



ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

Υλοποίηση HiNode μοντέλου σε Κατανεμημένη SQL βάση δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΚΑΠΕΛΕΤΙΩΤΗ ΧΑΡΙΛΑΟΥ

Επιβλέπων: Τσίχλας Κωνσταντίνος
Αναπληρωτής Καθηγητής

Συνεπιβλέπων: Σιούτας Σπυρίδων
Καθηγητής

Συνεπιβλέπων: Μακρής Χρήστος
Αναπληρωτής Καθηγητής

Πάτρα, Ιούλιος 2024



Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής

Υλοποίηση HiNode μοντέλου σε Κατανεμημένη SQL βάση δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΚΑΠΕΛΕΤΙΩΤΗ ΧΑΡΙΛΑΟΥ

Επιβλέπων: Τσίγλας Κωνσταντίνος
Αναπληρωτής Καθηγητής

Συνεπιβλέπων: Σιούτας Σπυρίδων
Καθηγητής

Συνεπιβλέπων: Μακρής Χρήστος
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10/07/2024.

(Υπογραφή)

(Υπογραφή)

(Υπογραφή)

.....
Τσίγλας Κωνσταντίνος
Αναπληρωτής Καθηγητής

.....
Σιούτας Σπυρίδων
Καθηγητής

.....
Μακρής Χρήστος
Αναπληρωτής Καθηγητής

Πάτρα, Ιούλιος 2024



Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής

Copyright ©–All rights reserved Καπελετιώτης Χαρίλαος, 2024.

Με την επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

Το περιεχόμενο αυτής της εργασίας δεν απηχεί απαραίτητα τις απόψεις του Τμήματος, του Επιβλέποντα, ή της επιτροπής που την ενέκρινε.

Υπεύθυνη Δήλωση

Βεβαιώνω ότι είμαι συγγραφέας αυτής της διπλωματικής εργασίας, και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην διπλωματική εργασία. Επίσης έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης, βεβαιώνω ότι αυτή η διπλωματική εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για τις απαιτήσεις του προγράμματος σπουδών του Τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής του Πανεπιστημίου Πατρών.

(Υπογραφή)

.....

Καπελετιώτης Χαρίλαος

Ευχαριστίες

Πρωτίστως, θα ήθελα να ευχαριστήσω τον καθηγητή μου, κ. Κωνσταντίνο Τσίχλα, για την δυνατότητα που μου παρείχε να ασχοληθώ με ένα θέμα των ενδιαφερόντων μου, καθώς επίσης και για την άψογη συνεργασία μας καθ' όλο το διάστημα της εκπόνησης της εργασίας μου. Ακόμα θα ήθελα να ευχαριστήσω εξίσου τον Αλέξανδρο Σπιτάλα για την καθοδήγησή του και τις ώρες που αφιέρωσε.

Τέλος, ευχαριστώ τη Ναυσικά, την οικογένεια και τους φίλους μου για την στήριξη και την αγάπη τους.

Περίληψη

Στην παρούσα διπλωματική εργασία παρουσιάζεται η σχεδίαση και υλοποίηση μοντέλων, τα οποία στηρίζονται στη μελέτη του κορυφο-κεντρικού και ασυμπτωτικά χωρικά βέλτιστου μοντέλου αποθήκευσης HiNode.

Αρχικά γίνεται μια εισαγωγή στις βασικές διαφορές των βάσεων δεδομένων, οι οποίες έχουν χρησιμοποιηθεί σε προηγούμενες εργασίες, με τη βάση δεδομένων που χρησιμοποιείται σε αυτή την υλοποίηση. Επιπλέον μελετώνται τα ιστορικά γραφήματα και πώς προέκυψε η ανάγκη δημιουργίας ενός μοντέλου αποθήκευσης για αυτά.

Στα επόμενα κεφάλαια αναλύονται τα μοντέλα και οι αλγόριθμοι που υλοποιήθηκαν στο πλαίσιο αυτής της εργασίας, καθώς επίσης και η τροποποίηση που έγινε στην γλώσσα ερωτημάτων Gremlin, με σκοπό τη προσομοίωση των χαρακτηριστικών του μοντέλου HiNode.

Τέλος, γίνεται η σύγκριση της παρούσας υλοποίησης με την προγενέστερή της, και σχολιάζονται τα αποτελέσματα.

Abstract

In this thesis the design and implementation of models is presented, which are based on the study of vertex-centric and asymptotically space-optimal HiNode storage model.

First, an introduction is made to the basic differences between databases, which have been used in previous work, and the database used in this implementation. In addition, historical graphs are studied and how the need arose to create a storage model for them.

In the following chapters the models and algorithms implemented in the framework of this thesis are analyzed, as well as the modification made to the Gremlin query language, in order to simulate the characteristics of the HiNode model.

Finally, the present implementation is compared to its predecessor, and the results are commented.

Στη μνήμη του πατέρα μου, Γιάννη

Περιεχόμενα

Περίληψη	ii
Abstract	iii
Κατάλογος Σχημάτων	viii
Ακρωνύμια και Συντμήσεις	x
1 Εισαγωγή	1
1.1 Τι είναι Distributed SQL	1
1.2 Η Αρχιτεκτονική της CockroachDB	3
1.2.1 Η αρχιτεκτονική του CockroachDB Cluster	3
1.2.2 Η στοίβα λογισμικού της CockroachDB	4
1.3 Ιστορικά Γραφήματα και HInode	7
1.3.1 Τα ιστορικά γραφήματα	8
1.3.2 Μοντέλα Ιστορικών Γραφημάτων	8
1.3.3 Η δομή του HiNode	11
2 Υλοποίηση Μέρους A	12
2.1 Go	12
2.1.1 Goroutines	13
2.1.2 pgx	15
2.2 Βασικά Στοιχεία Υλοποίησης	15
2.2.1 Διασύνδεση	15
2.2.2 SingleTable Μοντέλο	17
2.2.3 MultipleTable Μοντέλο	18
2.2.4 Ερωτήματα	19
3 Υλοποίηση Μέρους B	27
3.1 Gremlin	27
3.2 Apache Groovy	28
3.3 Gremlin Parser	31

4	Συγκρίσεις και Αποτελέσματα	32
4.1	CockroachDB και MongoDB	32
4.1.1	Εισαγωγή Δεδομένων	32
4.1.2	Degree Distribution	33
4.1.3	One Hop Neighborhood	34
4.2	Διαφορετικές υλοποιήσεις	34
4.3	Συμπεράσματα και μελλοντικές εργασίες	36
A'	Περισσότερες συγκρίσεις	38
	Βιβλιογραφία	41

Κατάλογος Σχημάτων

1.1	Αρχιτεκτονική του CockroachDB cluster	4
1.2	Η στοίβα λογισμικού της CockroachDB	5
1.3	Παράδειγμα εγγραφής σε LSM δέντρο	6
1.4	Μη-κατευθυνόμενος υπεργράφος	7
1.5	Κατευθυνόμενος υπεργράφος	7
1.6	Στιγμιότυπα από ιστορικό γράφημα	8
1.7	Ιστορικό γράφημα που αποθηκεύει πληροφορία στις κορυφές	10
2.1	Δομή του directory	16
2.2	Η δομή του SingleTable	17
2.3	Η δομή του MultipleTable	18
2.4	Έγκυρα χρονικά διαστήματα	20
2.5	Παράδειγμα επιστροφής ερωτήματος 2.10	25
3.1	Διαπέραση γραφήματος στην gremlin	27
4.1	Degree Distribution στις βάσεις δεδομένων (3/3 του χρόνου)	33
4.2	One Hop Neighborhood στις βάσεις δεδομένων (3/3 του χρόνου)	34
4.3	Χρησιμοποίηση μνήμης στις διαφορετικές υλοποιήσεις	35
4.4	Διαφορετικές υλοποιήσεις του degree distribution (3/3 του χρόνου)	35
A'.1	Degree Distribution (1/3 και 2/3 του χρόνου)	38
A'.2	One Hop Neighborhood (1/3 και 2/3 του χρόνου)	39

Παραδείγματα Κώδικα

2.1	Επιστροφή τιμών και διαχείριση σφάλματος	12
2.2	Δημιουργία struct και μεθόδου του	13
2.3	Goroutines και channels	14
2.4	Mutex και WaitGroup	14
2.5	Δημιουργία και σύνδεση σε βάση δεδομένων	16
2.6	Κλήση των ερωτημάτων	20
2.7	Ερώτημα get alive vertices	20
2.8	Ερώτημα get one hop neighborhood	21
2.9	Παλιό ερώτημα get degree distribution	21
2.10	Καινούριο ερώτημα get degree distribution	22
3.1	Χρήση της Gremlin μέσω της Groovy	29
3.2	Μεταπρογραμματισμός με τη Groovy	29
3.3	Εφαρμογή των custom gremlin steps	30
3.4	Αποθήκευση δεδομένων από την gremlin στην CockroachDB	31

Ακρωνύμια και Συντμήσεις

DB	Database
CRDB	CockroachDB
KV	Key-Value
SQL	Structured Query Language
SST	Sorted Strings Table
WAL	Write Ahead Log
CSP	Communicating Sequential Processes
ORM	Object Relational Mapping
ST	Single Table (Model)
MT	Multiple Table (Model)
JVM	Java Virtual Machine

Κεφάλαιο 1

Εισαγωγή

Στο κεφάλαιο αυτό θα μελετήσουμε τόσο τις διαφορές μεταξύ των βάσεων δεδομένων όσο και την αρχιτεκτονική της βάσης δεδομένων που χρησιμοποιείται στην υλοποίηση της εργασίας. Έπειτα θα γίνει μία εισαγωγή στα ιστορικά γραφήματα και στο θεωρητικό μοντέλο HiNode που εξετάζει η παρούσα διπλωματική εργασία.

1.1 Τι είναι Distributed SQL

Η σχεσιακή βάση δεδομένων, όπως ορίστηκε από τον E.F. Codd στην IBM το 1970 στην ερευνητική του εργασία [1] · ένα κατά τ' άλλα διαισθητικά δομημένο και καλώς ορισμένο μοντέλο, σχεδιάστηκε και εξελίχθηκε για την καταγραφή δεδομένων, με γνώμονα την συνέπεια και την κάθετη κλιμάκωση. Αυτές οι βάσεις δεδομένων εξυπηρετούσαν πολύ καλά το σκοπό τους, την εποχή, βέβαια, των μονολιθικών εφαρμογών που έτρεχαν σε ένα αφοσιωμένο single-server περιβάλλον. Ωστόσο, η αρχιτεκτονική αυτή έρχεται σε αντιπαράθεση με την ανάγκη των σύγχρονων καταναεμημένων εφαρμογών, όπου είναι απαιτητή η οριζόντια κλιμάκωση.

Η οριζόντια και η κάθετη κλιμάκωση είναι δύο θεμελιώδεις έννοιες στα συστήματα διαχείρισης βάσεων δεδομένων, που αφορούν την επεκτασιμότητα και τη βελτιστοποίηση των επιδόσεών τους. Η οριζόντια κλιμάκωση, γνωστή και ως scaling out, αναφέρεται στην προσθήκη περισσότερων κόμβων σε ένα σύστημα για την κατανομή του φόρτου εργασίας σε πολλαπλές μονάδες. Αντιθέτως, η κάθετη κλιμάκωση, που ονομάζεται και scaling up, αναφέρεται στην προσθήκη πόρων (CPU, RAM, ROM) ενός μεμονωμένου κόμβου, ώστε να αντιμετωπιστεί το αυξημένο φόρτο εργασίας. Το ζήτημα αυτό της απαίτησης της χρήσης συστημάτων που κλιμακώνονται οριζόντια, ήρθαν να επιλύσουν οι NoSQL βάσεις δεδομένων, η αρχιτεκτονική των οποίων δεν τους επιτρέπει να χρησιμοποιηθούν ως transactional βάσεις δεδομένων υπό την έννοια της αυστηρής συνέπειας. Κατά το θεώρημα CAP [2], ένα καταναεμημένο σύστημα αδυνατεί ταυτοχρόνως να εγγυηθεί όλα από τα παρακάτω:

1. **Consistency** (Συνέπεια): Κάθε ανάγνωση λαμβάνει την πιο πρόσφατη εγγραφή ή ένα σφάλμα.
2. **Availability** (Διαθεσιμότητα): Κάθε αίτημα λαμβάνει μια απάντηση, χωρίς να εγγυάται ότι περιέχει την πιο πρόσφατη εγγραφή.

3. **Partition tolerance** (Ανοχή Διαμερισμάτων): Το σύστημα συνεχίζει να λειτουργεί παρά τον αυθαίρετο αριθμό μηνυμάτων που απορρίπτονται (ή καθυστερούν) από το δίκτυο μεταξύ των κόμβων.

Το θεώρημα CAP αναδεικνύει τις συμβιβαστικές αποφάσεις που πρέπει να λάβει ένα κατανεμημένο σύστημα μεταξύ της συνέπειας και της διαθεσιμότητας, σε περιπτώσεις αποτυχίας κόμβων ή δικτύου. Το σύστημα πρέπει είτε να ακυρώσει τη διαδικασία και συνεπώς να μειώσει τη διαθεσιμότητα, αλλά να εξασφαλίσει τη συνέπεια (**CP**) · είτε να συνεχίσει τη διαδικασία και έτσι να εξασφαλίσει τη διαθεσιμότητα αλλά να διακινδυνεύσει την ασυνέπεια (**AP**).

Οι δημοσιεύσεις της Google το 2012 [3] και [4], παρουσίασαν μια νέα προσέγγιση στις σχεσιακές βάσεις δεδομένων, η οποία ενθυλακώθηκε στα θεμέλια των κατανεμημένων συστημάτων, και της μετεξέλιξης του παραδοσιακού σχεσιακού μοντέλου, τις Distributed SQL βάσεις δεδομένων. Εν αντιθέσει με τις NoSQL βάσεις δεδομένων, οι οποίες δεν επικεντρώνονται στην παροχή transactions (μτφρ. συναλλαγών), οι Distributed SQL σχεδιάζονται με γνώμονα την υποστήριξη ACID συναλλαγών, εξασφαλίζοντας ακεραιότητα και συνέπεια στα δεδομένα.

Σε αυτό το σημείο αξίζει να αναφερθεί, πως ο Distributed SQL είναι ένας πρόσφατος και ακόμη ρευστός όρος. Η αρχική ορολογία που χρησιμοποιείτο ήταν NewSQL, παρόλ' αυτά η ορολογία αυτή συμπεριλάμβανε βάσεις δεδομένων που επί της ουσίας ήταν ενδιάμεσο λογισμικό, που υλοποιούσαν εκ νέου sharding μεθόδους [5], και δεν ακολουθούσαν κάποια «καινούρια» αρχιτεκτονική, όπως αυτή που αποκαλούμε σήμερα Distributed SQL. Ως εκ τούτου, για να αναδείξουμε τις θεμελιώδεις διαφορές στις δυο αυτές αρχιτεκτονικές, στην εργασία αυτή θα χρησιμοποιούμε τον όρο Distributed SQL.

Επομένως, «Η **Distributed SQL** είναι μια ενιαία λογική βάση δεδομένων που αναπτύσσεται σε πολλαπλούς φυσικούς κόμβους, είτε σε ένα μόνο κέντρο δεδομένων είτε σε πολλά κέντρα δεδομένων κατανεμημένα σε όλο τον κόσμο, η οποία παρέχει εγγενώς ελαστική κλιμάκωση και ανθεκτικότητα» [6].

Η βάση δεδομένων που χρησιμοποιήσαμε

Οι πλέον δημοφιλέστερες Distributed SQL βάσεις δεδομένων οι οποίες εμπνεύστηκαν από την αρχιτεκτονική του Google Spanner και F1 που αναφέρθηκαν στην προηγούμενη ενότητα, είναι ανοικτού-κώδικα βάσεις δεδομένων, πράγμα το οποίο ήταν και κριτήριο για την επιλογή της βάσης δεδομένων που θα χρησιμοποιηθεί στη παρούσα εργασία. Μεταξύ λοιπόν των CockroachDB και YugabyteDB παρότι δεν μειονεκτούσε καμμία από τις δύο, όσον αφορά τα προσφερόμενα εργαλεία από αυτές, λόγω των επιδόσεων [7] που παρουσίασε η CockroachDB έναντι της YugabyteDB (τουλάχιστον για τις εκδόσεις v19.2.0. και v2.0.0. αντίστοιχα), επιλέχθηκε η CockroachDB. Κάτι που επίσης λήφθηκε υπόψιν είναι η επάρκεια των υπάρχοντων documentation και η κοινότητα των βάσεων δεδομένων. Στο σύνδεσμο¹ παραθέτω μία λίστα Distributed SQL βάσεων δεδομένων.

