# Σύγκριση Απόδοσης Κατανεμημένων Graph Databases

*Ορφέας Φιλιππόπουλος, el18082*
*ECE*
*NTUA*
*Athens, Greece*
*orfeasfil2000@gmail.com*

Δέσποινα Γραμμένου, el18061
*ECE*
*NTUA*
*Athens, Greece*
*dgrammenou@yahoo.com*

Θοδωρής Παπαρρηγόπουλος, el18040
*ECE*
*NTUA*
*Athens, Greece*
*paparrigopoulosthodoris@gmail.com*

Το κώδικα μας μπορείτε να τον βρείτε στο παρακάτω λινκ
https://github.com/thodpap/CompareGraphDatabases

## **I.** Σκοπός της εργασίας

Σκοπός της εργασίας αυτής είναι η ανάπτυξη ενός συστήματος που επιτρέπει τη σύγκριση απόδοσης κατανεμημένων Graph Databases. Πιο συγκεκριμένα θα συγκρίνουμε τις εξής δύο Graph Database:

- **Neo4j** [neo4j], Neo4j company,
  ACID compliance DBMS,
  Graph Query Language: **Cypher,**
  **Open Source!**
- **Hugegraph** [hugegraph], Apache SW company
  Fully compatible with **Gremlin** query language
  **Open Source!**

Η σύγκριση αυτή θα γίνει πάνω σε:

- **ίδια σύνολα δεδομένων**
- **ίδιο φόρτο εργασίας**
- **ίδιο περιβάλλον εκτέλεσης (Hardware)**

Το σύστημα αυτό εκτελεί τα προκαθορισμένα ερωτήματα σε κάθε σύστημα υπό δοκιμή (SUT – System Under Test) και μετρά το χρόνο που είναι απαραίτητος για την ολοκήρωση μιας συγκεκριμένης εργασίας, υπολογίζει για καθένα από αυτά τις καθορισμένες μετρικές απόδοσης και παράγει γραφικές παραστάσεις για κάθε μετρική / σύνολο SUT.

Πιο συγκεκριμένα, οι λειτουργίες – εργασίες που επιθυμούμε να συγκρίνουμε είναι:

- **δημιουργία (CREATE)**
- **ανάγνωση (READ)**
- **ενημέρωση (UPDATE)**
- **διαγραφή (DELETE)**

Επιπροσθέτως οι μετρικές που θα υπολογίζονται είναι:

- Μέσος χρόνος εκτέλεσης ερωτήματος: Μέσος χρόνος για την εκτέλεση ενός μεμονωμένου ερωτήματος τύπου x πολλές φορές έναντι του SUT.

- Ελάχιστος/Μέγιστος χρόνος εκτέλεσης ερωτήματος: Χρόνος εκτέλεσης κάτω και άνω ορίου για ερωτήματα τύπου x έναντι του SUT.

Όσον αφορά τα σύνολα δεδομένων που χρησιμοποιήθηκαν είναι της τάξης **10^n** με **n = 1, 2, 3, 4, 5, 6** κόμβων – nodes.

Τα βήματα υλοποίησης που ακολοθήθηκαν είναι:

1. Εγκατάσταση και εξοικείωση με τα συστήματα SUT
2. Καθορισμός συνόλου δεδομένων και ερωτημάτων
3. Σχεδίαση και υλοποίηση συστήματος
4. Υλοποίηση και δοκιμές ελέγχου

Τα οποία και θα αναλυθούν λεπτομερώς παρακάτω.

## **II.** Περιγραφή και Ανάλυση των βημάτων για το "στήσιμο" των συστημάτων και των πειραμάτων

### **A. Στήσιμο Συστημάτων (SUT)**

Όσον αφορά το **στήσιμο** των συστημάτων θα περιγράψουμε προφανώς ξεχωριστά για τα δύο SUTS (**Hugegraph, Neo4j)**. Να σημειωθεί ότι το Hugegraph είναι compatible **μόνο με Linux** ενώ το Neo4j είναι compatible και με **Linux και με Windows**. Για το λόγο αυτό, το development στο Hugegraph έγινε αποκλειστηκά σε Linux (Ubuntu 20.04) ενώ το development στο Neo4j έγινε τόσο σε Linux (Ubuntu 20.04) όσο και σε Windows (10).

Ξεκινώντας λοιπόν με το **Hugegraph** απαιτούσε αρκετά βήματα για το στήσιμό του. Πρώτον χρειάστηκε να κατεβάσουμε τη Java στον υπολογιστή μας, μιας και το Hugegraph είναι γραμμένο στη γλώσσα αυτή. Πιο συγκεκριμένα είναι compatible με **JDK** version 11 και 8. Ωστόσο για λόγους που θα σχολιάσουμε αργότερα επιλέξαμε το **JDK 11**. Αφού λοιπόν εγκαταστήσαμε τη Java, στη συνέχεια εγκαταστήσαμε, ακολουθώντας το **link**, το Core component του Hugegraph, τον Hugegraph Server (ακολουθώντας τον τρόπο *Source code compilation)*. Οι οδηγίες ήταν αρκετά σαφής και η εγκατάσταση πραγματοποήθηκε αρκετα γρήγορα. Στη συνέχεια, προκειμένου να μπορέσουμε να χρησιμοποιήσουμε το Hugegraph έπρεπε να επιλέξουμε το Backend το οποίο θα χρησιμοποιήσουμε. Πιο αναλυτικά τα Backends που μας παρέχει το Hugegraph είναι:

