cli/benchmark.py` και `cli/benchmark_consistency.py` αυτοματοποιήσαμε την εκτέλεση των πειραμάτων και για τις τρεις περιπτώσεις. Συγκεκριμένα, τρέξαμε τα insert, query και request πειράματα και για τους δύο τύπους consistency για αριθμό αντιγράφων 1, 3, και 5, με και χωρίς δρομολόγηση μέσω finger tables. Τα αποτελέσματα και ο σχολιασμός (χωρίς finger tables) παρουσιάζονται ακολούθως, (τα αποτελέσματα με finger tables βρίσκονται σε ξεχωριστή ενότητα στο τέλος).

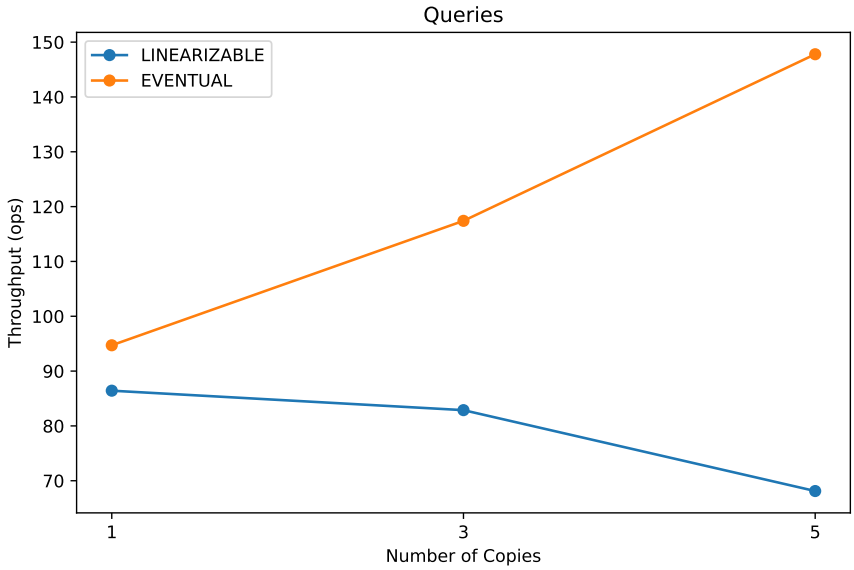
*https://chordify.ddnsfree.com

2.1 Insertions



Από το διάγραμμα παρατηρούμε ότι το write throughput και στις δύο περιπτώσεις consistency μειώνεται με την αύξηση replication factor. Ειδικότερα, στην περίπτωση του linearizability η μείωση αυτή γίνεται με πιο απότομο ρυθμό από την περίπτωση του eventual consistency. Η μείωση που παρατηρούμε είναι λογική καθώς με την αύξηση του k κάθε κόμβος είναι υπεύθυνος για την διατήρηση περισσότερων αρχείων με αποτέλεσμα να πρέπει να γίνουν περισσότερα writes και ανταλλαγές μηνυμάτων μεταξύ των κόμβων σε κάθε insert αυξάνοντας τον χρόνο. Μάλιστα, για το linearizablity επειδή τα writes πρέπει να γίνουν propagate στους επόμενους $k-1$ κόμβους που έχουν το αντίγραφο κατευθείαν η μείωση στο throughput είναι ακόμα πιο έντονη, σε αντίθεση με το eventual consistency όπου αυτό γίνεται lazily (αλλά και πάλι γίνονται παραπάνω HTTP requests οπότε έχουμε μείωση του throughput).

2.2 Queries



Από το διάγραμμα παρατηρούμε ότι με την αύξηση του replication factor το read throughput στην περίπτωση του linearizability μειώνεται ενώ στην περίπτωση του eventual consistency αυξάνεται (φυσικά για την περίπτωση $k = 1$ όπου δεν έχουμε replication έχουμε αρκετά κοντινό read throughput). Πράγματι, για το linearizability, με την αύξηση του k αφού θα πρέπει σε κάθε read να επιστρέφουμε την τιμή του τελευταίου κόμβου στο chain θα πρέπει να διατρέχουμε ολοένα και περισσότερους κόμβους για να φτάσουμε σε αυτόν με αποτέλεσμα την αύξηση του χρόνου απόκρισης. Αντίθετα, στην περίπτωση του eventual consistency, η ανάγνωση μπορεί να γίνει από οποιοδήποτε αντίγραφο συνεπώς με την αύξηση του k η πιθανότητα ένας κόμβος να έχει το ζητούμενο αρχείο (αντίγραφο) είναι μεγαλύτερη με αποτέλεσμα να χρειάζονται on expectation λιγότερα βήματα για την εύρεση του αρχείου, και ως εκ τούτου παρατηρούμε αύξηση του read throughput με την αύξηση του replication factor. Τέλος, σε σύγκριση με το write throughput στο πείραμα με τα insertions παρατηρούμε ότι το read throughput είναι πολύ μεγαλύτερο, πράγμα που είναι λογικό αφού τα reads εξυπηρετούνται πιο γρήγορα (αρκεί η απάντηση, δεν χρειάζεται να γίνει κάποιο propagation π.χ. όπως στη περίπτωση του linearizability).