¹ Λίστα Distributed SQL βάσεων δεδομένων: https://en.wikipedia.org/wiki/Distributed_SQL

1.2 Η Αρχιτεκτονική της CockroachDB

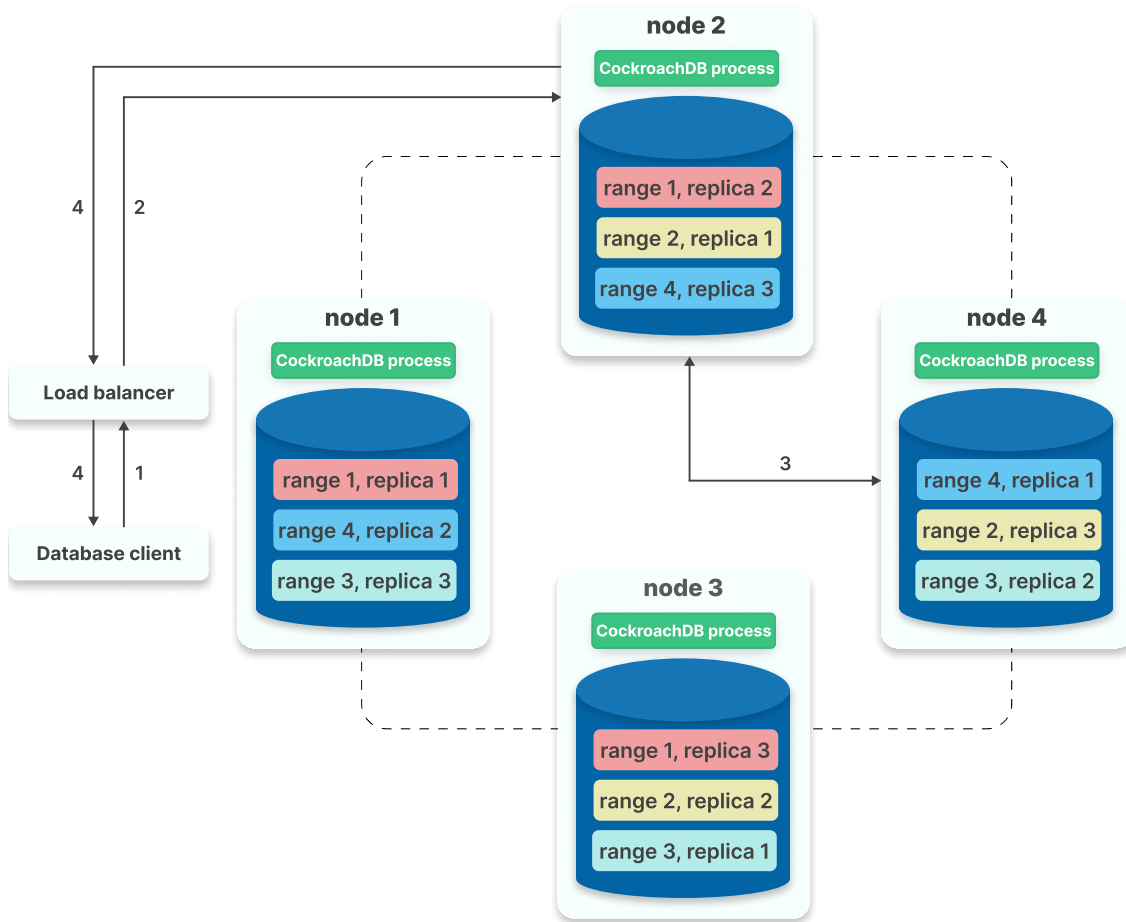
Η αρχιτεκτονική της CRDB αποτελείται από διάφορα επίπεδα που αλληλεπιδρούν με τα επίπεδα που βρίσκονται πάνω και κάτω από αυτά ως αδιαφανείς υπηρεσίες. Στην ενότητα αυτή θα κάνουμε μια επισκόπηση στην καθ' αυτή αρχιτεκτονική της βάσης δεδομένων, με κύριο σκοπό να παρουσιάσουμε τη ροή των δεδομένων σε αυτή, καθώς επίσης να παρουσιάσουμε και να εξηγήσουμε την ορολογία που χρησιμοποιείται. Στο υψηλότερο επίπεδο, η CRDB μετατρέπει τις εντολές SQL σε πράξεις κλειδιού-τιμής (KV), οι οποίες στη συνέχεια διανέμονται μεταξύ των κόμβων και εγγράφονται στο δίσκο.

1.2.1 Η αρχιτεκτονική του CockroachDB Cluster

Όπως περιγράφεται στο βιβλίο [8], μια υλοποίηση της CRDB αποτελείται από έναν ή και περισσότερους διακομιστές βάσης δεδομένων, οι οποίοι λειτουργούν απομακρυσμένα και ανεξάρτητα. Κάθε διακομιστής καλείται να διαχειριστεί το δικό του αποθηκευτικό χώρο, ακολουθώντας το γνωστό "shared-nothing" cluster μοντέλο. Οι κόμβοι μέσα στο cluster είναι συμμετρικοί και άρα δεν υπάρχει κάποιος «ειδικός» ή «κύριος» κόμβος. Τα δεδομένα κατανέμονται στο cluster με βάση τα εύρη κλειδιών (key ranges), τα οποία αντιγράφονται σε τουλάχιστον τρία μέλη του cluster.

Οι επικοινωνίες μεταξύ ενός διακομιστή και ενός χρήστη της βάσης δεδομένων πραγματοποιούνται μέσω του PostgreSQL wire πρωτοκόλλου. Σε μια πιο σύνθετη υλοποίηση, με σκοπό την διασφάλιση της ομοιόμορφης κατανομής αυτών των συνδέσεων σε όλους τους κόμβους του cluster, καθίσταται υπεύθυνος ένας (ή και περισσότεροι) load balancer · ο οποίος θα συνδέσει τον χρήστη με έναν από τους κόμβους του cluster, και έτσι ο κόμβος αυτός θα δράσει ως gateway server για την επικοινωνία αυτή. Ένα αίτημα του χρήστη ενδέχεται να απαιτεί ανάγνωση/εγγραφή από πολλαπλούς κόμβους. Έτσι, για κάθε εύρος KV, ορίζεται ένας leaseholder (μτφρ. ενοικιαστής) κόμβος ο οποίος ευθύνεται για τις αναγνώσεις/εγγραφές αυτού του εύρους.

Για να αποσαφηνιστούν τα προηγούμενα, στο σχήμα 1.1 απεικονίζεται σχηματικά η ροή αυτή. Ένας χρήστης της βάσης δεδομένων συνδέεται με έναν load balancer (1), ο οποίος επενεργεί ως proxy για το cluster αυτό. Ο load balancer προωθεί τα αιτήματα σε έναν διαθέσιμο κόμβο (2) και έτσι ο κόμβος αυτός γίνεται gateway κόμβος για αυτή την σύνδεση. Το συγκεκριμένο αίτημα απαιτεί δεδομένα για το εύρος 4, επομένως ο gateway κόμβος επικοινωνεί με τον leaseholder αυτού του εύρους (3), ο οποίος επιστρέφει τα απαιτούμενα δεδομένα στον gateway κόμβο, και αυτός με τη σειρά του στον χρήστη της βάσης δεδομένων (4) [8].



Σχήμα 1.1: Αρχιτεκτονική του CockroachDB cluster

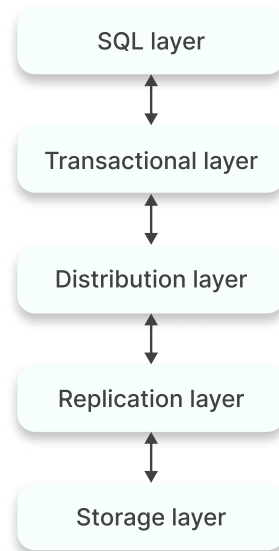
Κάθε κόμβος της CRDB, όπως φαίνεται και στο παραπάνω σχήμα, τρέχει ένα αντίγραφο του λογισμικού της βάσης δεδομένων, το οποίο είναι μια ενιαία πολυνηματική διεργασία. Όπως αναφέρθηκε προηγουμένως, κάθε διεργασία εσωτερικά είναι οργανωμένη σε πολλαπλά λογικά επίπεδα, όπως φαίνεται στο σχήμα 1.2.

1.2.2 Η στοίβα λογισμικού της CockroachDB

Στη συνέχεια θα περιγράψουμε τα επίπεδα που βρίσκονται στο σχήμα 1.2.

Το *SQL layer* είναι το τμήμα της στοίβας λογισμικού, το οποίο είναι υπεύθυνο για το χειρισμό των SQL αιτημάτων μέσω του PostgreSQL wire πρωτοκόλλου, και στη συνέχεια βελτιστοποιεί τα ερωτήματα και τα μετατρέπει σε KV πράξεις για να τις διαχειριστούν τα υποστρώματα.

Το *Transaction layer* είναι υπεύθυνο για τη διατήρηση της ατομικότητας των συναλλαγών, δηλαδή ότι όλες οι εργασίες σε μια συναλλαγή είτε εκτελούνται είτε διακόπτονται. Επίσης, στο επίπεδο αυτό εφαρμόζεται η διατήρηση της σειριακής απομόνωσης μεταξύ των συναλλαγών, και άρα διασφαλίζονται ορισμένες από τις ACID ιδιότητες που προσφέρονται από τη ΒΔ.



Σχήμα 1.2: Η στοίβα λογισμικού της CockroachDB

Ένας πίνακας αναπαρίσταται στην CockroachDB ως μια μονολιθική δομή KV, στην οποία το κλειδί είναι η συνένωση των πρωτευόντων κλειδιών του πίνακα και η τιμή είναι η συνένωση όλων των υπόλοιπων στηλών του πίνακα. Μια απλοϊκή αναπαράσταση είναι η εξής:

```
/<tableID>/<pkey1>/<pkey2>/<value1>/<value2>/<value3>/
```

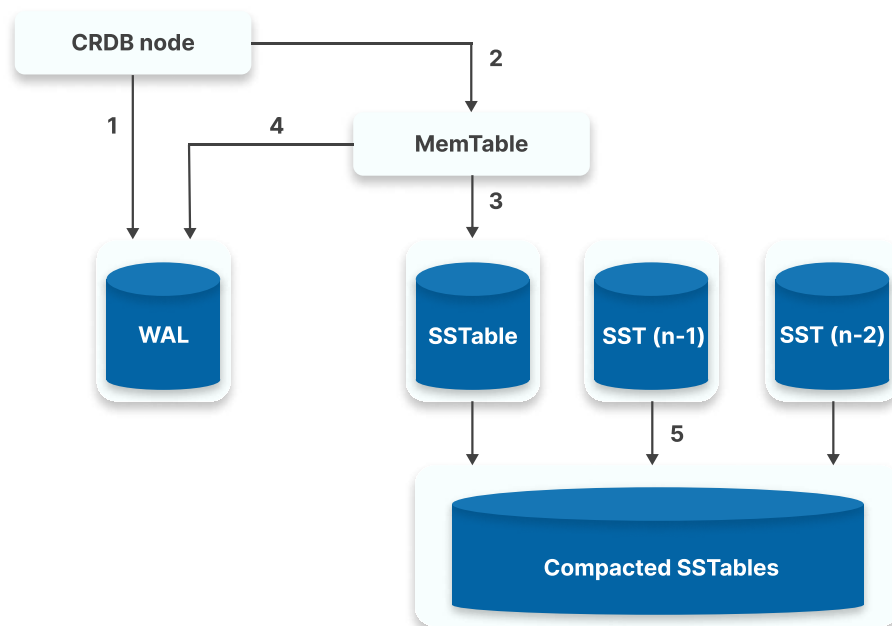
Το *Distribution layer* σπάει αυτή τη μονολιθική δομή σε τμήματα των ~512MB (το μέγεθος είναι τέτοιο ώστε να είναι διαχειρίσιμος ο αριθμός των ranges ανά κόμβο), και κατανέμει ομοιόμορφα τα δεδομένα στο cluster. Το *Replication layer* είναι ένα εξεζητημένο σύστημα, το οποίο εξασφαλίζει υψηλή διαθεσιμότητα και ανοχή σε σφάλματα. Χρησιμοποιεί ένα κατανεμημένο μοντέλο consensus (μτρφ. συναίνεση), ονόματι multi-active, το οποίο συνενώνει στοιχεία δύο αρχιτεκτονικών, των active-active και active-passive, σύμφωνα με το οποίο κάθε αντίγραφο έχει τη δυνατότητα διαχείρισης δεδομένων και αιτημάτων. Ωστόσο, τυχόν ενημερώσεις ή αλλαγές που εφαρμόζονται στη βάση δεδομένων, πρέπει πρώτα να λάβουν έγκριση από την πλειοψηφία των αντιγράφων ώστε να πραγματοποιηθούν. Η αντιγραφή βασίζεται στο πρωτόκολλο Raft, μέσω του οποίου το κατανεμημένο σύστημα υποστηρίζει υψηλή διαθεσιμότητα και ελαστική κλιμάκωση.

Το τελευταίο στη στοίβα επίπεδο είναι το *Storage layer*, το οποίο υλοποιεί τη φυσική αποθήκευση των KV τιμών. Από την έκδοση 20.0 κι έπειτα της CRDB, έχει σχεδιαστεί και ενσωματωθεί η μηχανή αποθήκευσης PebbleDB, αφορμώμενη από τις RocksDB και LevelDB. Η PebbleDB χρησιμοποιεί ένα LSM δέντρο, μία δομή που υποστηρίζει υψηλούς ρυθμούς εισαγωγής, διατηρώντας παράλληλα γρήγορη τυχαία πρόσβαση ανάγνωσης. Η βάση του δέντρου αποτελείται από δύο indexed “trees”:

- το MemTable, ένα δέντρο στη μνήμη το οποίο αποδέχεται όλες τις νέες εγγραφές

- τα SSTables, ένα πλήθος δέντρων στο δίσκο, τα οποία είναι δεδομένα που έχουν μεταφερθεί από την μνήμη.

Τα SSTs στοιβάζονται σε έξι επίπεδα, από το L0 έως το L6. Στο πρώτο επίπεδο (L0) εμπεριέχονται SSTs τα οποία είναι δέντρα που έχουν μεταφερθεί από το MemTable. Στη συνέχεια, σταδιακά, τα SSTs συμπύσσονται και καθώς μεγαλώνουν προχωρούν σε παρακάτω επίπεδα, μόνο που σε αυτή την περίπτωση τα δέντρα αυτά ταξινομούνται και δεικτοδοτούνται ώστε να εκτελούνται γρήγορα τα lookups. Φυσικά εάν ένας κόμβος «αποτύχει» ενώ τα δεδομένα βρίσκονται ακόμα στο MemTable, δηλαδή στην μνήμη, τότε ενδέχεται να χαθούν. Για αυτό το λόγο στη δομή αυτή περιλαμβάνεται ένα Write Ahead Log που διατηρεί τις συναλλαγές στον δίσκο.



Σχήμα 1.3: Παράδειγμα εγγραφής σε LSM δέντρο

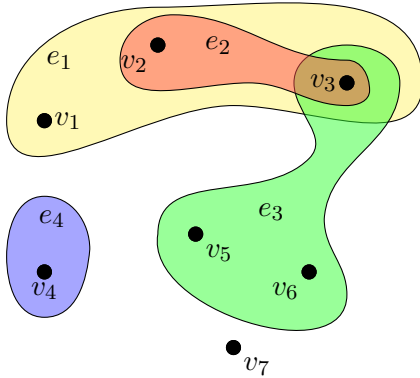
Στο σχήμα 1.3 περιγράφεται μία εγγραφή σε LSM δέντρο της PebbleDB. Αρχικά οι εγγραφές από το παραπάνω επίπεδο της στοίβας λογισμικού αποθηκεύονται στο WAL (1) και **μετά** στο MemTable (2). Τη στιγμή που το MemTable φτάσει ένα συγκεκριμένο μέγεθος, τα δεδομένα μεταφέρονται στον δίσκο και δημιουργείται ένα καινούριο SST (3). Όταν τα δεδομένα μεταφερθούν επιτυχώς στο δίσκο, τα δεδομένα που βρίσκονται στο WAL διαγράφονται (4). Τελικά τα SSTs συμπύσσονται σε μεγαλύτερα SSTs (5).

1.3 Ιστορικά Γραφήματα και Hinode

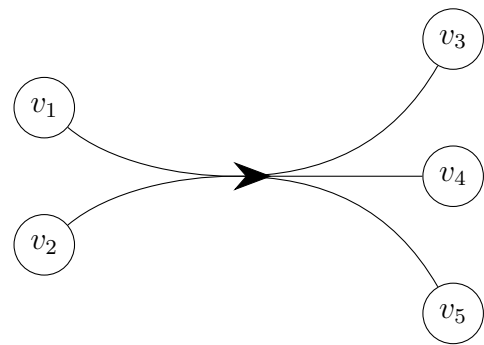
Τα γραφήματα παρέχουν και μοντελοποιούν ένα φυσικό πλαίσιο για την αναπαράσταση των συνδέσεων και της δυναμικής μεταξύ διαφορετικών οντοτήτων. Πολλά γραφήματα που εξομοιώνουν τα πραγματικά δίκτυα, εφόσον εξ' ορισμού είναι δυναμικά², εξελίσσονται με τη πάροδο του χρόνου. Στα γραφήματα αυτά δημιουργούνται νέες ακμές και παράλληλα εισέρχονται στο δίκτυο καινούριοι κόμβοι, ενώ αντίστοιχα υπάρχουν ακμές και κόμβοι, με τη πάροδο του χρόνου ενδεχομένως εγκαταλείπουν το δίκτυο ή αλλάζουν.