- **Memory**
- **RocksDB (Key-value Store)**
- **Cassandra (Wide Column Store)**
- **ScyllaDB,** Primary database model **Key-value Store,** Secondary database model **Wide Column Store**

- **HBase (Wide Column Store)**

Με βάση το **https://db-engines.com/en/ranking** παρατηρούμε πως **Overall** η καλύτερη επιλογή είναι η Cassandra με αρκετά μεγάλη διαφορά από τις υπόλοιπες. Πιο αναλυτικά η Cassandra έχει 113.79 score και η δεύτερη καλύτερη, HBase, έχει μόλις 37.62 score. Θα μπορούσαμε να χρσιμοποιήσουμε ως Backend πχ τη RocksDB που είναι optimized για γρήγορο storage, ωστόσο λόγο εύκολης εγκατάστασης αλλά και πολύ καλής κατάταξης με βάση το παραπάνω link επιλέξαμε ως Backend τη Cassandra. Για την εγκατάσταση της Cassandra ακοκουθήσαμε το **Installing the binary tarball** κατεβάζοντας την 4.0 version. Να σημειωθεί ότι η Cassandra 4.0 είναι και αυτή compatible με το JDK 11 οπότε δεν χρειαζόταν κάποια περαιτέρω ενέργεια.

Αφού λοιπόν έχουμε εγκαταστήσει και το Hugegraph-Server αλλά και το Backend το οποίο επιλέξαμε, στη συνέχεια μπορούμε να έχουμε το σύστημα μας up-and-running. Για να γίνει αυτό πήγαμε στο bin folder της Cassandra και εκτελέσαμε την εντολή **./cassandra**. Ακολούθως, κάναμε configure το Backend του Hugegraph-Server μέσω του αρχείου conf/hugegraph.properties προκειμένου να είναι η Cassandra. Τέλος, αφού τα κάναμε όλα αυτά πήγαμε στο bin folder του Hugegraph-Server και εκτελέσαμε τις εξής εντολές:

- **./init-store.sh**
- **./start-hugegraph.sh**

Να σημειωθεί ότι κάναμε configure (και πάλι μέσω του hugegraph.properties αρχείου) τον Hugegraph-Server να “ακούει” στη πόρτα 8081. Η Cassandra by default ακούει στη πόρτα 9042.

Εκτελώντας τις εντολές:

- **sudo netstat -tulpn | grep 9042**
- **sudo netstat -tulpn | grep 8081**

Επιβεβαιώσαμε πως και η Cassandra αλλά και ο Hugegraph-Server είναι up-and-running.

Στη συνέχεια όσον αφορά το **Neo4j**, η εγκατάσταση ήταν αρκετά πιο απλή, τόσο για Windows όσο και για Linux. Πιο συγκεκριμένα εγκαταστήσαμε τη Desktop εφαρμογή (δεν χρησιμοποιήσαμε online instances) ακολουθώντας το **link**. Η εγκατάσταση ήταν αρκετά απλή και γρήγορη και δεν αντιμετωπίσαμε κάποια δυσκολία.

Το Neo4j αντίστοιχα ακούει στη πόρτα 7687.

Αξιοσημείτωτο είναι το γεγονός ότι το Neo4j, και πάλι με βάση το **https://db-engines.com/en/ranking**, έχει 53.51 overall score, ενώ με βάση το **https://db-engines.com/en/system/Neo4j** είναι το καλύτερο GraphDBMS!

**Β. Περιγραφή και Ανάλυση Πειραμάτων**

Όσον αφορά τα πειράματα που πραγματατοποιήσαμε αφορούν τις τρεις εξής μεθόδους, σύμφωνα με την εκφώνηση,:

- **δημιουργία (CREATE)**
- **ανάγνωση (READ)**
- **ενημέρωση (UPDATE)**
- **διαγραφή (DELETE)**

Ωστόσο, προκειμένου να κάνουμε τα παραπάνω πειράματα, χρειαζόμαστε και τα κατάλληλα datasets. Από τη στιγμή που τα Systems Under Test είναι Graph DBMSs, αποφασίσαμε πως τα datasets θα δείχνουν σχέσεις μεταξύ ανθρώπων (γράφος). Πιο συγκεκριμένα κάθε **κόμβος** (τον ονομάσαμε **Person**) είναι και ένας **άνθρωπος** και κάθε **edge** (το ονομάσαμε **KNOWS**) **υποδηλώνει τη σχέση** του με κάποιον **άλλον άνθρωπο** (οι σχέσεις αυτές είναι αμφίδρομες προφανώς, άρα εάν ένας άνθρωπος ξέρει έναν άλλον, τότε και ο άλλος άνθρωπος ξέρει τον πρώτο, αυτό σημαίνει δύο ακμές μεταξύ κάθε ανθρώπων που συνδέονται). Κάθε κόμβος έχει ως primary key το name του (property του κόμβου), που απλά είναι ένας μοναδικός αριθμός που του έχει ανατεθεί, ενώ επίσης έχει ακόμα ένα property που ονομάζεται age και υποδηλώνει την ηλικία του. Αντίστοιχα, κάθε edge συνδέει ανθρώπους και υποδηλώνει, όπως προαναφέραμε, σχέσεις μεταξύ των ανθρώπων και έχει ένα property που το ονομάσαμε years που εκφράζει το πόσα χρόνια υπάρχει η αντίστοιχη σχέση μεταξύ των αθρώπων.