2.3 Requests

Για το τρίτο και τελευταίο πείραμα για να συγκρίνουμε το freshness των τιμών μεταξύ των δύο τύπων συνέπειας επιλέξαμε το εξής framework: ορίσαμε ένα τυχαίο interleaving των requests όλων των κόμβων ώστε να έχουμε ένα απόλυτο reference και να μπορούμε να συγκρίνουμε το freshness μεταξύ των υλοποιήσεων αφού σε διαφορετική περίπτωση θα είχαμε διαφορετική σειρά των operations μεταξύ των πειραμάτων για τους δύο τύπους οπότε η εξαγωγή συμπερασμάτων δεν θα ήταν εύκολη. Ουσιαστικά, καθορίζοντας μια σειρά των requests (π.χ. πρώτα το πρώτο request του 3 ύστερα το πρώτο του 5 κ.ο.κ.) μπορούμε να καθορίσουμε τι θα έπρεπε να βλέπουμε και να μετρήσουμε έτσι stale reads. Για λόγους πληρότητας, τρέξαμε το πείραμα για $k = 1, 3, 5, 8, 10$ και

λάβαμε τα εξής αποτελέσματα:

Consistency Model	Replication Factor	Stale Reads
LINEARIZABLE	1	0
EVENTUAL	1	0
LINEARIZABLE	3	0
EVENTUAL	3	0
LINEARIZABLE	5	0
EVENTUAL	5	2
LINEARIZABLE	8	0
EVENTUAL	8	16
LINEARIZABLE	10	0
EVENTUAL	10	22

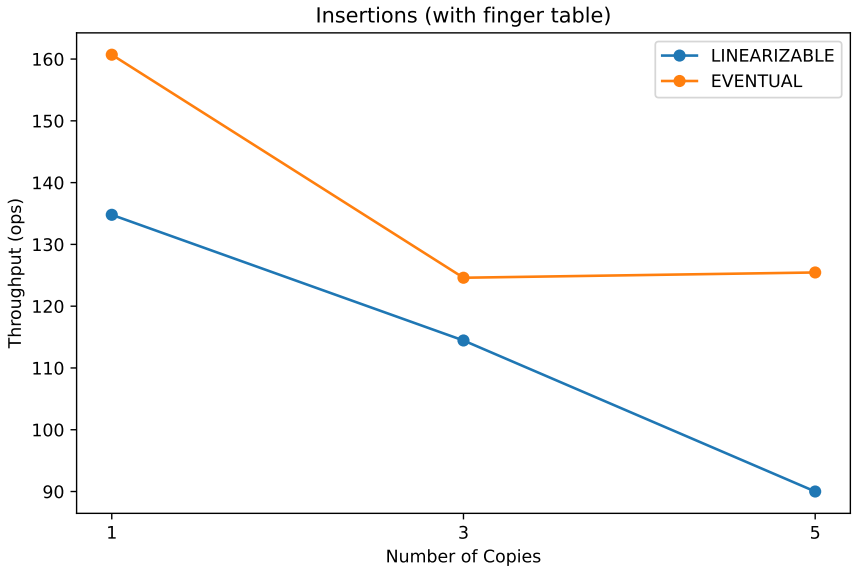
Όπως αναμενόταν τις πιο φρέσκες τιμές τις έχει το linearizability όπου σε κάθε περίπτωση έχουμε 0 stale reads (όπως θα έπρεπε άλλωστε εξ ορισμού). Στην περίπτωση του eventual consistency όσο αυξάνεται ο αριθμός των copies αυξάνονται τα stale reads έχουμε και άρα έχουμε λιγότερα fresh δεδομένα . Αυτό συμβαίνει καθώς με την αύξηση του k αυξάνεται ο χρόνος που απαιτείται για να γίνουν update όλα τα αντίγραφα αλλά εμείς πραγματοποιούμε όπως είδαμε στο δεύτερο πείραμα τα queries με μεγαλύτερη ταχύτητα, οπότε διαβάζουμε ολοένα και παλιότερες τιμές (θυμίζουμε ότι το τρίτο πείραμα περιέχει στην πλειοψηφία του queries). Βέβαια, για να δούμε την συμπεριφορά αυτή χρειάζεται μεγάλος αριθμός αντιγράφων (> 3) αφού για μικρό k υπάρχει αρκετός χρόνος ανάμεσα στα queries ώστε οι αλλαγές στα δεδομένα να προλαβαίνουν να γίνουν propagate σε όλα τα αντίγραφα.