Ένα γράφημα συνήθως αναπαρίσταται ως $G = (V, E)$, όπου V είναι ένα σύνολο κορυφών και $E \subseteq V \times V$ είναι ένα σύνολο ακμών. Τα γραφήματα μπορούν να κατηγοριοποιηθούν σε δύο ευρύτερες κατηγορίες, τα κατευθυνόμενα και τα μη-κατευθυνόμενα. Στα μη-κατευθυνόμενα γραφήματα οι ακμές παρουσιάζουν συμμετρικές σχέσεις που σημαίνει ότι μία ακμή (u, v) είναι ουσιαστικά ίδια με την (v, u) . Αντίθετα, τα κατευθυνόμενα γραφήματα εισάγουν την έννοια της κατευθυντικότητας, όπου μια ακμή (u, v) υποδηλώνει ότι υπάρχει σύνδεση από την κορυφή u προς την κορυφή v . Επιπλέον, τα γραφήματα μπορούν να σχηματίσουν πολυγράφους, επιτρέποντας έτσι πολλαπλές ακμές μεταξύ δύο κορυφών, πράγμα το οποίο είναι πολύ σημαντικό στην μοντελοποίηση σεναρίων όπου μπορεί να υπάρχουν πολλαπλές σχέσεις ή αλληλεπιδράσεις μεταξύ οντοτήτων.

Πέραν όμως της δομικής αναπαράστασης που διέπει τα γραφήματα, μπορούν να ενσωματωθούν σε αυτά πρόσθετες πληροφορίες μέσω ετικετών που σχετίζονται με κορυφές ή/και ακμές. Τα γραφήματα με βάρη είναι άλλη μία σημαντική περίπτωση κατά την οποία αποδίδονται στις ακμές αριθμητικές τιμές, που συχνά αντιπροσωπεύουν κόστη ή αποστάσεις. Για λόγους πληρότητας αλλά και όχι μόνο, σε αυτό το σημείο αξίζει να αναφερθούν οι υπεργράφοι, μια γενίκευση των γραφημάτων, στους οποίους μια ακμή μπορεί να συνδεθεί με οσοδήποτε κορυφές. Παραδείγματα υπεργράφων βρίσκονται στα παρακάτω σχήματα 1.4 και 1.5.



Σχήμα 1.4: Μη-κατευθυνόμενος υπεργράφος



Σχήμα 1.5: Κατευθυνόμενος υπεργράφος

²τὰ πάντα ρεῖ, μηδέποτε κατὰ τ' αὐτὸ μένειν

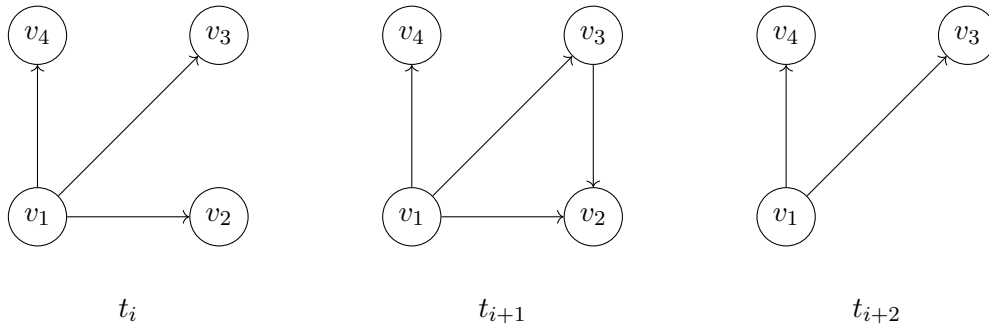
1.3.1 Τα ιστορικά γραφήματα

Όπως αναφέρθηκε προηγουμένως, τα γραφήματα υπόκεινται σε εξέλιξη με την πάροδο του χρόνου και ενημερώνονται · όχι όμως απαραίτητα μονάχα δομικά, αλλά και στα περιεχόμενα που σχετίζονται με αυτά, σχηματίζοντας έτσι ιστορικά γραφήματα που καταγράφουν τις μεταβολές σε ένα γραμμικά διατεταγμένα και διακριτό χρονικό πεδίο.

Αναφερόμαστε στο στιγμιότυπο ενός γραφήματος τη χρονική στιγμή t ως την κατάσταση που βρίσκεται το γράφημα ακριβώς εκείνη την χρονική στιγμή, όσον αφορά τις κορυφές, τις ακμές και τα περιεχόμενα του γραφήματος. Έχοντας αναφέρει όλα αυτά,

Ορισμός 1.1 (Ιστορικό Γράφημα). Ένα ιστορικό γράφημα $G_{[t_i, t_j]}$ στο χρονικό διάστημα $[t_i, t_j]$ είναι μία ακολουθία $\{G_{t_i}, G_{t_i+1}, \dots, G_{t_j}\}$ των στιγμιότυπων του γραφήματος. [9]

Στην παρακάτω εικόνα παρατίθενται τρία στιγμιότυπα από ένα ιστορικό γράφημα $G = (V, E)$. Στη χρονική στιγμή t_{i+1} έχει προστεθεί η ακμή (v_3, v_2) , ενώ στη χρονική στιγμή t_{i+2} έχει αφαιρεθεί η κορυφή v_2 καθώς επίσης και όλες οι σχετικές ακμές.



Σχήμα 1.6: Στιγμιότυπα από ιστορικό γράφημα

1.3.2 Μοντέλα Ιστορικών Γραφημάτων

Μία σημαντική πρόκληση που θέτει η φύση αυτών των γραφημάτων, και άρα των αντίστοιχων δικτύων τους, είναι η αποτελεσματική διαχείριση και αποθήκευση του ιστορικού τους. Οι πρώιμες προσεγγίσεις αποθήκευσης τέτοιων γραφημάτων ήταν η COPY και η LOG. Στην πρώτη προσέγγιση αποθηκεύεται κάθε στιγμιότυπο της ακολουθίας ως ξεχωριστό γράφημα, πράγμα το οποίο σημαίνει πως οποιοδήποτε στιγμιότυπο είναι άμεσα διαθέσιμο. Παρόλ' αυτά όπως είναι αναμενόμενο αυτός ο τρόπος αποθήκευσης, για πραγματικά μεγάλα γραφήματα, απαιτεί πολλούς αποθηκευτικούς πόρους. Το μοντέλο LOG έχει γνώμονα τις μεταβολές μεταξύ διαδοχικών στιγμιότυπων του γραφήματος, οι οποίες ονομάζονται δέλτα. Ένα σύστημα που διατηρεί ένα σύνολο από αυτές τις μεταβολές, είναι σε θέση να ανακατασκευάσει οποιοδήποτε στιγμιότυπο του ιστορικού γραφήματος εφαρμόζοντας σε ένα αρχικό στιγμιότυπο όλα τα δέλτα διαδοχικά, μέχρι την επιθυμητή χρονική στιγμή. Η μέθοδος αυτή εκμεταλλεύεται τα διαδοχικά παρόμοια στιγμιότυπα του γραφήματος, εφόσον αποθηκεύει (σχεδόν) μόνο τα δέλτα, αποφεύγει τον πλεονάζοντα αποθηκευτικό χώρο που καταλαμβάνει η COPY.

Οι περισσότερες έρευνες που έχουν διενεργηθεί μέχρι στιγμής [10], [11], χρησιμοποιούν μια υβριδική προσέγγιση των δυο προαναφερθέντων, που ονομάζεται COPY + LOG. Στη προκειμένη μέθοδο, αντί της αποθήκευσης ενός μόνο στιγμιότυπου, αποθηκεύεται ένα σύνολο επιλεγμένων στιγμιότυπων του γραφήματος, μαζί με τα δέλτα που τα συσχετίζουν. Κατά την κατασκευή ενός συγκεκριμένου στιγμιότυπου που δεν περιλαμβάνεται στο επιλεχθέν σύνολο, επιλέγεται ένα στιγμιότυπο από αυτό και εφαρμόζονται τα αντίστοιχα δέλτα. Ακολουθώντας αυτή τη τακτική, είναι επίσης δυνατή η αντιστροφή ενός δέλτα, του οποίου η εφαρμογή θα παράξει ένα προηγούμενο στιγμιότυπο.

Κατά τη σχεδίαση τέτοιων μοντέλων, πρέπει να ληφθούν υπόψιν οι τύποι των ερωτημάτων που επιδέχονται τα ιστορικά γραφήματα. Τα ερωτήματα αυτά μπορούν να κατηγοριοποιηθούν σε δύο κλάσεις, τα τοπικά ερωτήματα (1_a), κατά τα οποία απαιτείται πληροφορία από μία κορυφή και τα καθολικά ερωτήματα (1_b) στα οποία απαιτείται πληροφορία από όλο το γράφημα. Ασχέτως με το πλήθος των κορυφών που υπάρχουν στο γράφημα, ένα ερώτημα ενδέχεται να απαιτεί πληροφορίες είτε από μόνο ένα στιγμιότυπο (2_a), είτε από πολλά περισσότερα (2_b). Τα συστήματα που υποστηρίζουν τα γραφήματα αυτά, θα πρέπει να υποστηρίζουν όλους τους δυνατούς συνδυασμούς των (1) με των (2) ερωτημάτων. Ένα παράδειγμα τοπικού ερωτήματος επί ενός στιγμιότυπου μιας κορυφής είναι «*Ποια ήταν η 1-hop γειτονιά της κορυφής v τη χρονική στιγμή t ;*», αντίστοιχα ένα παράδειγμα καθολικού ερωτήματος επί ολόκληρου του ιστορικού του γραφήματος είναι «*Βρες το μέσο όρο του βαθμού των κορυφών του γραφήματος για κάθε χρονιά*».

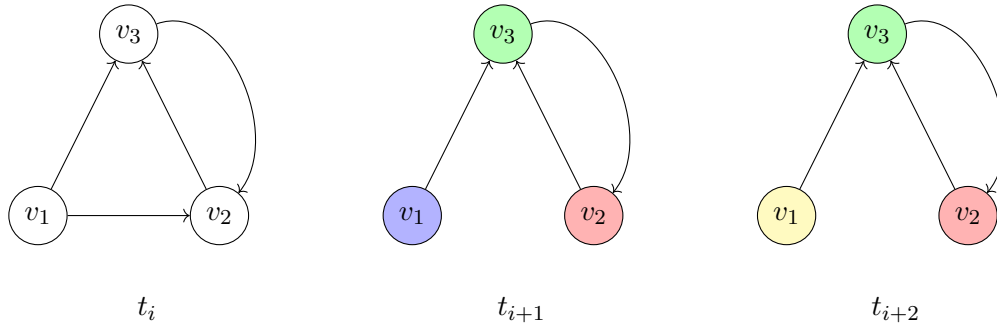
Η βελτίωση των επιδόσεων ενός συστήματος μπορεί να επιτευχθεί σε μεγάλο βαθμό, αναπτύσσοντας ένα μηχανισμό δεικτοδότησης (indexing) [12] και αποθήκευσης, εχμεταλλευτόμενο την παρουσία αμετάβλητων κορυφών και ακμών μεταξύ διαδοχικών στιγμιότυπων. Επιπλέον, δεδομένου του μεγέθους των γραφημάτων, κατά τη σχεδίαση αυτή του μηχανισμού πρέπει να ληφθεί υπόψιν το γεγονός ότι η πλειονότητα των δεδομένων του γραφήματος, πρέπει να αποθηκευτεί στο δίσκο. Ως εκ τούτου, ο αποτελεσματικός και στοχευμένος σχεδιασμός των επιμέρους στοιχείων του μηχανισμού αποτελεί καθοριστικό βήμα στην ελαχιστοποίηση των ανεπιθύμητων συχνών προσβάσεων στον δίσκο κατά την επεξεργασία των ιστορικών γραφημάτων.

Οι δύο κυριότερες προσεγγίσεις που έχουν αναπτυχθεί στο σχεδιασμό αυτών των συστημάτων, είναι το χρονοκεντρικό και το οντοκεντρικό μοντέλο. Τα χρονοκεντρικά συστήματα μπορούν να σχεδιαστούν βάσει των στιγμιότυπων τους αλλά και γενικότερων χρονικών παραμέτρων. Κατά βάση τα δεδομένα οργανώνονται σύμφωνα με το κάθε στιγμιότυπο στο οποίο ανήκουν, πράγμα το οποίο διευκολύνει την εξαγωγή ενός στιγμιότυπου χωρίς να χρειαστεί ανακατασκευή για αυτό, καθώς επίσης και την ταχύτερη εκτέλεση πράξεων σε γειτονικά στιγμιότυπα. Ωστόσο κατά την αποθήκευση μεγάλων γραφημάτων τα οποία έχουν πολλά δέλτα, αναπόφευκτα, θα αποθηκευτούν ακριβώς τα ίδια δεδομένα πολλαπλές φορές, χρησιμοποιώντας πλεονάζοντα αποθηκευτικό χώρο. Στα οντοκεντρικά συστήματα τα δεδομένα του γραφήματος αποθηκεύονται βάσει του κόμβου στον οποίο ανήκουν, πράγμα το οποίο έχει ως αποτέλεσμα την κατά το πλείστον μείωση της αποθήκευσης ίδιων δεδομένων.

Το αποθηκευτικό μοντέλο **HiNode**³ που προτείνεται από τους συγγραφείς του [13], διαφοροποιείται σε σχέση με τα προαναφερθέντα μοντέλα, δηλαδή τόσο στα χρονοκεντρικά όσο και στα αυτά τα οποία χρησιμοποιούν τα δέλτα ανακατασκευάζοντας τα στιγμιότυπα. Το μοντέλο αυτό αφορμάται από το οντοκεντρικό μοντέλο, αλλά πιο συγκεκριμένα, το προσεγγίζει με μια κορυφοκεντρική⁴ οπτική.

Σύμφωνα με την προσέγγιση αυτή, κάθε κορυφή του γραφήματος συνδέεται άμεσα με το ιστορικό της, αποθηκεύοντάς το εσωτερικά της κορυφής, η οποία είναι δομημένη με τέτοιο τρόπο ώστε να υποστηρίζονται διάφορα ερωτήματα.

Στην παρακάτω εικόνα τα τρία στιγμιότυπα ανήκουν σε ένα κορυφοκεντρικό ιστορικό γράφημα $G = (V, E, I)$ το οποίο αποθηκεύει πληροφορία στις κορυφές, όπου $I : V \rightarrow \Sigma$ είναι μία συνάρτηση η οποία αναθέτει σε μία κορυφή V πληροφορία από ένα σύνολο Σ . Για το συγκεκριμένο παράδειγμα υποθέτουμε ότι η πληροφορία που είναι αποθηκευμένη στη κορυφή είναι το χρώμα της. Τη χρονική στιγμή t_{i+1} αποθηκεύεται πληροφορία στις κορυφές, καθώς επίσης έχει αφαιρεθεί η ακμή (v_1, v_2) , ενώ τη χρονική στιγμή t_{i+2} ενημερώνεται η πληροφορία που βρίσκεται στην κορυφή v_1 .



Σχήμα 1.7: Ιστορικό γράφημα που αποθηκεύει πληροφορία στις κορυφές

³Από το “history of (diachronic) node”

⁴Μετάφραση του vertex-centric

1.3.3 Η δομή του HiNode

Μία κορυφή v ενός συγκεκριμένου στιγμιότυπου G_i του χρονικού γραφήματος, ορίζεται από ένα σύνολο χαρακτηριστικών (attributes) (i), ένα σύνολο εισερχόμενων ακμών (ii) που προέρχονται από άλλες κορυφές του G_i και ένα σύνολο εξερχόμενων ακμών (iii) προς άλλες κορυφές του G_i . Με σκοπό την παρακολούθηση της «διάρκειας ζωής» των κορυφών του γραφήματος, χρησιμοποιείται ένα εξωτερικό **δέντρο αποστάσεων**⁵ $\mathcal{I}$, το οποίο διατηρεί διαστήματα $\{T_{t_s, t_e}^v\}$, που υποδηλώνουν το χρονικό διάστημα ύπαρξης μιας κορυφής, δηλαδή από την χρονική στιγμή t_s έως και την t_e . Μπορούμε να ορίσουμε μια κορυφή να είναι «ενεργή» μέχρι και την πιο πρόσφατη χρονική στιγμή, θέτοντάς της την τιμή $+\infty$, στην χρονική στιγμή t_e του διαστήματός της. Επιπρόσθετα, κάθε διάστημα στο $\mathcal{I}$ συνδέεται με έναν **διαχρονικό κόμβο** $\mathcal{D}_v$, ο οποίος είναι μια δομή δεδομένων που περιέχει ένα ολοκληρωμένο ιστορικό των μεταβολών που έχει υποστεί η κορυφή v σε ολόκληρη την ακολουθία στιγμιότυπων του G .

Ο διαχρονικός κόμβος $\mathcal{D}_v$, διατηρεί ένα εξωτερικό δέντρο διαστημάτων $\mathcal{I}_v$, το οποίο αποτυπώνει όλη τη πληροφορία της κορυφής v (attributes και ακμές) καθ' όλη την χρονική διάρκεια του G . Ένα διάστημα $\mathcal{I}_v$ αποθηκεύεται ως την εξής τετράδα $(f, \{l_1, l_2, \dots\}, t_s, t_e)$, όπου f αναπαριστά το attribute της κορυφής που λαμβάνει τις τιμές $l_{1,2,\dots}$ κατά τη χρονική διάρκεια $[t_s, t_e]$. Είναι σημαντικό πως μπορούν να αποθηκευτούν ακριβώς με τον ίδιο τρόπο και οι ακμές που συνδέονται με την κορυφή v . Επιπλέον, ο διαχρονικός κόμβος διατηρεί ένα B-δέντρο⁶ για κάθε attribute και για κάθε μεμονωμένη ακμή της κορυφής, μέσω του οποίου μπορούμε να αναζητήσουμε αποδοτικά τις ενδιαφερόμενες τιμές για συγκεκριμένα χρονικά διαστήματα, εξοικονομώντας χώρο και χρόνο. Περαιτέρω ανάλυση βρίσκεται στο [13].