Κατά αυτό το τρόπο “οργανώσαμε” τα δεδομένα μας, τώρα μας μένει να τα βρούμε/παράξουμε. Τα dataset μας είναι text αρχεία που σε κάθε γραμμή περιλαμβάνουν 2 κόμβους και υποδηλώνει την σχέση μεταξύ των δύο αυτών κόμβων.

Για **αριθμό κόμβων ίσο με 10** παράξαμε, με το script create_data.py, εμείς τα δεδομένα (node_10.txt στο repo) καθώς δεν καταφέραμε να βρούμε online dataset με 10 κόμβους. Ο τρόπος που παρήχθηκαν ήταν πρώτον να δημιουργήσουμε 10 κόμβους με names 1-10 και να έχουμε συνολικά 30 σχέσεις μεταξύ τους (άρα 30 γραμμές στο node_10.txt αρχείο). Ο γράφος αυτός είναι αρκετά πυκνός μιας και ο μέγιστος αριθμός σχέσεων που θα μπορούσαμε να έχουμε είναι n*(n-1)/2 = 5*9 = 40.

Για **αριθμό κόμβων ίσο με 100** παράξαμε, με το script create_data.py και πάλι εμείς τα δεδομένα (node_100.txt το repo) καθώς και πάλι δεν καταφέραμε να βρούμε online dataset με 100 κόμβους. Ο τρόπος που παρήχθηκαν ήταν πρώτον να δημιουργήσουμε κόμβους με names 1-100 και να έχουμε συνολικά 600 σχέσεις. Και πάλι ο γράφος είναι σχετικά πυκνός καθώς το 600 είναι συγκρίσιμο με τον μέγιστον αριθμό σχέσεων που μπορούν να υπάρξουν (συνολικά 100*49/2 = 2450).

Για **αριθμό κόμβων ίσο με 1000, 10000, 100000 και 1000000** πήραμε έτοιμα datasets (τα οποία ακολουθούν το ίδιο format με τα προηγούμενα, δηλαδή οι γραμμές αποτελούνται από 2 κόμβους κα υποδηλώνουν σχέσεις μεταξύ τους) από https://networkrepository.com/socfb.php. Το site αυτό παρέχει online datasets τα οποίο υποδηλώνουν social networks εμπνευσμένα από το facebook. Ωστόσο δεν παρείχε datasets με ακριβώς πχ 1000 κόμβους και για το λόγο αυτό με το script write_files.ipynb κάναμε κατάλληλες τροποποιήσεις. Πιο αναλυτικά, **για τους 1000 κόμβους** χρησιμοποιήσαμε το socfb-Haverford76 το οποίο έχει 1400 κόμβους και 60000 σχέσεις. Ο τρόπος που κάναμε downgrade στους 1000 κόμβους ήταν ότι όποτε διαβάζαμε στο παραπάνω αρχείο κόμβο με αριθμό μεγαλύτερο του 1000 τότε με τυχαίο τρόπο τον αντικαταστούσαμε με κάποιον άλλον με αριθμό 1-1000. Κατά αυτό το τρόπο πλέον

κατασκευάσαμε ένα dataset (node_1000.txt) . **Για τους 10000 κόμβους** χρησιμοποιήσαμε το socfb-Bingham82 το οποίο έχει ακριβώς 10000 κόμβους και 363000 σχέσεις, οπότε δεν χρειάστηκε κάποια περαιτέρω επεξεργασία. **Για τους 100000 κόμβους** χρησιμοποιήσαμε το socfb-wosn-friends το οποίο έχει 63.7K κόμβους και 1.3M σχέσεις. Για να κάνουμε upgrade στους 100000 κόμβους απλά προσθέσαμε random σχέσεις στις οποίες τουλάχιστον ο ένας κόμβος έχει name μεταξύ 63701 και 100000. Επίσης, με μία απλή validation συνάρτηση επιβεβαιώσαμε πως κάθε κόμβος ανήκει σε τουλάχιστον μία σχέση. Τέλος **όσον αφορά τους 1000000** χρησιμοποιήσαμε το socfb-A-anon, το οποίο έχει 3.1M κόμβους και 23.7M σχέσεις. Για να κάνουμε downgrade και πάλι τους κόμβους από 3.1M σε 1M, όποιο κόμβο βρίσκαμε σε κάποια σχέση που δεν ανήκε στο διάστημα 1-1M τον αντικαθιστούσαμε με κόμβο στο διάστημα αυτό.