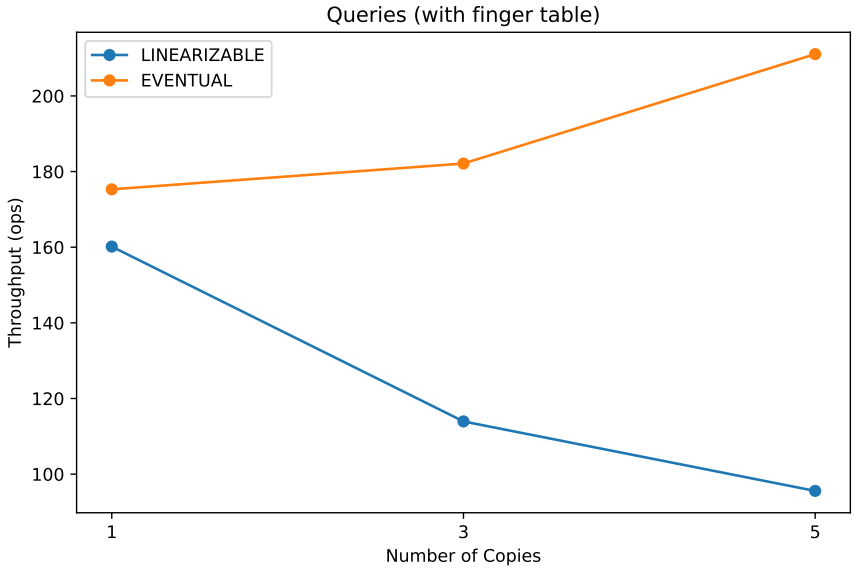
3 Συμπεράσματα

Συμπερασματικά, τα παραπάνω πειράματα ανέδειξαν το tradeoff μεταξύ throughput και freshness των τιμών στους δύο τύπους consistency. Ενώ το eventual consistency επιτρέπει υψηλότερο throughput με την αύξηση του replication factor, αυτό γίνεται εις βάρος του freshness των δεδομένων, καθώς τα αντίγραφα ενημερώνονται με καθυστέρηση. Αντίθετα, το linearizability διασφαλίζει τις πιο πρόσφατες τιμές σε κάθε ανάγνωση, αλλά με σημαντικό κόστος στο throughput, ιδιαίτερα σε write-heavy workloads. Είναι, λοιπόν, φανερό ότι δεν υπάρχει ένα σωστό είδος consistency αλλά οι ανάγκες της εκάστοτε εφαρμογής ορίζουν το πλαίσιο μέσα στο οποίο θα πρέπει να γίνει η επιλογή του κατάλληλου τύπου συνέπειας.

4 Πειράματα με finger tables

Προαιρετικά, υλοποιήσαμε και δρομολόγηση διαμέσου finger tables. Τρέχοντας τα ίδια πειράματα με παραπάνω, λάβαμε τα ακόλουθα αποτελέσματα:


Insertions (with finger table)
LINEARIZABLE
EVENTUAL
Throughput (ops)
160
150
140
130
120
110
100
90
1
3
5
Number of Copies


Queries (with finger table)
LINEARIZABLE
EVENTUAL
Throughput (ops)
200
180
160
140
120
100
1
3
5
Number of Copies

Consistency Model	Replication Factor	Stale Reads
LINEARIZABLE	1	0
EVENTUAL	1	0
LINEARIZABLE	3	0
EVENTUAL	3	0
LINEARIZABLE	5	0
EVENTUAL	5	4
LINEARIZABLE	8	0
EVENTUAL	8	15
LINEARIZABLE	10	0
EVENTUAL	10	35

Αρχικά, παρατηρούμε ότι τόσο το read όσο και το write throughput είναι αυξημένα όπως αναμέναμε καθώς με την χρήση δρομολόγησης μέσω finger table χρειάζονται on expectation λιγότερα βήματα για να φτάσουμε στον επιθυμητό κόμβο. Ποιοτικά, τα αποτελέσματα είναι παρόμοια με εκείνα χωρίς finger tables. Αξίζει να σημειώσουμε για το τρίτο πείραμα ότι τα stale reads αυξήθηκαν, πιθανώς διότι το throughput είναι μεγαλύτερο οπότε το σύστημα δεν προλαβαίνει στον ίδιο βαθμό να έχει updated όλα τα αντίγραφα όπως προηγουμένως.