⁵Interval tree: https://en.wikipedia.org/wiki/Interval_tree

⁶B-tree: <https://en.wikipedia.org/wiki/B-tree>

Κεφάλαιο 2

Υλοποίηση Μέρος Α

Στο κεφάλαιο αυτό αναφέρονται και αναλύονται τα εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του πρώτου μέρους της παρούσας εργασίας, καθώς επίσης εμβαθύνει στην σχεδίαση των επιμέρους υλοποιήσεων. Πιο συγκεκριμένα το κεφάλαιο αυτό καλύπτει τη σχεδίαση και υλοποίηση μιας προσαρμοσμένης βάσης δεδομένων, η οποία στηρίζεται στα χαρακτηριστικά του HiNode που αναφέρθηκε στο προηγούμενο κεφάλαιο. Ο κώδικας που αναπτύχθηκε κατά την εκπόνηση της διπλωματικής εργασίας βρίσκεται στον ιστότοπο¹.

2.1 Go

Η γλώσσα προγραμματισμού που χρησιμοποιήθηκε για την υλοποίηση της εργασίας είναι η Go, ή συχνά αναφερόμενη και ως Golang, η οποία σχεδιάστηκε² από την Google το 2007 και έγινε δημόσια διαθέσιμη το 2009. Η Go είναι μία statically typed και compiled γλώσσα προγραμματισμού, η οποία συνδέει τα συντακτικά χαρακτηριστικά της C, με ασφάλεια μνήμης και garbage collection. Επιπλέον, υποστηρίζει concurrency τύπου CSP, παρέχοντας έτσι ένα περιβάλλον προγραμματισμού που συνδυάζει την αποτελεσματικότητα με την ασφάλεια μνήμης και την ταυτόχρονη εκτέλεση εργασιών.

Στη συνέχεια παρατίθενται μερικά κομμάτια κώδικα τα οποία συνηθίζονται κατά την ανάπτυξη κώδικα με την Go, αλλά και χρησιμοποιούνται στην υλοποίηση της παρούσας διπλωματικής εργασίας.

Οι συναρτήσεις και οι μέθοδοι μπορούν να επιστρέφουν πολλαπλές τιμές και συνήθως συνδυάζεται με ένα ζευγάρι τιμών result και err, με τη διαχείριση του σφάλματος να γίνεται αμέσως μετά την κλήση της συνάρτησης, όπως φαίνεται στον παρακάτω κώδικα.

```
1 result, err := foo()
2 if err != nil {
3     log.Fatal("Error:", err)
4 }
```

Κώδικας 2.1: Επιστροφή τιμών και διαχείριση σφάλματος

¹HiNode CockroachDB: <https://github.com/xariskap/HiNode-CockroachDB>

²Οι σχεδιαστές της παρακινήθηκαν κυρίως από την κοινή τους απέχθεια προς τη C++

Στον κώδικα παρακάτω υπάρχουν οι δηλώσεις για ένα struct (αφού η Go δεν είναι αντικειμενοστραφής γλώσσα προγραμματισμού), μια συνάρτηση που δημιουργεί structs και μια μέθοδο η οποία θέτει τιμή σε ένα πεδίο του struct. Το συντακτικό (`p *Person`) στη γραμμή 10, ονομάζεται *method receiver*, μέσω του οποίου μας επιτρέπεται να συνδέσουμε την εκάστοτε μέθοδο με το struct, και πιο συγκεκριμένα στο παράδειγμα αυτό, με έναν δείκτη σε ένα struct `Person`. Ο receiver συνηθίζεται να έχει ένα μικρό όνομα μεταβλητής, όπως και στο παρόν έχουμε το `p`.

```
1 type Person struct {
2     Name string
3     Age  int
4 }
5
6 func NewPerson(name string, age int) *Person {
7     return &Person{Name: name, Age: age}
8 }
9
10 func (p *Person) SetAge(age int) {
11     p.Age = age
12 }
```

Κώδικας 2.2: Δημιουργία struct και μεθόδου του

2.1.1 Goroutines

Οι goroutines είναι ένα κεντρικό χαρακτηριστικό της Go που επιτρέπει την **ταυτόχρονη** (όχι απαραίτητα παράλληλη) εκτέλεση κώδικα. Οι goroutines συχνά αναφέρονται και ως «ελαφριά νήματα» λόγω του ότι απαιτούνται λιγότεροι πόροι συγκρίσει των παραδοσιακών νημάτων ή των διεργασιών του λειτουργικού συστήματος, όσον αφορά τη χρησιμοποίηση της μνήμης. Η δημιουργία μιας goroutine γίνεται πολύ εύκολα, τοποθετώντας τη λέξη *go* πριν τη κλήση μιας συνάρτησης. Με αυτό τον τρόπο οι συναρτήσεις τρέχουν ταυτόχρονα με το κύριο πρόγραμμα ή άλλες τέτοιες ρουτίνες. Οι διαχείριση αυτών γίνεται από τον Go runtime scheduler, ο οποίος τις χρονοπρογραμματίζει αποτελεσματικά και τις πολυπλέκει σε ένα μικρό αριθμό νημάτων του λειτουργικού συστήματος, που συνήθως είναι ίσο με τον αριθμό των διαθέσιμων πυρήνων της CPU. Η διάρκεια ζωής μιας goroutine είναι άμεσα συνδεδεμένη με την διάρκεια εκτέλεσης του προγράμματος, πράγμα το οποίο σημαίνει πως όταν η συνάρτηση `main` τερματίσει, τότε τερματίζονται και όλες οι goroutines ακόμη και αν δεν έχουν ολοκληρώσει τον κώδικα προς εκτέλεση.

Ένα τρόπος επικοινωνίας και συγχρονισμού μεταξύ των αυτών των ρουτινών είναι τα *κανάλια*, τα οποία παρέχουν έναν ασφαλή τρόπο για αποστολή και λήψη δεδομένων μεταξύ goroutines. Έστω `chan` ένα κανάλι το οποίο στέλνει και λαμβάνει boolean τιμές. Το συντακτικό `chan <- true` μας επιτρέπει να στείλουμε τη τιμή `true` μέσω του καναλιού, ενώ το συντακτικό `x <- chan`, μας επιτρέπει να αποθηκεύσουμε στη μεταβλητή `x` ό,τι στάλθηκε από το κανάλι `chan`. Στον κώδικα 2.3 βλέπουμε όσα έχουν περιγραφεί παραπάνω.

```
1 func worker(done chan bool) {
2     fmt.Println("working...")
3     time.Sleep(time.Second)
4     fmt.Println("done")
5
6     // Send a value to notify that we're done.
7     done <- true
8 }
9
10 func main() {
11     var x boolean
12
13     // Start a worker goroutine, giving it the channel to notify on.
14     done := make(chan bool, 1)
15     go worker(done)
16
17     // Block until we receive a notification from the worker on the channel.
18     x <-done
19 }
```

Κώδικας 2.3: Goroutines και channels

Ακόμη, θα ήθελα να αναφερθώ σε δυο πολύ σημαντικές λειτουργίες του πακέτου *sync*, οι οποίες αποτελούν κρίσιμο μέσο για την επίτευξη συγχρονισμού και ασφάλειας σε πολυνηματικά περιβάλλοντα. Μέσω των *sync.Mutex* μπορούμε να υλοποιήσουμε την αμοιβαία αποκλειστική πρόσβαση (mutal exclusion) σε κρίσιμα τμήματα κώδικα, διασφαλίζοντας ότι μόνο μια goroutine μπορεί να έχει πρόσβαση σε κοινόχρηστους πόρους.

Τέλος, τα *sync.WaitGroup* αποτελούν έναν τρόπο να διαχειριζόμαστε τον τερματισμό πολλών goroutine και να εξασφαλίζουμε ότι η εκτέλεση του προγράμματος δεν θα συνεχίσει πριν ολοκληρωθούν ορισμένες λειτουργίες.

```
1 func incrementCounter(wg *sync.WaitGroup) {
2     mutex.Lock() // Lock the Mutex before incrementing the counter
3     defer mutex.Unlock() // Unlock the Mutex even if an error occurs
4     defer wg.Done() // Decrease the WaitGroup counter when the goroutine
        completes
5
6     counter++
7 }
8
9 func main() {
10     var wg sync.WaitGroup
11     for i := 0; i < 10; i++ {
12         wg.Add(1) // Add a wait for each goroutine to the WaitGroup
13         go incrementCounter(&wg)
14     }
15     wg.Wait() // Wait for all goroutines to complete
16 }
```

Κώδικας 2.4: Mutex και WaitGroup

2.1.2 pgx

Εφόσον είχε επιλεγεί η Go ως γλώσσα προγραμματισμού για την ανάπτυξη κώδικα, μια εναλλακτική για την διασύνδεση με την CockroachDB ήταν το πακέτο GORM³. Το πακέτο αυτό είναι μια ORM βιβλιοθήκη, δηλαδή ένας μεσάζοντας που γεφυρώνει τα αντικειμενοστραφή προγράμματα, με τις σχεσιακές βάσεις δεδομένων. Παρόλο που κατά την ανάπτυξη κώδικα αυτά τα εργαλεία φαίνεται να είναι εύλογα, εν προκειμένω έπρεπε να αποφευχθεί η χρήση τους, καθώς σε περίπλοκα ερωτήματα στη βάση δεδομένων φέρουν χειρότερα χρονικά αποτελέσματα έναντι των εργαλείων που χρησιμοποιούν SQL.

Επομένως, στη παρούσα εργασία χρησιμοποιήθηκε το πακέτο pgx⁴, το οποίο είναι ένας PostgreSQL Driver. Με αυτόν το τρόπο θα μπορούσαμε να αξιοποιήσουμε τις δυνατότητες που μας παρέχει η CockroachDB, εφόσον και αυτή είναι πλήρως συμβατή με το PostgreSQL wire πρωτόκολλο. Επιπλέον, χρησιμοποιώντας αυτό το πακέτο, πέραν των δυνητικά ταχύτερων αποκρίσεων από άποψη χρόνου, θα είχαμε και πολύ περισσότερο έλεγχο και διαφάνεια πάνω στη βάση δεδομένων.

2.2 Βασικά Στοιχεία Υλοποίησης

Σε αυτήν την ενότητα θα μελετήσουμε και θα εξηγήσουμε τα κομμάτια κώδικα της υλοποίησης της εργασίας. Όπως φαίνεται και στην εικόνα 2.1, στο directory υπάρχουν τέσσερις φάκελοι και το αρχείο main.go, στο οποίο βρίσκεται η συνάρτηση main του προγράμματος. Με τον φάκελο gremlin θα ασχοληθούμε στο επόμενο κεφάλαιο.

Ένα γενικότερο σχόλιο πάνω στην υλοποίηση είναι ότι προσπάθησε να αποκλίνει από τις δύο προηγούμενες, δηλαδή αυτή της Cassandra και της MongoDB [14] [15], αφενός για την αξιολόγηση παρόμοιων σχεδιασμών με διαφορετικά εργαλεία, αφετέρου με σκοπό τη βελτίωση και την ανανέωση των προϋπαρχόντων υλοποιήσεων.

Στον φάκελο utils, βρίσκονται οι εκ νέου υλοποιήσεις των βασικών δομών που χρησιμοποιούνται από τα γραφήματα που υποστηρίζει το HiNode, καθώς επίσης και ένα ακόμη αρχείο, το utils.go, στο οποίο περιέχονται βοηθητικές συναρτήσεις. Οι δομές που δημιουργήθηκαν δεν χρησιμοποιήθηκαν περαιτέρω στην εργασία αυτή, παρόλ' αυτά έχουν ολοκληρωθεί για μετέπειτα εργασίες.

2.2.1 Διασύνδεση

Όσον αφορά την διασύνδεση του προγράμματος με τη βάση δεδομένων, στο αρχείο db.go του φακέλου db έχουν δημιουργηθεί συναρτήσεις, μέσω των οποίων μπορούμε να επενεργήσουμε πάνω στις βάσεις δεδομένων. Η εκκίνηση του cockroach cluster με έναν κόμβο γίνεται από τη γραμμή εντολών με την εντολή 'cockroach start-single-node --insecure'. Το flag --insecure χρησιμοποιείται **μόνο** σε περιβάλλοντα δοκιμής και ανάπτυξης εφαρμογών δι-

³GORM: <https://gorm.io/>

⁴pgx: <https://github.com/jackc/pgx>

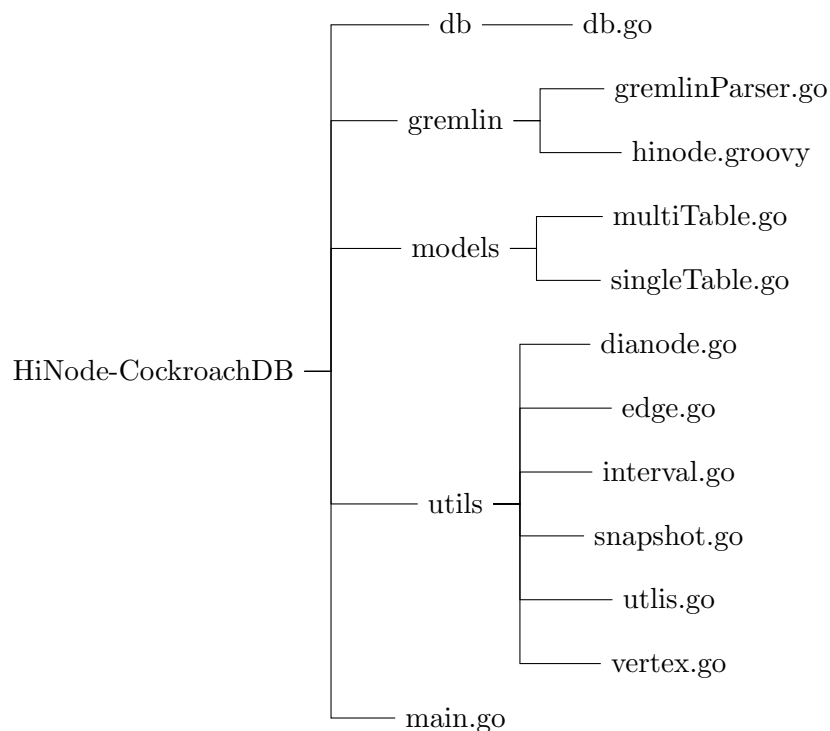
ότι χρησιμοποιώντας αυτό, παραλείπεται η χρήση πιστοποιητικών και επαλήθευσης για την προστασία των δεδομένων και της επικοινωνίας στο σύστημα της βάσης δεδομένων.

Αφού το cluster είναι ενεργό, καλώντας στην main τις συναρτήσεις του αρχείου db.go, μπορούμε είτε να δημιουργήσουμε μια καινούρια βάση δεδομένων είτε να συνδεθούμε με μια υπάρχουσα. Στο παρακάτω παράδειγμα φαίνονται οι λειτουργίες αυτές, με τις συναρτήσεις Create@Model() και USE@(), όπου @ το εκάστοτε μοντέλο, τα οποία θα αναλυθούν στην επόμενη ενότητα.

```

1 func main() {
2     var dbConn = db.GetConnection()
3     var dbConnST = db.GetConnection()
4
5     defer dbConn.Close(context.Background())
6     defer dbConnST.Close(context.Background())
7
8     mt := db.CreateMtModel("sf3", dbConn) // creating a database
9     st := db.USEst("st_sf10", dbConnST) // connect to an existing database
10 }
```

Κώδικας 2.5: Δημιουργία και σύνδεση σε βάση δεδομένων



Σχήμα 2.1: Δομή του directory

Μοντέλα

Στη παρούσα υλοποίηση, όπως και στις δύο προηγούμενες, δημιουργήθηκαν δύο μοντέλα για την αποθήκευση και αναπαράσταση των δεδομένων των κορυφών του γραφήματος. Στο *SingleTable* μοντέλο τα δεδομένα αποθηκεύονται σε έναν μόνο πίνακα, εν αντιθέσει με το *MultipleTable* μοντέλο όπου τα δεδομένα αποθηκεύονται σε πολλαπλούς πίνακες της CockroachDB. Στον φάκελο `models` βρίσκονται οι επιμέρους υλοποιήσεις αυτών των μοντέλων.

2.2.2 SingleTable Μοντέλο

Παρόλο που η αποθήκευση δεδομένων σε έναν και μόνο πίνακα έρχεται σε αντιπαράθεση με τον σχεδιασμό σε σχεσιακές βάσεις δεδομένων, το ST μοντέλο κατασκευάστηκε καταρχάς για την σύγκριση μεταξύ των δύο μοντέλων και κατά δεύτερον για την πληρότητα της εργασίας. Όπως φαίνεται και στον κώδικα 2.5, καλώντας την συνάρτηση `CreateStModel()`, δημιουργείται στην βάση δεδομένων ο πίνακας `dianode`, ο οποίος έχει την δομή που φαίνεται στο σχήμα 2.2.

dianode	
vid	STRING
vstart	STRING
vend	STRING
vlabel	STRING
attributes	JSONB
edge	JSONB

Σχήμα 2.2: Η δομή του SingleTable

Οι στήλες `vid`, `vstart`, `vend` και `vlabel`, αναφέρονται στο `id` της κορυφής του γραφήματος, την χρονική στιγμή που άρχισε να είναι ενεργή, την χρονική στιγμή που σταμάτησε να είναι ενεργή και τον χαρακτηρισμό της κορυφής αντίστοιχα. Στη στήλη `attributes` αποθηκεύονται επιπλέον χαρακτηριστικά της κορυφής σε μορφή `jsonb`, ενώ στη στήλη `edge` αποθηκεύονται σε `jsonb` τα δεδομένα σχετικά με την κατευθυνόμενη ακμή του γραφήματος.