Αφού αναλύσαμε λεπτομερώς όλη τη προεπεξεργασία που κάναμε πάνω στα dataset μας, τώρα μπορούμε να περιγράψουμε και να αναλύσουμε τα πειράματα που κάναμε.

Όσον αφορά λοιπόν τα πειράματα – δοκιμές που πραγματοποιήσαμε, ήταν ίδια προφανώς μεταξύ Hugegraph και Neo4j, ωστόσο εκμετεαλλευτήκαμε τη δυνατότητα του Hugegraph για κάποες περαιτέρω δοκιμές τις οποίες θα αναλύσουμε παρακάτω.

Πιο αναλυτικά, τα πειράματα που αρχικά κάναμε ήταν τα εξής:

- **CREATE** όλα τα δεδομένα (δημιουργία πρώτα όλων των κόμβων και μετά όλων των σχέσεων – ακμων μεταξύ των κόμβων). Χρονομετρούσαμε κάθε query και από εκεί εύκολα υπολογίσαμε το mean, max και min χρόνο τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους.
- **READ** όλα τα δεδομένα (ανάγνωση όλων των κόμβων και όλων των σχέσεων-ακμών μεταξύ των κόμβων). Και πάλι χρονομετρούσαμε το κάθε query και εύκολα υπολογίσαμε το mean, max και min χρόνο τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους.
- **UPDATE** όλα τα δεδομένα (ενημέρωση της ηλικίας (age property) του κάθε κόμβου και στη συνέχεια ενημέρωση των χρόνων (years property) κάθε ακμής). Και πάλι χρονομετρούσαμε το κάθε query και εύκολα υπολογίσαμε το mean, max και min χρόνο τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους.
- **DELETE** όλα τα δεδομένα (διαγραφή όλων των σχέσεων-ακμών και στη συνέχεια διαγραφή όλων των κόμβων). Και πάλι χρονομετρούσαμε το κάθε query και εύκολα υπολογίσαμε το mean, max και min χρόνο τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους.

Τα παραπάνω πειράματα είχαν ικοποιητικούς χρόνου για τα μικρά datasets (μέχρι και 1000 κόμβους), ωστόσο από τους 10000 κόμβους και μετά οι συνολικοί χρόνοι άρχισαν να γίνονται απαγορευτικά μεγάλοι, τόσο στο Hugegraph όσο και στο Neo4j (πχ για 1000000 στο Hugegraph ήθελε συγκεκριμένα 35 ώρες (σχεδόν μιάμιση μέρα) για να ολοκληρωθεί μόνο το insert των δεδομένων, λόγο του τεράστιου όγκου σχέσεων-ακμών). Για το λόγο αυτό έπρεπε να σκεφτούμε, τόσο στο Hugegraph όσο και στο Neo4j, τρόπους προκειμένου να πραγματοποιούμε τις παραπάνω ενέργειες πιο γρήγορα και πιο αποδοτικά. Διαβάζοντες αρκετές διαφορετικές υλοποιήσεις και προσεγγίσεις, τελικά καταλήξαμε σε μία κοινή λογική και στα δύο Graph DBMSs. Πιο συγκεκριμένα αποφασίσαμε να κάνουμε τα παραπάνω requests (ή όσα μπορούσαμε) σε **batches**, για παράδειγμα να κάνουμε batched insert των δεδομένων (πχ 100 κόμβους με ένα insert-create request ή 1000 σχέσεις- edges με ένα insert-create request). Όσον αφορά το Hugegraph αντιμετωπίσαμε κάποιους περιορισμούς καθώς υποστηρίζει μόνο batched creation-insertion και batched update δεδομένων. Αυτούς τους περιορισμούς τους ακολουθήσαμε και στο Neo4j για λόγους σύγκρισης των δύο DBMSs, εάν και εκεί μας δινόναν η δυνατότα να κάνουμε batched αναγνώσεις και batched διαγραφές. Με βάση τα παραπάνω λοιπόν προστέθηκαν ακόμα δύο πειράματα πέρα από τα αρχικά:

- **Batched CREATE** όλων των δεδομένων (δημιουργία όλων των κόμβων σε batches και μετά όλων των σχέσεων – ακμών μεταξύ των κόμβων). Χρονομετρούσαμε κάθε batched query τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους. Από εκεί διαιρούσαμε με το μέγεθος το batch για να βγάλουμε τους χρόνους και κάθε σχέση – ακμή και για κάθε κόμβο αντίστοιχα.
- **Batched UPDATE** όλων των δεδομένων (ενημέρωση της ηλικίας (age property) των κόμβων σε batches και στη συνέχεια ενημέρωση των χρόνων (years property) των ακμών σε batches). Χρονομετρούσαμε κάθε batched query τόσο για τις ακμές-σχέσεις όσο και για τους κόμβους. Από εκεί διαιρούσαμε με το μέγεθος το batch για να βγάλουμε τους χρόνους και κάθε σχέση – ακμή και για κάθε κόμβο αντίστοιχα.

Αφού λοιπόν εξηγήσαμε και τα batches στη συνέχεια θα αναφέρουμε και άλλα πειράματα που κάναμε για λόγους πληρότητας αλλά κυρίως για λόγους δικού μας πειραματισμού.

Πιο συγκεκριμένα πειραματιστήκαμε και με το κομμάτι των **traversers**, δηλαδή κάναμε queries για το Shortest Path (Reachability Queries) μεταξύ δύο κόμβων τόσο στο Hugegraph όσο και στο Neo4j. Επίσης στο Hugegraph πειραματιστήκαμε τόσο με το K-out (Adjacency Queries category) όσο και με το K-Neighbors (Adjacency Queries category). Για περισσότερες πληροφορίες μπορείτε να επισκεφτείτε:

https://hugegraph.apache.org/docs/clients/restful-api/traverser/

όπου και εξηγεί αναλυτικά (χρειάζεται μετάφραση από τα Κινέζικα) το πως λειτουργούν οι παραπάνω traversers.

Μία γρήγορη αναφορά είναι:

- Shortest Path: το συντομότερο μονοπάτι μεταξύ δύο κόμβων.
- K-out: γείτονες που απέχουν ακριβώς N βήματα μακριά
- K-neighbor: όλοι οι γείτονες που απέχουν το πολύ N βήματα μακριά

Αφού πειραματηστίκαμε και με τους traversers που προαναφέραμε, τέλος κάναμε και κάποιες δοκιμές, στο Hugegraph, και με το **Gremlin** (Graph Query Language με το οποίο το Hugegraph είναι fully compatible). Θα μπορούσαμε να πειραματιστούμε και στο Neo4j με το Gremlin καθώς υπάρχει διαθέσιμο plug in, αλλά εν τέλη κάτι

τέτοιο δεν το κάναμε (εξηγείται στο τέλος το γιατί) και περιοριστήκαμε στη χρήση Cypher που μας φάνηκε ιδιαίτερα εύχρηστο και ενδιαφέρον.

Όσον αφορά το Gremlin λοιπόν στο Hugegraph, υλοποιήσαμε όλες τις βασικές λειτουργίες, δηλαδή υλοποιήσαμε την δημιουργία, ανάγνωση, ενημέρωση και διαγραφή τόσο των κόμβων όσο και των σχέσεων - ακμών. Προκειμένου να τρέξουμε τα Gremlin queries που υλοποιήσαμε για τις βασικές λειτουργίες έπρεπε να "χτυπήσουμε" ένα συγκεκριμένο URL που μας παρείχε ο Hugegraph-Server και να έχουμε ως δεδομένα αυτά τα Gremlin queries (data field στο HTTP request). Ωστόσο, διαβάζοντας περαιτέρω το documentation του Hugegraph-Server ανακαλύψαμε πως μπορούμε ασύγχρονα να υποβάλλουμε Gremlin queries (tasks) πιστεύοντας πως έτσι θα επιταχυνθεί η διαδικασία εισαγωγής, ανάγνωσης, ενημέρωσης και διαγραφής των δεδομένων. Αφού υλοποιήσαμε και αυτό, παρατηρήσαμε πως ο συνολικός χρόνος εκτέλεσης, σε μερικές περιπτώσεις μειώθηκε, σε άλλες έμεινε ο ίδιος και σε άλλες αυξήθηκε, κάτι που μας εξέπληξε και ψάξαμε περαιτέρω το λόγο (θα παρουσιαστεί παρακάτω). Ωστόσο αυτό θα συζητηθεί πιο αναλυτικά σε επόμενο κεφάλαιο. Σημαντικό είναι να αναφερθεί πως καθόλη τη διάρκεια που υλοποιούσαμε τα Gremlin Queries, συμβουλευόμασταν το Gremlin-Graph-Guide.

## **III** Περιγραφή της υποδομής και του software που χρησιμοποιήθηκε

**Υποδομή:**

Όλο το development έγινε στο Visual Studio Code έχοντας οργανώσει κατάλληλα τους κώδικες σε folders και αρχεία. Επίσης χρησιμοποιήσαμε το github δημιουργώντας ένα github repo (έχει δοθεί κατάλληλο link) με δύο branches, ένα για το Hugegraph και ένα για το Neo4j, για σωστή και αποδοτική συνεργασία μεταξύ των μελών της ομάδας μας. Άμα λοιπόν περιηγηθείτε στο κάθε branch του repo μας αντίστοιχα μπορείτε να δείτε την δομή των project μας η οποία είναι αρκετά σαφής και απλή. Οποιαδήποτε απορία περί της οργάνωσης και της δομής του project μπορεί να λυθεί κατά τη διάρκεια της ζωντανής παρουσίασης.