Κάθε εγγραφή στον πίνακα αντιπροσωπεύει και μια ακμή μεταξύ δύο κορυφών του γραφήματος. Ο πίνακας σχεδιάστηκε με αυτό τον τρόπο, διότι στην τρέχουσα έκδοση της CockroachDB δεν ήταν εφικτή η αποθήκευση πολλαπλών `json(b)` αντικειμένων σε ένα `array/list`. Μια απλουστευμένη επιστροφή από ένα ερώτημα επί του πίνακα `dianode` είναι η εξής.

```
$ SELECT * FROM dianode;
 row   vid   vstart   vend    vlabel   attributes   edge
-----
 row 1 -> 1, 2010-12-21, 2011-03-20, Person, {name: Xaris}, {targetid: 2,
                                estart: 2010-12-21 , eend: 2011-03-20}
```

row	vid	vstart	vend	vlabel	attributes	edge
row 2 ->	1,	2010-12-21,	2011-03-20,	Person,	{name: Xaris},	{targetid: 3,
		estart: 2010-12-21	,	eend: 2011-01-03}		
row 3 ->	2,	2010-01-01,	2012-12-31,	Person,	{name: Alex},	{targetid: 3,
		estart: 2010-01-12	,	2011-04-14}		

2.2.3 MultipleTable Μοντέλο

Στο MT μοντέλο, τα δεδομένα κατανέμονται και αποθηκεύονται σε τρεις διαφορετικούς πίνακες. Αντίστοιχα με το προηγούμενο μοντέλο, καλώντας την συνάρτηση `CreateMtModel()`, δημιουργούνται στην βάση δεδομένων οι πίνακες `vertices`, `attributes` και `edges`. Στο σημείο αυτό πρέπει να αναφερθεί ότι ο πίνακας `edges` αφορά τις εξερχόμενες ακμές των κορυφών του γραφήματος, παρόλ' αυτά μπορεί να κατασκευαστεί πολύ εύκολα ένας ακόμη πίνακας που να αναπαριστά τις εισερχόμενες ακμές των κορυφών, με μια μόνο διαπέραση του πίνακα `edges`.

Επιπλέον σε σχέση με τις προηγούμενες υλοποιήσεις υπάρχει μια μικρή διαφοροποίηση στον τρόπο που αποθηκεύονται τα `attributes` κάθε κορυφής. Στη παρούσα υλοποίηση, κάθε `attribute` που αφορά μια κορυφή αποθηκεύεται σε έναν μόνο πίνακα, με κάθε σειρά να αναπαριστά ένα `attribute` μιας συγκεκριμένης κορυφής. Αντίθετα στις προηγούμενες υλοποιήσεις, κάθε ξεχωριστό `attribute` αποθηκεύονταν σε διαφορετικό πίνακα. Ο λόγος που έγινε η τροποποίηση αυτή έγκειται στην άρση της υποχρεωτικότητας κάθε `dataset` να ακολουθεί ένα συγκεκριμένο σχήμα.

Στο παρακάτω σχήμα φαίνεται η δομή του MultipleTable μοντέλου.

vertices	
vid	STRING
vstart	STRING
vend	STRING
vlabel	STRING

attributes	
aid	STRING
alabel	STRING
attribute	JSONB

edges	
label	STRING
sourceid	STRING
targetid	STRING
weight	STRING
estart	STRING
eend	STRING

Σχήμα 2.3: Η δομή του MultipleTable

Αντίστοιχα, στο επόμενο σχήμα φαίνονται απλουστευμένες επιστροφές από ερωτήματα επί των πινάκων αυτού του μοντέλου.

```
$ SELECT * FROM vertices;
  row   vid   vstart      vend      vlabel
-----
row 1 -> 1, 2010-12-21, 2011-03-20, Person}
row 2 -> 2, 2010-01-01, 2012-12-31, Person}
```

```
$ SELECT * FROM attributes;
  row   aid alabel      attribute
-----
row 1 -> 1, Person, {name: Xaris, start: 2010-12-21, end: 2099-01-01}
row 2 -> 1, Person, {color: blue, start: 2010-12-21, end: 2012-03-01}
row 3 -> 2, Person, {name: Alex, start: 2010-01-01, end:2099-01-01}
row 4 -> 2, Person, {color: yellow, start: 2010-01-01, end:2011-11-22}
```

```
$ SELECT * FROM edges;
  row   label   sourceid targetid weight      estart      eend
-----
row 1 -> Knows,      1,        2,        1,        2010-12-21, 2011-03-20}
row 2 -> Knows,      1,        3,        1,        2010-12-21, 2011-01-03}
row 3 -> Knows,      2,        3,        1,        2010-01-12, 2011-04-14}
```

Σχόλια επί των μοντέλων

Και στα δύο μοντέλα αποθηκεύονται ακριβώς τα ίδια δεδομένα, μόνο που στο ST τα δεδομένα είναι πιο πυκνά ως προς την πληροφορία που φέρουν ανά εγγραφή του πίνακα, ενώ στο MT είναι πιο αραιά. Επιπλέον, στο MT μοντέλο η διαχείριση των δεδομένων, λόγω της δομής τους, είναι πιο ανώδυνη σε σχέση με το έτερο μοντέλο. Τέλος, τα γράμματα v, a, e στην αρχή των εκάστοτε στηλών των πινάκων, σημαίνει vertex, attribute και edge αντίστοιχα. Επομένως το vid ή το estart σημαίνει vertex id και edge start.

Η σύγκριση των δύο αυτών μοντέλων θα γίνει σε επόμενο κεφάλαιο.

2.2.4 Ερωτήματα

Κατά την εκπόνηση της εργασίας αναπτύχθηκαν τρία ερωτήματα και στα δύο προαναφερθέντα μοντέλα, τα οποία υλοποιήθηκαν με αντίστοιχο τρόπο όπως και στις προηγούμενες εργασίες. Τα ερωτήματα αυτά είναι το *get alive vertices*, το *get one hop neighborhood* και το *get degree distribution*, δίνοντας έμφαση στο τελευταίο υλοποιώντας το με διαφορετικούς τρόπους, καθώς αυτό ήταν και το ερώτημα το οποίο απαιτούσε τον περισσότερο υπολογιστικό χρόνο. Τα ερωτήματα αυτά έχουν υλοποιηθεί για το κάθε μοντέλο στα αντίστοιχα αρχεία του φακέλου models και οι συναρτήσεις αυτών καλούνται από την main όπως φαίνεται στον κώδικα 2.6.

```

1 func main() {
2     var dbConn = db.GetConnection()
3     var dbConnST = db.GetConnection()
4
5     defer dbConn.Close(context.Background())
6     defer dbConnST.Close(context.Background())
7
8     mt := db.USEmt("sf3", dbConn)
9     st := db.USEst("st_sf10", dbConnST)
10
11     aliveV := st.GetAliveVertices("2010-01-01", "2011-08-10")
12     oneHop := st.GetOneHopNeighborhood("1", "2012-01-25")
13     degDistr := mt.GetDegreeDistribution("2010-01-01", "2012-08-13")
14 }

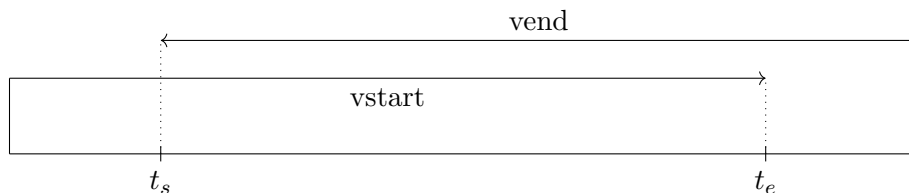
```

Κώδικας 2.6: Κλήση των ερωτημάτων

Για τα επόμενα παραδείγματα, θα χρησιμοποιηθεί το MultipleTable μοντέλο εφόσον αυτό έχει το μεγαλύτερο ενδιαφέρον, καθώς επίσης οι επιμέρους υλοποιήσεις του ST είναι τα ερωτήματα του MT προσαρμοσμένα στο ST μοντέλο.

Alive vertices

Στο ερώτημα αυτό επιστρέφονται οι κορυφές του γραφήματος οι οποίες ήταν ενεργές κάποια στιγμή μέσα σε ένα χρονικό διάστημα t_s, t_e . Στο σχήμα 2.4 φαίνονται τα χρονικά διαστήματα που εξετάζει το ερώτημα αυτό, όπου $vstart < vend$.



Σχήμα 2.4: Έγκυρα χρονικά διαστήματα

Για το MultipleTable μοντέλο, το ερώτημα βρίσκεται στον κώδικα 2.7. Τα δεδομένα που επιστρέφονται από το ερώτημα αποθηκεύονται σε ένα map, το οποίο ως κλειδί έχει την χρονιά που η εκάστοτε κορυφή ξεκίνησε να είναι ενεργή, και ως τιμή μία λίστα από τα ids των αντίστοιχων κορυφών.

```

1 SELECT
2     vid, EXTRACT(YEAR FROM DATE(vstart))
3 FROM vertices
4 WHERE
5     DATE(vstart) <= te AND DATE(vend) >= ts

```

Κώδικας 2.7: Ερώτημα get alive vertices

One hop neighborhood

Στο συγκεκριμένο ερώτημα, το οποίο φαίνεται στον κώδικα 2.8, επιστρέφονται τα ids όλων των γειτονικών κορυφών μιας συγκεκριμένης κορυφής μέχρις ενός χρονικού σημείου t_e . Τα δεδομένα που επιστρέφονται από τη ερώτηση αυτό, αποθηκεύονται σε μία λίστα που κρατά τα ids των επιστρεφόμενων κορυφών.

```

1 SELECT
2   targetid
3 FROM edges
4 WHERE
5   sourceid = vid AND DATE(estart) <= te

```

Κώδικας 2.8: Ερώτημα get one hop neighborhood

Degree distribution

Τέλος, υλοποιήθηκε το ερώτημα degree distribution, κατά το οποίο επιστρέφεται η κατανομή των βαθμών των κορυφών του γραφήματος μέσα σε ένα χρονικό διάστημα t_s, t_e . Το ερώτημα αυτό υλοποιήθηκε με δύο διαφορετικά SQL ερωτήματα, εκ των οποίων το πρώτο ακολουθεί τον τρόπο των προηγούμενων υλοποιήσεων και το δεύτερο είναι μια εκ νέου υλοποίηση. Επιπλέον το δεύτερο ερώτημα αναπτύχθηκε με τρεις διαφορετικούς τρόπους και θα περιγραφούν στη συνέχεια με τη σειρά που αναπτύχθηκαν.

Τα δύο διαφορετικά ερωτήματα φαίνονται στους κώδικες 2.9 και 2.10 με τις αντίστοιχες επιστροφές τους από την βάση δεδομένων.

```

1 SELECT
2   sourceid, estart, eend
3 FROM edges
4 WHERE
5   DATE(eend) >= ts AND DATE(estart) <= te

```

Κώδικας 2.9: Παλιό ερώτημα get degree distribution

row	sourceid	estart	eend

row 1 ->	1,	2010-12-21,	2011-03-20
row 2 ->	1,	2010-12-21,	2011-04-21
row 3 ->	1,	2010-12-24,	2012-12-31
row 4 ->	1,	2010-02-04,	2010-12-25
row 5 ->	2,	2010-02-04,	2012-12-31

```

1 SELECT
2   sourceid,
3   COUNT(targetid),
4   EXTRACT(YEAR FROM DATE(estart))::int AS start,
5   least(EXTRACT(YEAR FROM DATE(eend))::int, EXTRACT(YEAR FROM DATE(te))::
6         int)::int AS end
7 FROM edges
8 WHERE DATE(eend) >= ts AND DATE(estart) <= te
9 GROUP BY
10  sourceid,
11  EXTRACT(YEAR FROM DATE(estart)),
12  EXTRACT(YEAR FROM DATE(eend))
ORDER BY sourceid ASC

```

Κώδικας 2.10: Καινούριο ερώτημα get degree distribution

row	sourceid	count	start	end
row 1 ->	1,	2,	2010,	2011
row 2 ->	1,	1,	2010,	2012
row 3 ->	1,	1,	2010,	2010
row 4 ->	2,	1,	2010,	2012

Όπως φαίνεται στα παραπάνω παραδείγματα, τα οποία εφαρμόζονται στην ίδια βάση δεδομένων, τα ερωτήματα επιστρέφουν διαφορετικά αποτελέσματα, με το πρώτο να είναι ουσιαστικά ένα αφιltrάριστο full table scan (στο χρονικό πεδίο ενδιαφέροντος $[t_s, t_e]$), ενώ το δεύτερο ερώτημα μέσω του COUNT() και GROUP BY να συμπύκνωση τα δεδομένα. Όπως φαίνεται και παραπάνω στο ερώτημα 2.10, και πιο συγκεκριμένα στις σειρές 1 έως 3, επιστρέφεται για κάθε κορυφή ο βαθμός που έχει (*count*) σε ένα σύνολο επικαλυπτόμενων χρονικών περιόδων (*start* και *end*). Με το τρόπο αυτό, σύμφωνα με τα datasets που δοκιμάστηκαν, η βάση δεδομένων επιστρέφει $\sim 12 - 25$ φορές λιγότερες σειρές, όπως φαίνεται και στον πίνακα 2.1, με κάθε σειρά να φέρει περισσότερη πληροφορία.

Επομένως, οι πράξεις που απαιτούνται για την επεξεργασία των δεδομένων που επιστρέφει η βάση δεδομένων είναι σημαντικά λιγότερες, όμως στην πραγματικότητα τις πράξεις αυτές τις εκτελεί η βάση δεδομένων (εν αντιθέσει με τον υπολογιστή που «ρωτά» τη βάση δεδομένων), το κόστος των οποίων θα δούμε στη συνέχεια.

Μέγεθος dataset	Ερώτημα 2.9	Ερώτημα 2.10
small	668.431	53.216
medium	2.304.951	155.090
large	10.499.491	424.893

Πίνακας 2.1: Αριθμός σειρών που επιστρέφονται από τα ερωτήματα

Ας ονομάσουμε αλγόριθμος₁, αλγόριθμος₂, αλγόριθμος₃ και αλγόριθμος₄ τους τέσσερις αλγόριθμους που υλοποιήθηκαν για την επεξεργασία των δεδομένων που επιστρέφονται από τη βάση δεδομένων. Ο αλγόριθμος₁ είναι αυτούσιος ο αλγόριθμος που αναπτύχθηκε στην προηγούμενη εργασία που υλοποιήθηκε με την MongoDB και είναι ο μόνος που χρησιμοποιεί το ερώτημα 2.9. Αντίστοιχα, ο αλγόριθμος₂ είναι παραπλήσιος αλγόριθμος, αλλά χρησιμοποιεί το ερώτημα 2.10. Τέλος ο αλγόριθμος₃ είναι μια βελτιστοποιημένη εκδοχή του προηγούμενου, ενώ ο αλγόριθμος₄ είναι ίδιος με τον αλγόριθμο₃, μόνο που χρησιμοποιεί τις τεχνικές που μελετήσαμε στην υποενότητα 2.1.1, δηλαδή την ταυτόχρονη επεξεργασία (concurrency). Παρακάτω παρατίθενται οι ψευδοκώδικες για τους αλγορίθμους.