**Software:**

Η προγραμματιστική γλώσσα – Software που χρησιμοποιήθηκε είναι αποκλειστηκά και μόνο η Python. Ωστόσο αυτή η επιλογή δεν είναι τυχαία, υπάρχει λόγος και για τα δύο Graph DBMSs:

**Hugegraph:**

Για το Hugegraph, με μία πρώτη ματιά ίσως είναι πιο "σωστή" επιλογή να χρησιμοποιηθεί η Java ως προγραμματιστική γλώσσα – Software. Ωστόσο, επειδή είμαστε αρκετά εξοικειωμένοι με τη Python προσπαθήσαμε να βρούμε εναλλακτικές λύσεις. Με ένα γρήγορο search στα search engines βρήκαμε το repo https://github.com/tanglion/PyHugeGraph το οποίο παρέχει έτοιμο driver (τον PyHugeGraph driver) ο οποίος αυτοματοποιεί τη διαδικασία των requests στο REST API που παρέχει ο Hugegraph-Server. Ποιο συγκεκριμένα παρέχει συναρτήσεις όπως create_property (που τη χρησιμοποιήσαμε για παράδειγμα να ορίσουμε τα properties age και years), insert_vertex, insert_edge, insert_multivertex (για batch insertions των vertices), insert_multiedge (για batch insertions των edges), update_vertex_properties (για update πχ του age κάθε node), update_edge_properties (για update πχ του years property του κάθε edge), read_vertex_by_id, read_edge_by_id (για ανάγνωση των κόμβων και των edges αντίστοιχα με βαση το ID που τους έχει ανατεθεί), read_vertex_by_properties (πχ ανάγνωση κόμβου με βάση το name του, που είναι και το primary key) read_edge_by_properties,delete_vertex_by_id και αντίστοιχα delete_edge_by_id, ακόμα έχει delete_vertex_by_properties, delete_edge_by_properties και πολλά άλλα. Αυτό είναι το βασικό API που μας παρέχει, ωστόσο παρέχει ακόμα ένα πλήθος συναρτήσεων για τους traversers που αναφέραμε νωρίτερα, όπως traverser_shortest_path (βρίσκει το shortest path μεταξύ δύο κόμβων, traverser_kout (βρίσκει τους κόμβους που απέχουν ακριβώς N steps μακριά από έναν κόμβο) και τέλος traverser_kneighbor (βρίσκει όλους τους κόμβους που απέχουν το πολύ N steps μακριά από έναν κόμβο).

Πέρα από τις συναρτήσεις που μας παρέχει ο driver αυτός, χρειάστηκε να επεκτείνουμε το functionality του προσθέτοντας κάποιες δικές μας συναρτήσεις. Πιο συγκεκριμένα προσθέσαμε τη συνάρτηση update_multi_vertex η οποία κάνει update τα properties ενός batch κόμβων και τη συνάρτηση update_multi_edge η οποία κάνει update τα properties ενός batch σχέσεων - ακμών.

Βέβαια, όλες οι παραπάνω συναρτήσεις εκφυλίζονται σε απλά HTTP requests (put, get, update, delete etc) στο REST API του Hugegraph-Server και για το λόγο αυτό χρησιμοποιήσαμε (όπως και ο δοσμένος driver) το package *requests* για όποια συνάρτηση προσθέσαμε στο driver αλλά και όποια άλλη συνάρτηση γενικά φτιάξαμε (πχ για να πειραματιστούμε με το gremlin φτιάξαμε δικές μας συναρτήσεις εκτός driver η οποίες προφανώς χρησιμοποιούσαν και αυτές το package *requests* της Python).

Τέλος, αξιοσημείωτο είνα το γεγονός ότι χρησιμοποιώντας το REST API του Hugegraph-Server είναι ο πιο άμεσος τρόπος προκειμένου να στέλνουμε requests και αυτό φαίνεται από τη παρακάτω φωτογραφία η οποία παρουσιάζει το stack του Hugegraph:

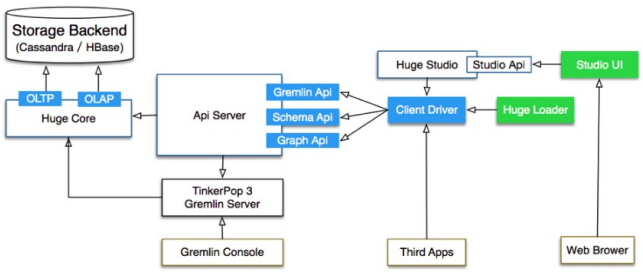


Figure 1. Stack hugegraph

Το component το οποίο "χτυπάμε" εμείς με τη υλποίηση μας είναι ο API Server που φαίνεται στη φωτογραφία. Για περισσότερες επεξηγήσεις περί των components μπορείτε να βρείτε στο Architecture_Overview.

**Neo4j:**

Όσον αφορά το Neo4j, τα περισσότερα documentations που βρήκαμε ακόμα, και στο original site, χρησιμοποιούσαν τη γλωσσα python και για το λόγο αυτό την επιλέξαμε. Υπήρχαν αρκετά tutorials που ήταν πολύ βοηθητικά και μας έδωσαν σαφείς επεξηγήσεις για το πως θα στέλνουμε τα requests. Ωστόσο υπήρξε μία σύγχυση καθώς διαθέτει αρκετούς διαφορετικούς τρόπους (με διαφορετικά performances) με τους οποίους μπορούμε να στείλουμε requests και μετά από αρκετό ψάξιμο καταλήξαμε πως ο πιο efficient τρόπος να στέλνουμε τα requests είναι με τη χρήση των συναρτήσεων execute_read και execute_write και περνώντας σαν arguments τις παραμέτρους των requests δίνοντας έτσι τη δυνατότητα στο Neo4j να cache-αρει τα αποτελέσματα τους σε περίπτωση που μελλοντικά