Algorithm 1 GetDegreeDistributionFetchAllVertices ερώτημα 2.9

```

1: while rows.hasNext() do ▷ full table scan
2:   id, degree, start, end ← row.id, row.degree, row.start, row.end
3:   for year from start to end do
4:     vertexDistr[id][year]++ ▷ main difference with Algorithm 2
5:   end for
6: end while

7: for each v in vertexDistr do
8:   for year, degree in v do
9:     degreeDistribution[year][degree]++
10:  end for
11: end for

```

Algorithm 2 GetDegreeDistribution ερώτημα 2.10

```

1: while rows.hasNext() do ▷ reduced rows
2:   id, degree, start, end ← row.id, row.degree, row.start, row.end
3:   for year from start to end do
4:     vertexDistr[id][year] += degree
5:   end for
6: end while

7: for each v in vertexDistr do
8:   for year, degree in v do
9:     degreeDistribution[year][degree]++
10:  end for
11: end for

```

Algorithm 3 GetDegreeDistributionOptimized ερώτημα 2.10

```

1: prewid ← NULL
2: while rows.hasNext() do ▷ reduced rows and memory
3:   newid, degree, start, end ← row.id, row.degree, row.start, row.end
4:   if newid ≠ prewid && prewid ≠ NULL then
5:     for k, v in vertexDistr do
6:       degreeDistr[k][v] ++
7:     end for
8:     vertexDistr ← empty map[int]int
9:   end if

10:  for year from start to end do
11:    vertexDistr[year] += degree
12:  end for

13:  prewid ← newid
14: end while

```

Βασικές διαφορές μεταξύ αλγορίθμων

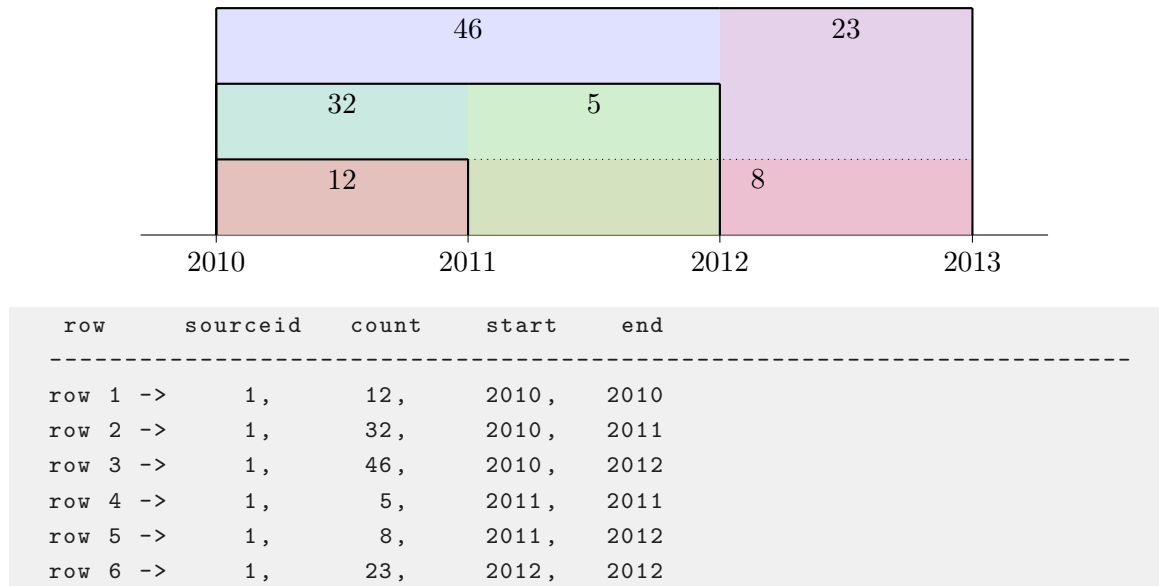
Ο αλγόριθμος₁ και ο αλγόριθμος₂ αρχικά αποθηκεύουν όλα τα δεδομένα που επιστρέφει η βάση δεδομένων στο map vertexDistr, το οποίο συσχετίζει τους βαθμούς **όλων** των κορυφών του γραφήματος ανά χρονιά, και έπειτα διαπερνάται αυτό το map ώστε να υπολογιστεί η κατανομή. Ο αλγόριθμος₃, αντιθέτως, αποθηκεύει στο map vertexDistr τους βαθμούς ανά χρονιά της **τρέχουσας** κορυφής και υπολογίζει την κατανομή.

Επομένως η κατανομή υπολογίζεται σταδιακά, με τρόπο τέτοιο ώστε να μην αποθηκεύεται πληροφορία η οποία πλέον δεν μας αφορά, και έτσι κατ' αυτό το τρόπο κατά το run-time επιτυγχάνουμε αρκετά λιγότερη χρησιμοποίηση της μνήμης, όπου σε πραγματικά μεγάλο όγκο δεδομένων εξοικονομούμε σημαντικά πόρους και βελτιώνουμε την απόδοση του συστήματος χωρίς περιττή επιβάρυνση.

Επιπλέον, πρέπει να σημειωθεί πως ο αλγόριθμος₃ **δεν** επιτυγχάνει μικρότερη χρονική πολυπλοκότητα, παρόλ' αυτά μέσω αυτού του αλγορίθμου μειώνεται σημαντικά το N , πράγμα το οποίο κάνει την εκτέλεσή του να είναι ταχύτερη.

Εμβάθυνση στα ερωτήματα

Για την καλύτερη κατανόηση του τρόπου που επιστρέφουν τα ερωτήματα 2.9 και 2.10 τα δεδομένα, παρατίθεται στο σχήμα 2.5 η επιστροφή του ερωτήματος 2.10 για μια συγκεκριμένη κορυφή του γραφήματος.



Σχήμα 2.5: Παράδειγμα επιστροφής ερωτήματος 2.10

Όπως έχουμε αναφέρει προηγουμένως, το ερώτημα αυτό συμπτύσσει σημαντικά τα δεδομένα που βρίσκονται στη βάση δεδομένων σε σχέση με το ερώτημα 2.9. Πιο συγκεκριμένα, για τα ίδια δεδομένα το ερώτημα 2.9 επιστρέφει 126 σειρές ($12 + 32 + 46 + 5 + 8 + 23 = 126$), και αυτό συμβαίνει γιατί κάθε σειρά που επιστρέφεται, αναπαριστά **ακριβώς μια** ακμή του γραφήματος, εν αντιθέσει με το ερώτημα 2.10 που αναπαριστά περισσότερες.

Έστω $e = \# \text{ακμών}$, $v = \# \text{κορυφών}$ και $y = \# \text{ετών}$

Επομένως εάν γενικεύσουμε τα παραπάνω, ο αριθμός των σειρών που επιστρέφει το ερώτημα 2.9 είναι αναλογικός του e του γραφήματος, ενώ στο ερώτημα 2.10 εξαρτάται αποκλειστικά από το v του γραφήματος και το y που εφαρμόζεται στο ερώτημα, αφού για κάθε κορυφή επιστρέφονται **το πολύ** $\binom{y}{2} + y$ σειρές. Ο διωνυμικός συντελεστής χρησιμοποιείται για τον υπολογισμό όλων των δυνατών συνδυασμών των ετών ανά δύο (δηλαδή 2010-2011, 2010-2012, 2011-2012...) και η πρόσθεση του y για τον υπολογισμό των υπόλοιπων συνδυασμών των ετών με τον εαυτό τους (δηλαδή 2010-2010, 2011-2011...).

$$q_1 = e \quad (1)$$

$$q_2 = v \times \binom{y}{2} + y = v \times \frac{y!}{2!(y-2)!} + y = \frac{v}{2}y(y+1) \quad (2)$$

άρα, όσο $e > \frac{v}{2}y(y+1)$, το q_2 είναι πιο οικονομικό

Σε μεγάλα γραφήματα όμως, που εξομοιώνουν πραγματικά δίκτυα, για ένα ερώτημα degree distribution ισχύει $y \ll v < e$. Παρακάτω παρατίθεται, ένας πίνακας που υπολογίζει τα q_1 και q_2 για διάφορες πειραματικές τιμές y, v, e . Στο σημείο αυτό πρέπει να αναφερθεί πως για τα παρακάτω αποτελέσματα έχουν γίνει οι εξής παραδοχές:

1. Όλες οι κορυφές στο q_1 , έχουν τον ίδιο βαθμό.
2. Όλες οι κορυφές στο q_2 , υπάρχουν καθ' όλη τη διάρκεια χρονική διάρκεια που ορίζεται από το ερώτημα.

Επομένως, και για τα δύο ερωτήματα υπολογίζουμε τις χειρότερες περιπτώσεις.

	v	y	e per vertex	rows
$q_1 \times v$	1.000	-	500	5×10^5
q_2	1.000	3	-	6×10^3
$q_1 \times v$	100.000	-	300	3×10^7
q_2	100.000	10	-	$5,5 \times 10^6$
$q_1 \times v$	1.000.000	-	500	5×10^8
q_2	1.000.000	5	-	$1,5 \times 10^7$
$q_1 \times v$	10.000.000	-	5.000	5×10^{10}
q_2	10.000.000	5	-	$1,5 \times 10^8$
$q_1 \times v$	10.000.000	-	10.000	1×10^{11}
q_2	10.000.000	20	-	$2,1 \times 10^9$
$q_1 \times v$	10.000.000	-	100	1×10^9
q_2	10.000.000	25	-	$3,25 \times 10^9$

Πίνακας 2.2: Αποτελέσματα των q_1 και q_2

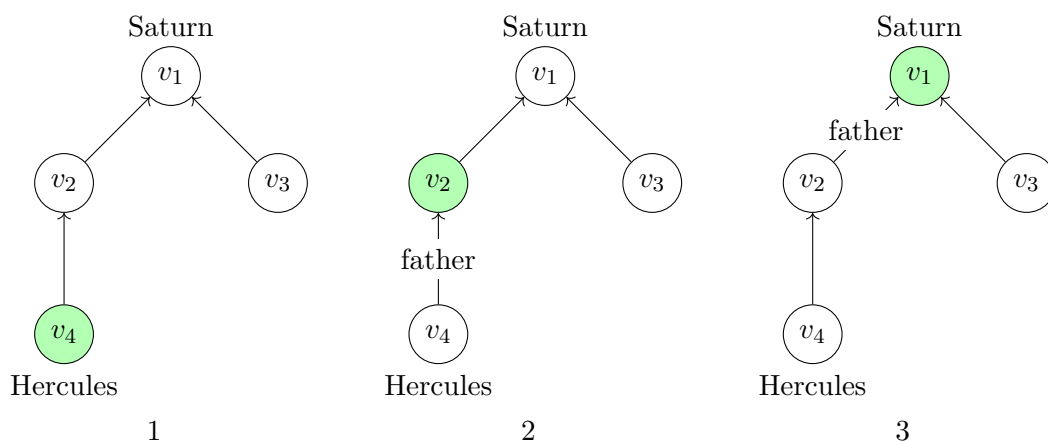
Κεφάλαιο 3

Υλοποίηση Μέρος Β

Στο κεφάλαιο αυτό εξετάζεται και αναλύεται η υλοποίηση του δεύτερου μέρους της παρούσας διπλωματικής εργασίας. Πιο συγκεκριμένα η υλοποίηση αυτή αφορά καταρχάς την τροποποίηση της γλώσσας ερωτημάτων gremlin, και κατά δεύτερον τη διασύνδεση αυτής με την CockroachDB με σκοπό την αποθήκευση των δεδομένων σύμφωνα με τα μοντέλα που έχουν αναπτυχθεί.

3.1 Gremlin

Η gremlin είναι μια γλώσσα ερωτημάτων (query language), η οποία χρησιμοποιείται για την ανάκτηση και τροποποίηση δεδομένων σε γραφήματα και γενικότερα σε δομές γραφημάτων. Η γλώσσα αυτή είναι μια functional γλώσσα, στην οποία οι τελεστές διάσχισης/διαπέρασης του γραφήματος συνδέονται αλυσιδωτά ώστε να σχηματίσουν εκφράσεις που προσομοιάζουν μονοπάτια του γραφήματος. Για παράδειγμα ένα ερώτημα πάνω στο γράφημα του σχήματος 3.1 είναι, «από τον Ηρακλή (1), διάσχισε τον πατέρα του (2), και μετά τον πατέρα του πατέρα του (3) και επέστρεψε το όνομα του παππού».



Σχήμα 3.1: Διαπέραση γραφήματος στην gremlin

Το παραπάνω ερώτημα αποτυπώνεται στην gremlin με τον εξής τρόπο¹.

```
gremlin> g.V().has('name', 'hercules').out('father')
          .out('father').values('name')
==>Saturn
```

Τα gremlin steps, είναι οι θεμελιώδεις πράξεις που χρησιμοποιούνται για την διάσχιση και την διαχείριση των δεδομένων στα γραφήματα και κατηγοριοποιούνται ανάλογα με τη λειτουργικότητά τους. Τα κυριότερα από αυτά βρίσκονται και επεξηγούνται στον πίνακα 3.2.

Transform Steps

V()	Επιστρέφει όλες τις κορυφές του γραφήματος
E()	Επιστρέφει όλες τις ακμές του γραφήματος
out(label)	Επιστρέφει όλες τις κορυφές που συνδέονται με εξερχόμενη ακμή (με το καθορισμένο label) με την τρέχουσα κορυφή
in(label)	Επιστρέφει όλες τις κορυφές που συνδέονται με εισερχόμενη ακμή (με το καθορισμένο label) με την τρέχουσα κορυφή
outE(label)	Επιστρέφει όλες τις εξερχόμενες ακμές (με το καθορισμένο label) της τρέχουσας κορυφής
inE(label)	Επιστρέφει όλες τις εισερχόμενες ακμές (με το καθορισμένο label) της τρέχουσας κορυφής
both(label)	Επιστρέφει όλες τις κορυφές που συνδέονται με ακμή (με το καθορισμένο label) με την τρέχουσα κορυφή
bothE(label)	Επιστρέφει όλες τις ακμές (με το καθορισμένο label) της τρέχουσας κορυφής

map/sideEffect Steps

addV(label)	Προσθέτει μια κορυφή (με το καθορισμένο label) στο γράφημα
addE(label)	Προσθέτει μια ακμή (με το καθορισμένο label) στο γράφημα
property(key, value)	Προσθέτει το property key με τιμή value στο τρέχον στοιχείο

Filter Steps

has(key, value)	Επιστρέφει όλα τα στοιχεία, τα οποία έχουν το καθορισμένο key/value property
-----------------	--

Πίνακας 3.1: Gremlin steps

3.2 Apache Groovy

Η Groovy είναι μια αντικειμενοστραφής γλώσσα προγραμματισμού, η οποία αναπτύχθηκε ώστε να τρέχει στο JVM και ως εκ τούτου είναι πλήρως συμβατή με τη Java καθώς επίσης μπορεί να αξιοποιήσει και τις βιβλιοθήκες της.

¹Παράδειγμα από: <https://docs.janusgraph.org/getting-started/gremlin/>

Επιπλέον, ένα από τα χαρακτηριστικά της Groovy είναι ότι διαθέτει εκτεταμένη υποστήριξη για μεταπρογραμματισμό². Εκμεταλλευόμενη αυτό το χαρακτηριστικό, η Groovy μπορεί να μεταγλωττίσει Gremlin κώδικα, σε χαμηλού επιπέδου Java Pipes. Ο κώδικας 3.1, παρουσιάζει ένα τυπικό σχεδιαστικό μοτίβο κατά το οποίο συνυπάρχουν παράλληλα η Groovy, η Java και η Gremlin. Στο παρακάτω παράδειγμα μια κλάση της Java καλεί τη μέθοδο μιας κλάσης της Groovy, η οποία χρησιμοποιεί την Gremlin για την διάσχιση μιας graph βάσης δεδομένων.

```

1 // a Groovy class
2 class GraphAlgorithms {
3     static {
4         Gremlin.load()
5     }
6
7     public static Map<Vertex, Integer> eigenvectorRank(Graph g) {
8         Map<Vertex,Integer> m = [:]; int c = 0
9         g.V.as('x').out.groupCount(m).loop('x') {c++ < 1000}.iterate()
10        return m
11    }
12 }
13
14 // a Java class
15 public class GraphFramework {
16     public static void main(String[] args) {
17         System.out.println(GraphAlgorithms.eigenvectorRank(new Neo4jGraph("/tmp
18         /graphdata")));
19     }
20 }

```

Κώδικας 3.1: Χρήση της Gremlin μέσω της Groovy

Στην παρούσα διπλωματική εργασία η Groovy χρησιμοποιήθηκε για να τροποποιήσουμε την Gremlin, ώστε να είναι πιο εύχρηστη στην προσομοίωση των μοντέλων που αναλύθηκαν στην ενότητα 2.2 . Μέσω του μεταπρογραμματισμού που μας προσφέρει η Groovy, μπορούμε κατά το run-time του προγράμματος να τροποποιήσουμε ή να προσθέσουμε μεθόδους σε μια κλάση ή σε ένα interface, επεκτείνοντας έτσι την λειτουργικότητά τους χωρίς να τροποποιήσουμε τον πηγαίο κώδικα. Ένα παράδειγμα το οποίο καταδεικνύει αυτή την υλοποίηση βρίσκεται στον κώδικα 3.2, στον οποίο εισάγουμε στην Gremlin ένα καινούριο keyword, το lifetime, το οποίο αποθηκεύεται σαν property της εκάστοτε κορυφής/ακμής. Τα αρχεία της υλοποίησης βρίσκονται στον φάκελο gremlin όπως φαίνεται και στο σχήμα 2.1.

```

1 GraphTraversal.metaClass.lifetime = { String startValue, String endValue ->
2     delegate.property('start', startValue)
3     delegate.property('end', endValue)
4 }

```

Κώδικας 3.2: Μεταπρογραμματισμός με τη Groovy

²Μεταπρογραμματισμός με Groovy: <https://groovy-lang.org/metaprogramming.html>

Παρακάτω παρατίθεται ο πίνακας των custom steps που υλοποιήθηκαν και η επεξήγησή τους.

Custom Steps	
lifetime(start, end)	Αποθηκεύει σαν property το χρονικό διάστημα που ήταν ενεργή η κορυφή/ακμή
vid(value)	Αποθηκεύει σαν property το vid της κορυφής
insertE(label, sourceid, targetid)	Εισάγει μια κατευθυνόμενη ακμή (με το καθορισμένο label) στο γράφημα από την κορυφή με vid = sourceid στην κορυφή με vid = targetid
weight(value)	Αποθηκεύει σαν property το καθορισμένο βάρος της ακμής
addA(name, value, start, end)	Εισάγει properties στην τρέχουσα κορυφή για το καθορισμένο χρονικό διάστημα
deleteV(vid, end)	Θέτει τον τερματισμό της κορυφής με το καθορισμένο vid την καθορισμένη χρονική στιγμή
deleteE(sourceid, targetid, end)	Θέτει τον τερματισμό της καθορισμένης ακμής την καθορισμένη χρονική στιγμή

Πίνακας 3.2: Custom gremlin steps

Στον κώδικα 3.3 φαίνονται στη πράξη τα custom steps που υλοποιήθηκαν κατά την εκπόνηση της εργασίας. Αρχικά δημιουργούνται δύο κορυφές και προστίθενται σε αυτές ένα attribute. Σε περίπτωση που οριστεί το χρονικό διάστημα για το οποίο θα ισχύσει το attribute, το διάστημα αυτό αποθηκεύεται ως meta-property του attribute. Στην αντίθετη περίπτωση, αποθηκεύεται εσωτερικά ως meta-property στο attribute, το χρονικό διάστημα στο οποίο είναι ενεργός ο κόμβος. Στη συνέχεια, προστίθεται μια ακμή μεταξύ των δύο αυτών κορυφών και τέλος, ενημερώνεται η τιμή του end της κορυφής και της ακμής αντίστοιχα.

```
gremlin> g.addV('Person').vid('1').lifetime('2010-01-01', '2099-01-01')
      .addA('color', 'blue', '2010-01-01', '2011-01-01')
==>v[0]
gremlin> g.addV('Person').vid('2').lifetime('2011-01-01', '2013-12-31')
      .addA('color', 'blue').addA('name', 'Xaris')
==>v[2]
gremlin> g.insertE('PersonKnowsPerson', '1', '2')
      .lifetime('2011-01-01', '2012-12-31').weight('1')
==>e[10][0-PersonKnowsPerson->5]
gremlin> g.deleteV('1', '2012-12-31')
==>v[0]
gremlin> g.deleteE('1', '2', '2011-12-31')
==>e[10][0-PersonKnowsPerson->5]
```

Κώδικας 3.3: Εφαρμογή των custom gremlin steps

3.3 Gremlin Parser

Στο πλαίσιο της υλοποίησης της εργασίας, αναπτύχθηκε επίσης ένας parser στην Go, με σκοπό την αποθήκευση δεδομένων από κώδικα της gremlin στην CockroachDB. Ο parser βασίζεται στις κανονικές εκφράσεις, όπως φαίνεται στο παράδειγμα 3.4 παρακάτω. Στην υλοποίηση αυτή δημιουργήθηκαν δύο structs (Vertex και Edge) στα οποία αποθηκεύονται τα δεδομένα που επιστρέφονται από τις κανονικές εκφράσεις και έπειτα επιστρέφονται στην μέθοδο του εκάστοτε μοντέλου, ώστε να αποθηκευτούν στη βάση δεδομένων.

```

1 type Edge struct {
2     Label      string
3     SourceID   string
4     TargetID   string
5     Start      string
6     End        string
7     Weight     string
8 }
9
10 func parseInsertE(input string) Edge {
11     e := Edge{}
12
13     //parsing label, sourceid, targetid
14     labelRegex := regexp.MustCompile(`g\.insertE\('([^\']+)`,
15     `'([^\']+)`, `'([^\']+)'\)`
16     labelMatch := labelRegex.FindStringSubmatch(input)
17     if len(labelMatch) > 0 {
18         e.Label = labelMatch[1]
19         e.SourceID = labelMatch[2]
20         e.TargetID = labelMatch[3]
21     }
22
23     //parsing lifetime
24     lifetimeRegex := regexp.MustCompile(`\.lifetime\('([^\']+)`,
25     `'([^\']+)'\)`
26     lifetimeMatch := lifetimeRegex.FindStringSubmatch(input)
27     if len(lifetimeMatch) > 0 {
28         e.Start = lifetimeMatch[1]
29         e.End = lifetimeMatch[2]
30     }
31
32     //parsing weight
33     weightRegex := regexp.MustCompile(`\.weight\('([^\']+)'\)`
34     weightMatch := weightRegex.FindStringSubmatch(input)
35     if len(weightMatch) > 0 {
36         e.Weight = weightMatch[1]
37     }
38     return e
39 }

```

Κώδικας 3.4: Αποθήκευση δεδομένων από την gremlin στην CockroachDB

Κεφάλαιο 4

Συγκρίσεις και Αποτελέσματα

Στο τελευταίο κεφάλαιο της διπλωματικής εργασίας θα παρουσιαστούν καταρχάς τα αποτελέσματα των συγκρίσεων των επιμέρους υλοποιήσεων μεταξύ των SingleTable και MultipleTable μοντέλων, καθώς επίσης θα εξεταστούν και οι επιδόσεις της βάσης δεδομένων που χρησιμοποιήθηκε στην παρούσα εργασία, δηλαδή την CockroachDB, εν συγκρίσει με τη βάση δεδομένων που χρησιμοποιήθηκε σε προηγούμενη εργασία, την MongoDB, όσον αφορά τα ερωτήματα που αναπτύχθηκαν. Τέλος θα γίνει μια σύντομη αναφορά σε πιθανές μελλοντικές εργασίες, που αποσκοπούν στην ανάπτυξη και την πληρότητα της έρευνας.

4.1 CockroachDB και MongoDB

4.1.1 Εισαγωγή Δεδομένων

Για τη διεξαγωγή των συγκρίσεων χρησιμοποιήθηκαν τρία datasets, όπως ήδη έχουμε αναφέρει (μικρού, μεσαίου και μεγάλου μεγέθους) και εφαρμόστηκαν και στις δύο βάσεις δεδομένων. Στον πίνακα 4.1 φαίνονται οι χρόνοι εισαγωγής των αρχείων στις βάσεις δεδομένων.

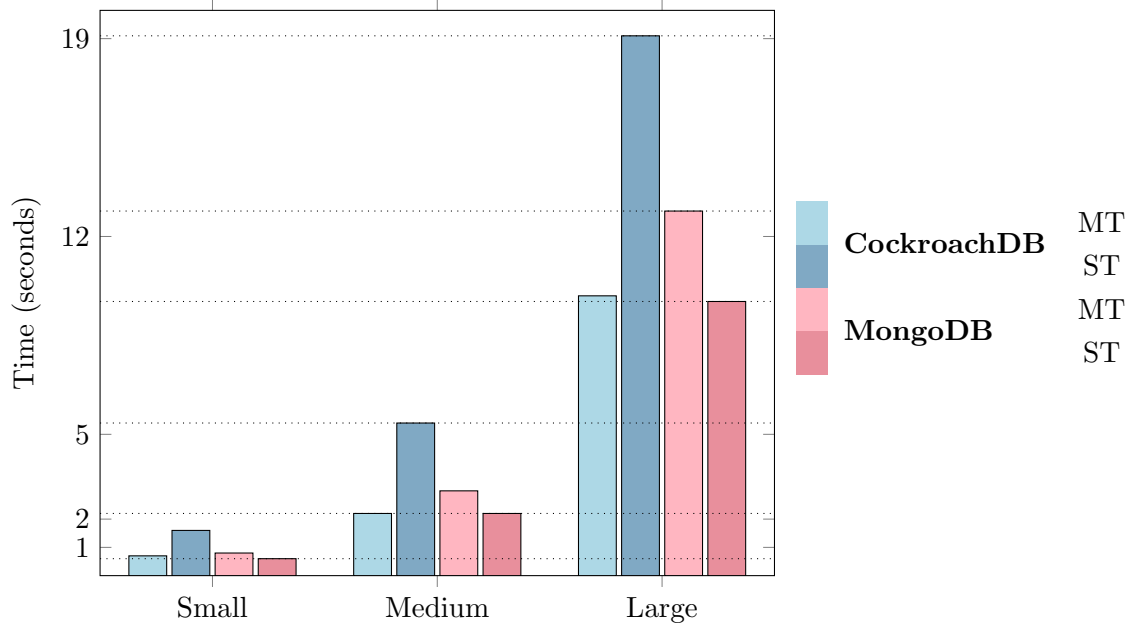
Database	Model	Small	Medium	Large
CockroachDB	SingleTable	107 min	310 min	1773 min
	MultipleTable	23 min	77 min	258 min
MongoDB	SingleTable	9 min	31 min	311 min
	MultipleTable	2 min	7 min	23 min

Πίνακας 4.1: Χρόνοι εισαγωγής δεδομένων

Όπως φαίνεται, η MongoDB εισάγει τα δεδομένα αρκετά ταχύτερα σε σχέση με την CockroachDB. Επίσης και στις δύο βάσεις δεδομένων, το MultipleTable μοντέλο έχει καλύτερες επιδόσεις συγκριτικά με το SingleTable, όσον αφορά τη συγκεκριμένη μέτρηση.

4.1.2 Degree Distribution

Η σύγκριση αυτού του ερωτήματος χρησιμοποιεί τον αλγόριθμο₁, δηλαδή οι επιμέρους υλοποιήσεις των παρακάτω ερωτημάτων είναι (όσο επιτρέπεται) αντίστοιχες. Επιπλέον, οι τιμές αυτές αναφέρονται σε ερωτήματα όπου το ενδιαφερόμενο χρονικό διάστημα εφαρμογής $[t_s, t_e]$ είναι το μέγιστο. Συγκρίσεις για μικρότερα χρονικά διαστήματα βρίσκονται στο παράρτημα Α'.

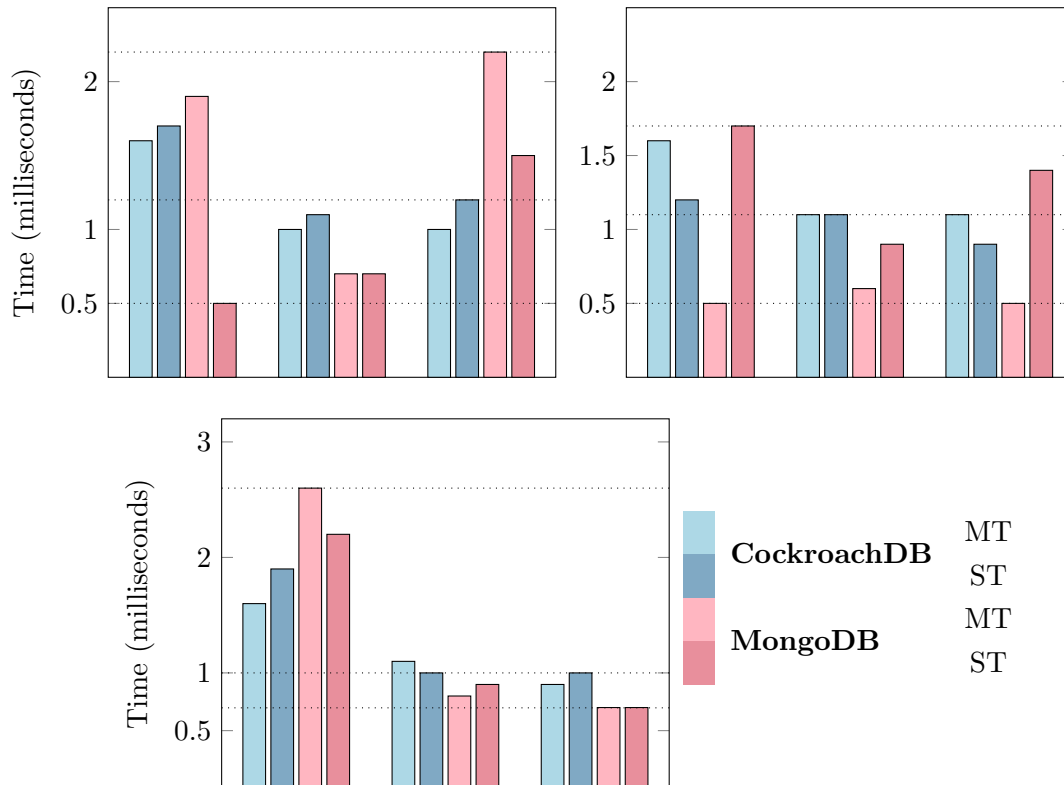


Σχήμα 4.1: Degree Distribution στις βάσεις δεδομένων (3/3 του χρόνου)

Όπως φαίνεται και στο σχήμα 4.1, στον χρόνο εκτέλεσης του ερωτήματος, για τα συγκεκριμένα datasets, στο Multiple Table μοντέλο της CockroachDB, σε σχέση με το αντίστοιχο της MongoDB βελτιώνεται κατά $\sim 12 - 23\%$. Αντίθετα, το Single Table μοντέλο της CockroachDB εκτελείται $\sim 50 - 60\%$ πιο αργά σε σχέση με το αντίστοιχο της MongoDB. Κάτι που αξίζει να σημειωθεί είναι πως το μοντέλο που εκτελεί γρηγορότερα το συγκεκριμένο ερώτημα στις δύο βάσεις δεδομένων διαφέρει. Πιο συγκεκριμένα, στην CockroachDB στο MT μοντέλο το ερώτημα εκτελείται ταχύτερα ενώ στην MongoDB στο ST, και αν συγκρίνουμε τα δύο αυτά μοντέλα, έχουν παρόμοιες επιδόσεις επί αυτού του ερωτήματος.

4.1.3 One Hop Neighborhood

Το συγκεκριμένο ερώτημα εκτελέστηκε για τυχαίες κορυφές του γραφήματος των τριών dataset. Όπως φαίνεται παρακάτω, ως επί το πλείστον, η MongoDB έχει καλύτερες επιδόσεις σε αυτό το ερώτημα και πιο συγκεκριμένα εκτελείται $\sim 10-70\%$ γρηγορότερα. Όσον αφορά τα δύο μοντέλα, Multiple Table και Single Table στην CockroachDB, παρουσίασαν παρόμοιους χρόνους εκτέλεσης σε αυτό το ερώτημα. Όπως και στη προηγούμενη σύγκριση, αυτές οι τιμές αναφέρονται στο μέγιστο χρονικό διάστημα εκτέλεσης του ερωτήματος. Περαιτέρω συγκρίσεις βρίσκονται στο στο παράρτημα Α'.

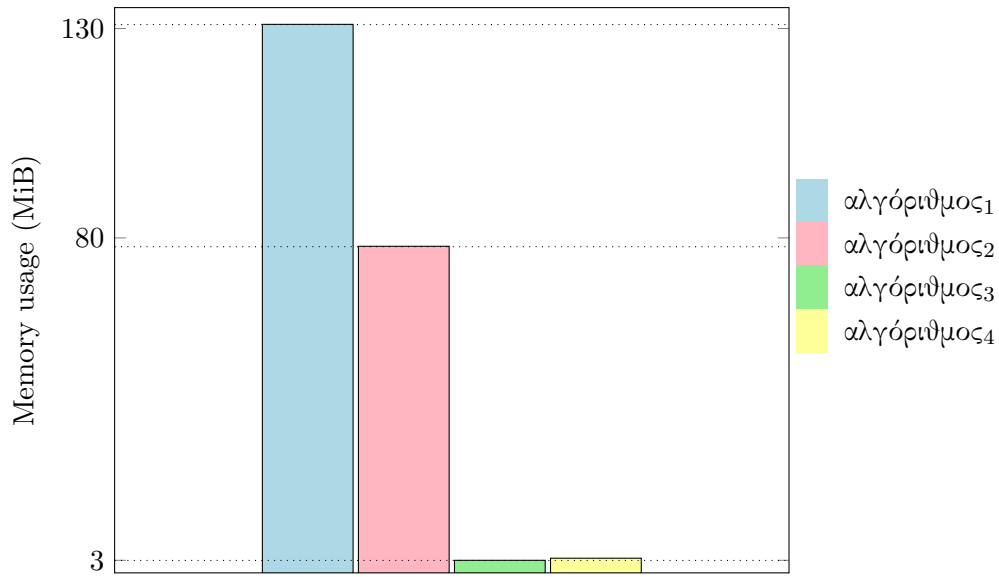


Σχήμα 4.2: One Hop Neighborhood στις βάσεις δεδομένων (3/3 του χρόνου)

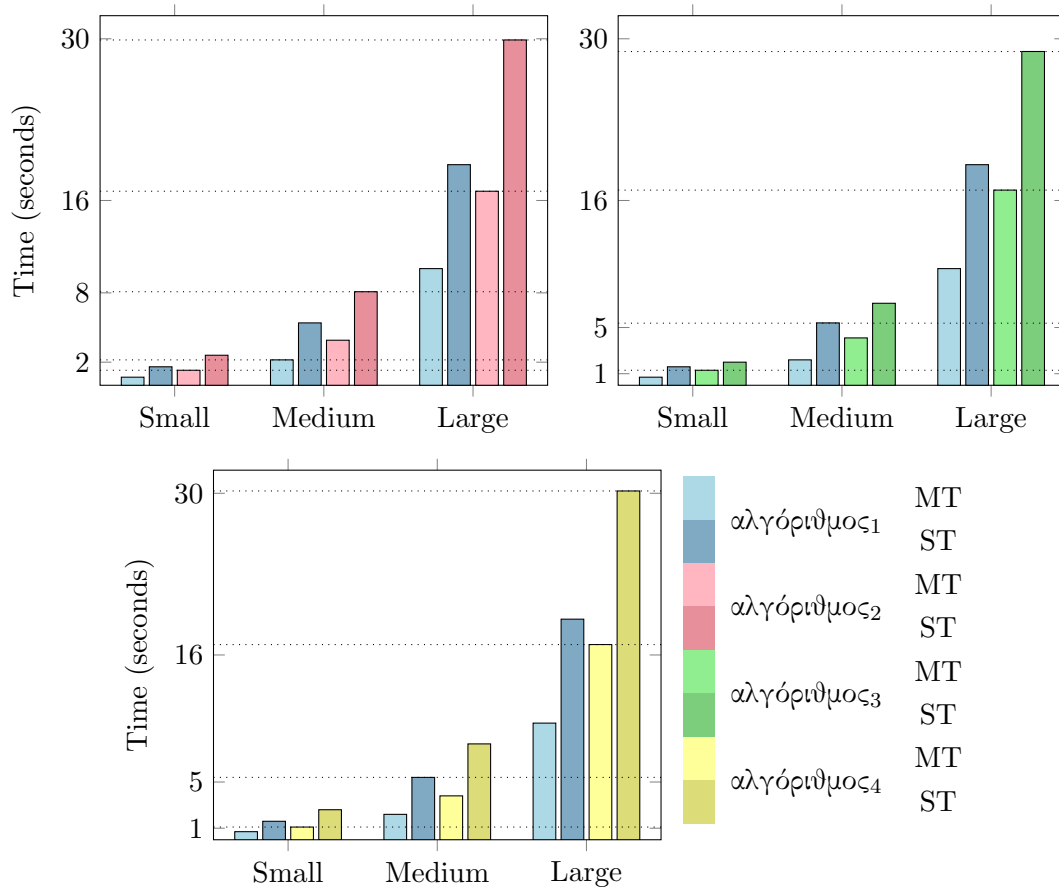
4.2 Διαφορετικές υλοποιήσεις

Στην ενότητα αυτή θα εξεταστούν οι διαφορετικές υλοποιήσεις του ερωτήματος degree distribution που αναπτύχθηκαν για την εργασία αυτή, οι οποίες παρουσιάστηκαν στην υποενότητα 2.2.4.