επαναληφθούν. Κάτι αρκετά ενδιαφέρον για το Neo4j είναι το perfomance tunning που κάναμε στο heap του JVM. Πιο αναλυτικά η default τιμή του ήταν στα 512MB και η max τιμή του στο 1GB. Ακολουθώντας το documentation https://neo4j.com/developer/guide-performance-tuning/#_heap_sizing αλλάζοντας τις τιμές αυτές σε τιμές που προτείνει το παραπάνω link παρατηρήσαμε βελτίωση στα batch requests. Ο πειραματισμος με αυτές τις τιμές ήταν αρκετά ενδιαφέρον και επικοδομητικός! Επιπροσθέτως αυξήσαμε και το μέγεθος του pagecache size προκειμένου να επιτρέπουμε στο Neo4j να cache-αρει ακόμα περισσότερα requests. Βεβαια όλα αυτά είχαν νόημα προφανώς σε μεγάλους γράφους (#κόμβων >= 10000).

## **IV** Περιγραφή των αποτελεσμάτων και των συμπερασμάτων της εργασίας

Η περιγραφή και σύγκριση των αποτελέσματω αρχικά θα γίνει μεταξύ ίδιων μεθόδων πάνω στο ίδιο Graph DMBS (πχ insert vs batch insert) και στη συνέχεια θα γίνει μεταξύ των δύο Graph DBMS (Hugegraph vs Neo4j) καθώς και τα δύο έχουν αρκετό ενδιαφέρον. Κάναμε και barplots για τη σύγκριση των αποτελέσματων, ωστόσο ήταν (υπερβολικά) πολλά και για αυτό δεν τα παρουσιάζουμε στη παρουσίαση αυτή. Σε περίπτωση που επιθυμείτε να τα δείτε τα παρεχουμε στα αρχείο main.ipynb στα branches του repo μας.

Σημείωση: Ο συμβολισμός NA σημαίνει Not Acceptable και σημαίνει πως ο συνολικός χρόνος ήταν υπερβολικά μεγάλος για να γίνει τοπικά στο μηχάνημά μας (για αυτό το βλέπετε μόνο στα insert καθώς έπρεπε να τρέξουν ολόκληρα).

### Hugegraph:

**Hyperparameters:**

- batch_vertex size: 200
- batch_edge_size: 100

Table 1.1. Παρουσίαση χρόνου σε: Insert vertices - Hugegraph

| | Insert | | | Batch Insert | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0035 | 0.0074 | 0.014 | 0.0005 | 0.0005 | 0.0005 |
| 100 | 0.0028 | 0.0046 | 0.011 | 0.0001 | 0.0001 | 0.0001 |
| 1K | 0.0026 | 0.0036 | 0.007 | 7.6e-5 | 0.0001 | 0.0002 |
| 10K | NA | NA | NA | 7.3e-5 | 9.5e-5 | 0.0004 |
| 100K | NA | NA | NA | 6.8e-5 | 9.7e-5 | 0.0008 |
| 1M | NA | NA | NA | 6.7e-5 | 8.7e-5 | 0.0003 |

Table 1.2. Παρουσίαση χρόνου σε: Insert edges - Hugegraph

| | Insert | | | Batch Insert | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0036 | 0.0057 | 0.015 | 0.0009 | 0.0009 | 0.0009 |
| 100 | 0.0031 | 0.0044 | 0.021 | 0.0003 | 0.0004 | 0.0007 |
| 1K | 0.0027 | 0.0036 | 0.006 | 0.0003 | 0.0004 | 0.0012 |
| 10K | NA | NA | NA | 0.0003 | 0.0004 | 0.0037 |
| 100K | NA | NA | NA | 0.0002 | 0.0003 | 0.0026 |
| 1M | NA | NA | NA | 0.0003 | 0.0007 | 0.0057 |

Table 1.3. Παρουσίαση χρόνου σε: Update vertices - Hugegraph

| | Update | | | Update Gremlin | | | Batch Update | | |
|---|---|---|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0063 | 0.0131 | 0.0230 | 0.018 | 0.0211 | 0.0254 | 0.0024 | 0.0024 | 0.0024 |
| 100 | 0.0030 | 0.0038 | 0.0055 | 0.0098 | 0.0129 | 0.0571 | 0.0001 | 0.0001 | 0.0001 |
| 1K | 0.0028 | 0.0037 | 0.0061 | 0.0095 | 0.0128 | 0.0871 | 0.0001 | 0.0001 | 0.0003 |
| 10K | 0.0035 | 0.0041 | 0.0078 | 0.0130 | 0.0170 | 0.0258 | 7.8e-5 | 9.4e-5 | 0.0002 |
| 100K | 0.0047 | 0.0056 | 0.0092 | 0.0163 | 0.0221 | 0.0310 | 8.3e-5 | 0.0001 | 0.0003 |
| 1M | 0.0089 | 0.0042 | 0.0167 | 0.0206 | 0.0270 | 0.0337 | 0.0002 | 0.0003 | 0.0005 |