Στο σχήμα 4.3 παρουσιάζεται η χρησιμοποίηση της μνήμης των διαφορετικών αλγορίθμων κατά το run time του προγράμματος, πράγμα το οποίο ήταν και το κύριο έναυσμα για την δημιουργία τους. Οι συγκεκριμένες μετρήσεις έγιναν στο MT μοντέλο και για το μεγαλύτερο dataset, παρόλ' αυτά στο ST οι τιμές ήταν παρόμοιες.



Σχήμα 4.3: Χρησιμοποίηση μνήμης στις διαφορετικές υλοποιήσεις



Σχήμα 4.4: Διαφορετικές υλοποιήσεις του degree distribution (3/3 του χρόνου)

Όσον αφορά τους χρόνους εκτέλεσης των υλοποιήσεων αυτών, στο σχήμα 4.4 φαίνονται οι συγκρίσεις των αλγορίθμων με τον αλγόριθμο₁, και άρα η έμμεση σύγκριση αυτών με την υλοποίηση που έγινε στην MongoDB. Ο αλγόριθμος₁ στη συγκεκριμένη υλοποίηση είναι ~ 35 – 45% ταχύτερος σε σχέση με τους υπόλοιπους, ωστόσο υπάρχουν κάποια ζητήματα τα οποία πρέπει να εξεταστούν. Σε αυτό το σημείο να υπενθυμίσω πως ο αλγόριθμος₁ χρησιμοποιεί το ερώτημα 2.9 ενώ οι υπόλοιποι το ερώτημα 2.10.

Όπως έχουμε αναφέρει, σε όλες τις δοκιμές και τις μετρήσεις που πραγματοποιήθηκαν, το cluster της βάσης δεδομένων είχε μόνο έναν κόμβο. Ως εκ τούτου, σε ένα κατανεμημένο σύστημα θεωρώ πως οι αλγόριθμοι που χρησιμοποιούν το ερώτημα 2.10 θα εκτελεστούν ταχύτερα από τον έτερο αλγόριθμο, καθότι σε αυτή την περίπτωση η βάση δεδομένων εκτελεί πολύ περισσότερες πράξεις και έτσι αυτές θα εκτελεστούν παράλληλα σε πολλούς κόμβους.

Επιπλέον, η χρησιμοποίηση της μνήμης βελτιώνεται μέχρι και 97% στον αλγόριθμο₃ σε σχέση με τον αλγόριθμο₁, πράγμα το οποίο σημαίνει ότι σε πολύ μεγάλα datasets, αυτός ο αλγόριθμος πλεονεχτεί, ειδικά όταν το σύστημα που τον εκτελεί χρησιμοποιεί πόρους σε άλλες διεργασίες. Τέλος μια παρατήρηση ακόμα, είναι πως το φόρτο εργασίας του αλγορίθμου₄ δεν είναι αρκετό ώστε η ταυτόχρονη επεξεργασία να επιφέρει χρονικές βελτιώσεις.

4.3 Συμπεράσματα και μελλοντικές εργασίες

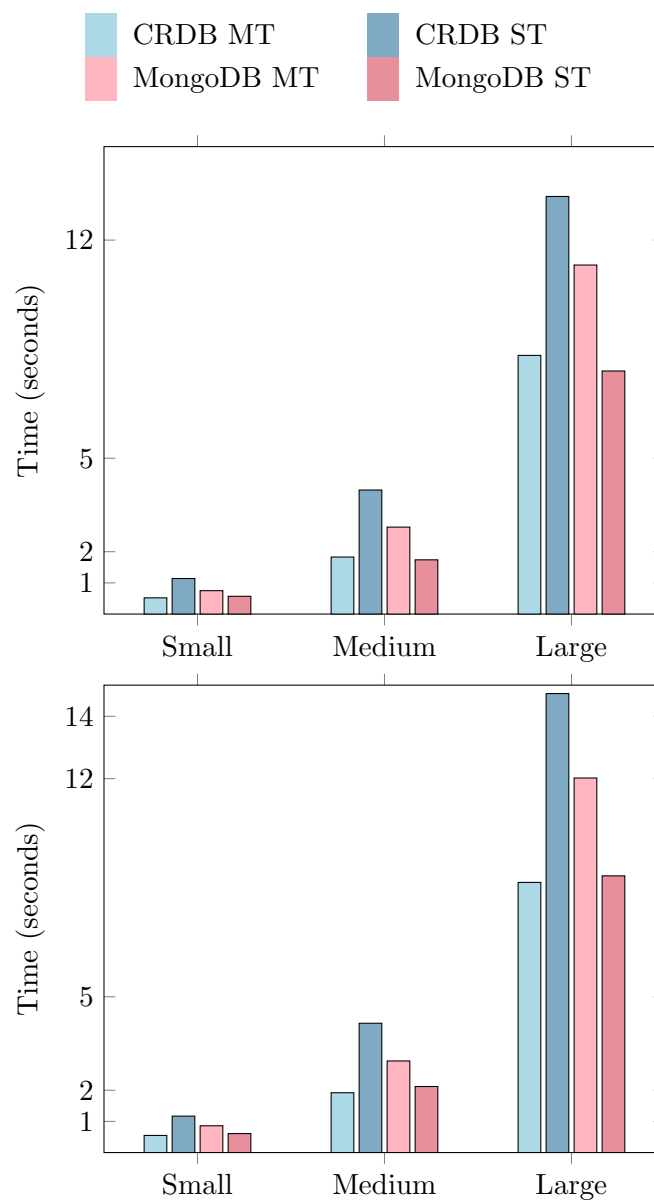
Λαμβάνοντας υπόψιν τα αποτελέσματα των συγκρίσεων των δύο υλοποιήσεων, μπορούμε να συμπεράνουμε πως εν γένει στην καλύτερη περίπτωση η υλοποίηση με την CockroachDB θα έχει παρόμοιες χρονικές εκτελέσεις των ερωτημάτων με αυτές της MongoDB. Παρόλ' αυτά σε αυτό το σημείο, να επισημάνω πως αυτές οι δύο βάσεις παρουσιάζουν θεμελιώδη διαφορά στην αρχιτεκτονική τους και άρα η CockroachDB υπερτερεί σε άλλους τομείς. Όσον αφορά τα μοντέλα MT και ST της CockroachDB, το Multiple Table φαίνεται πως έχει καλύτερες επιδόσεις σε όλες τις περιπτώσεις που εξετάστηκαν στην εργασία αυτή.

Ωστόσο, προς επαλήθευση των παραπάνω και ολοκλήρωση της έρευνας, απαιτούνται και μετρήσεις από ένα αντίστοιχο κατανεμημένο σύστημα, πράγμα το οποίο δεν ήταν εφικτό κατά τη διάρκεια εκπόνησης της εργασίας.

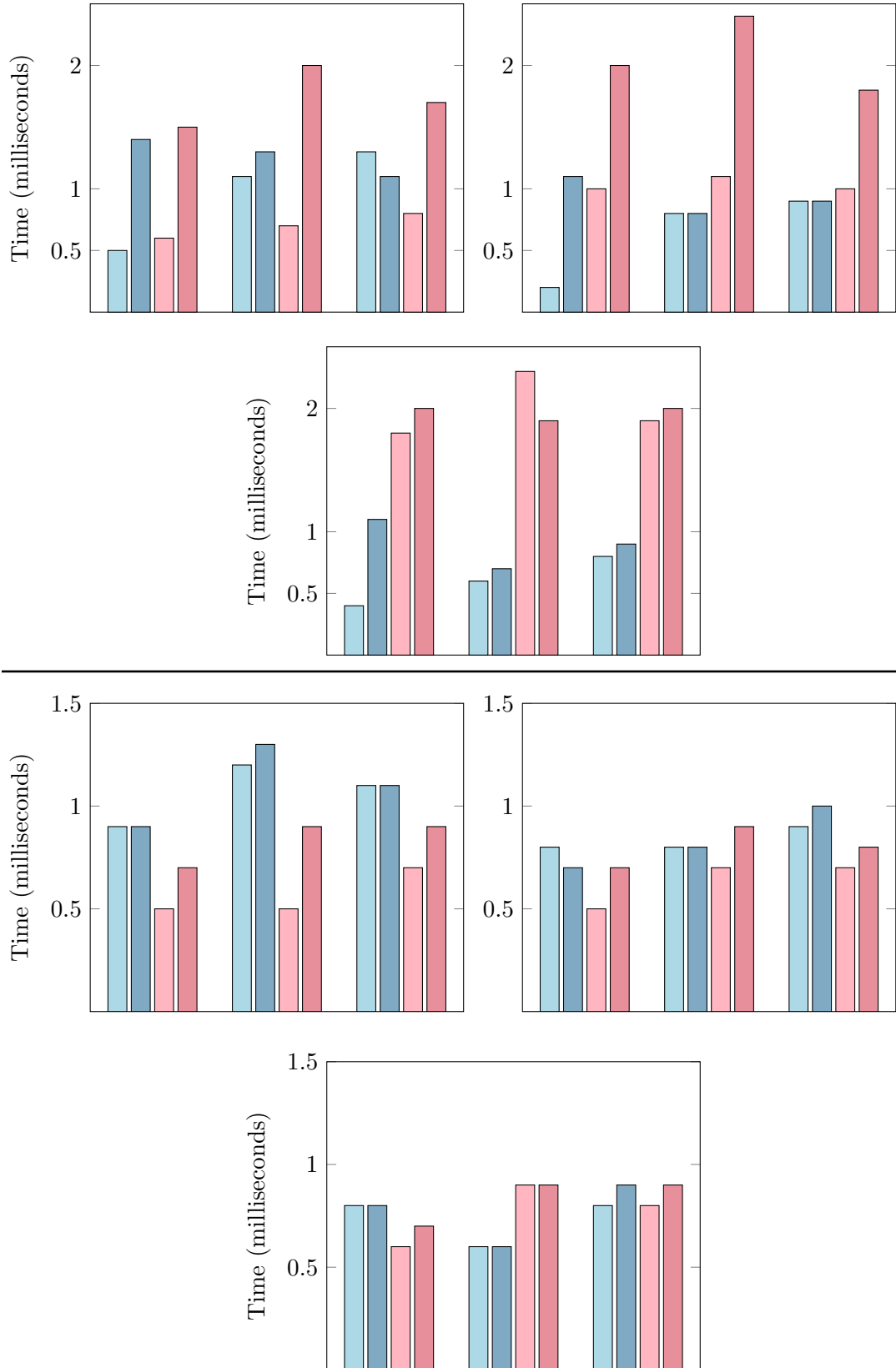
Όσον αφορά τις μελλοντικές εργασίες, θα ήταν εύλογο να διενεργηθεί μια έρευνα για την πρακτική υλοποίηση του θεωρητικού μοντέλου HiNode, και πως αυτό θα μπορούσε να εφαρμοστεί σε μια graph βάση δεδομένων. Αναφορικά με τα μοντέλα της παρούσας υλοποίησης, σίγουρα υπάρχουν περιθώρια βελτίωσης τόσο στο σχήμα και τον τύπο που αποθηκεύονται τα δεδομένα όσο και στην μετέπειτα διαχείριση αυτών. Επιπλέον μια ακόμη αλλαγή που θα μπορούσε να γίνει είναι η χρήση πιο εξειδικευμένων indexes και optimizations στα ερωτήματα που γίνονται στη βάση δεδομένων. Με το τρόπο αυτό ενδέχεται τα ερωτήματα να εκτελούνται ταχύτερα στη βάση δεδομένων, καθώς επίσης και οι μετέπειτα πράξεις που γίνονται από τον χρήστη και το υπολογιστικό κόστος να μειωθούν ακόμα περισσότερο.

Παράρτημα Α΄

Περισσότερες συγκρίσεις



Σχήμα Α΄.1: Degree Distribution (1/3 και 2/3 του χρόνου)



Σχήμα Α'2: One Hop Neighborhood (1/3 και 2/3 του χρόνου)

Βιβλιογραφία

- [1] Codd, E. F., ‘A relational model of data for large shared data banks’, τόμ. 13, αρθμ. 6, 377–387, 1970. διεύθυν.: <https://doi.org/10.1145/362384.362685>.
- [2] Gilbert Seth and Lynch Nancy, ‘Perspectives on the CAP Theorem’, *Computer*, τόμ. 45, αρθμ. 2, σσ. 30–36, 2012.
- [3] James C. Corbett and Jeffrey Dean and Michael Epstein and Andrew Fikes and Christopher Frost and JJ Furman and Sanjay Ghemawat and Andrey Gubarev and Christopher Heiser and Peter Hochschild and Wilson Hsieh and Sebastian Kanthak and Eugene Kogan and Hongyi Li and Alexander Lloyd and Sergey Melnik and David Mwaura and David Nagle and Sean Quinlan and Rajesh Rao and Lindsay Rolig and Dale Woodford and Yasushi Saito and Christopher Taylor and Michal Szymaniak and Ruth Wang, ‘Spanner: Google’s Globally-Distributed Database’, στο *OSDI*, 2012.
- [4] Jeff Shute and Radek Vingralek and Bart Samwel and Ben Handy and Chad Whipkey and Eric Rollins and Mircea Oancea and Kyle Littlefield and David Menestrina and Stephan Ellner and John Cieslewicz and Ian Rae and Traian Stancescu and Himani Apte, ‘F1: A Distributed SQL Database That Scales’, στο *VLDB*, 2013.
- [5] Pavlo Andrew and Aslett Matthew, ‘What’s Really New with NewSQL?’, *SIGMOD Rec.*, τόμ. 45, αρθμ. 2, 45–55, Σεπτ. 2016. διεύθυν.: <https://doi.org/10.1145/3003665.3003674>.
- [6] Walker Jim and Gienow Michelle. ‘What is distributed SQL? An evolution of the database’. (Σεπτ. 2023), διεύθυν.: <https://www.cockroachlabs.com/blog/what-is-distributed-sql/>.
- [7] Mattis Peter and Walker Jim, *Yugabyte DB vs CockroachDB*, 1η έκδοση. Cockroach Labs, 2020, σ. 23.
- [8] Harrison Guy and Seldess Jesse and Darnell Ben, *CockroachDB: The definitive guide: Distributed data at scale*, 1η έκδοση. O’Reilly Media, Incorporated, 2022, σ. 460.
- [9] Pitoura Evaggelia, ‘Historical Graphs: Models, Storage, Processing: 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2–7, 2017, Tutorial Lectures’, στο Ιούλ. 2018, σσ. 84–111.

- [10] Koloniari Georgia and Souravlias Dimitris and Pitoura Evaggelia, ‘On Graph Deltas for Historical Queries’, Φεβ. 2013.
- [11] Khurana Udayan and Deshpande Amol, ‘Efficient snapshot retrieval over historical graph data’, στο *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, σειρά ICDE ’13, USA: IEEE Computer Society, 2013, 997–1008. διεύθυν.: <https://doi.org/10.1109/ICDE.2013.6544892>.
- [12] Khurana Udayan and Deshpande Amol, ‘Storing and Analyzing Historical Graph Data at Scale’, Σεπτ. 2015.
- [13] Kosmatopoulos Andreas and Tsihclas Kostas and Gounaris Anastasios and Sioutas Spyros and Pitoura Evaggelia, ‘HiNode: an asymptotically space-optimal storage model for historical queries on graphs’, *Distrib. Parallel Databases*, τόμ. 35, αρθμ. 3–4, 249–285, Δεκ. 2017. διεύθυν.: <https://doi.org/10.1007/s10619-017-7207-z>.
- [14] Kosmatopoulos Andreas and Gounaris Anastasios and Tsihclas Konstantinos, ‘Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data’, *Computing*, τόμ. 101, Δεκ. 2019. ΔΟΙ: 10.1007/s00607-019-00715-6.
- [15] Spitalas Alexandros and Gounaris Anastasios and Tsihclas Kostas and Kosmatopoulos Andreas, ‘Investigation of Database Models for Evolving Graphs’, στο *28th International Symposium on Temporal Representation and Reasoning (TIME 2021)*, Combi Carlo and Eder Johann and Reynolds Mark, επιμελητής, σειρά Leibniz International Proceedings in Informatics (LIPIcs), τόμ. 206, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 6:1–6:13, ISBN: 978-3-95977-206-8. ΔΟΙ: 10.4230/LIPIcs.TIME.2021.6. διεύθυν.: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TIME.2021.6>.