Table 1.4. Παρουσίαση χρόνου σε: Update edges - Hugegraph

| | Update | | | Update Gremlin | | | Batch Update | | |
|---|---|---|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0057 | 0.0105 | 0.0640 | 0.0164 | 0.0288 | 0.0506 | 0.0025 | 0.0029 | 0.0050 |
| 100 | 0.0038 | 0.0051 | 0.0216 | 0.0103 | 0.0158 | 0.0380 | 0.0004 | 0.0006 | 0.0022 |
| 1K | 0.0035 | 0.0044 | 0.0350 | 0.0086 | 0.0130 | 0.2123 | 0.0004 | 0.0007 | 0.0026 |
| 10K | 0.0040 | 0.0047 | 0.0388 | 0.0090 | 0.0099 | 0.0613 | 0.0004 | 0.0005 | 0.0028 |
| 100K | 0.0051 | 0.0060 | 0.0640 | 0.0104 | 0.0150 | 0.0850 | 0.0002 | 0.0004 | 0.0027 |
| 1M | 0.0072 | 0.0998 | 0.0732 | 0.0173 | 0.0222 | 0.0967 | 0.0010 | 0.0013 | 0.0043 |

Table 1.5. Παρουσίαση χρόνου σε: Read vertices - Hugegraph

| | Read | | | Read Gremlin | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0024 | 0.0041 | 0.0057 | 0.0145 | 0.0171 | 0.0208 |
| 100 | 0.0018 | 0.0025 | 0.0044 | 0.008 | 0.0102 | 0.0179 |
| 1K | 0.0015 | 0.0022 | 0.0190 | 0.0081 | 0.0110 | 0.0926 |
| 10K | 0.0016 | 0.0023 | 0.0042 | -- | -- | -- |
| 100K | 0.0025 | 0.0031 | 0.0112 | -- | -- | -- |
| 1M | 0.0018 | 0.0024 | 0.0041 | -- | -- | -- |

Table 1.6. Παρουσίαση χρόνου σε: Read edges - Hugegraph

| | Read | | | Read Gremlin | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0006 | 0.0008 | 0.0012 | 0.0011 | 0.0015 | 0.0026 |
| 100 | 0.0002 | 0.0003 | 0.0007 | 0.0002 | 0.0004 | 0.0010 |
| 1K | 6.9e-5 | 0.0001 | 0.0008 | 6.9e-5 | 0.0001 | 0.0016 |
| 10K | 6.5e-5 | 0.0001 | 0.0033 | -- | -- | -- |
| 100K | 6.4e-5 | 0.0001 | 0.0033 | -- | -- | -- |
| 1M | 8.6e-5 | 0.0001 | 0.0031 | -- | -- | -- |

Table 1.7. Παρουσίαση χρόνου σε: Delete vertices - Hugegraph

| | Delete | | | Delete Gremlin async | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0054 | 0.0065 | 0.010 | -- | 0.0037 | -- |
| 100 | 0.0034 | 0.0042 | 0.0072 | -- | 0.0007 | -- |
| 1K | 0.0037 | 0.0094 | 0.0558 | -- | 0.0299 | -- |
| 10K | 0.0034 | 0.0088 | 0.0476 | -- | -- | -- |
| 100K | 0.0033 | 0.0082 | 0.1325 | -- | -- | -- |
| 1M | 0.0042 | 0.0149 | 0.2111 | -- | -- | -- |

Table 1.8. Παρουσίαση χρόνου σε: Delete edges - Hugegraph

| | Delete | | | Delete Gremlin async | | |
|---|---|---|---|---|---|---|
| #nodes | min | mean | max | min | mean | max |
| 10 | 0.0048 | 0.0073 | 0.0156 | -- | 0.0005 | -- |
| 100 | 0.0034 | 0.0048 | 0.0509 | -- | 0.1778 | -- |
| 1K | 0.0033 | 0.0043 | 0.0736 | -- | 0.2441 | -- |
| 10K | 0.0033 | 0.0042 | 0.0343 | -- | -- | -- |
| 100K | 0.0033 | 0.0040 | 0.0361 | -- | -- | -- |
| 1M | 0.0034 | 0.0046 | 0.0374 | -- | -- | -- |

Να σημειωθεί ότι το Delete και το Read των κόμβων και των σχέσεων – ακμών στις περιπτώσεις όπου #κόμβων > 10000 έγινε πάνω στους πρώτους 100 κόμβους και τις αντίστοιχες ακμές του γιατί οι χρόνοι ήταν απαγορευτικοί. Όλες οι υπόλοιπες ενέργειες και οι αντίστοιχες μετρήσεις τους έχουν γίνει πάνω σε όλο το γράφο για πιο αντικειμενικά αποτελέσματα!

**Σύγκριση αποτελεσμάτων:**

Τα παραπάνω αποτελέσματα έχουν πολύ ενδιαφέρον. Συγκρίνοντας αρχικά τα κανονικά CRUD operations με τα αντίστοιχα Batched τους, μπορεί να παρατηρήσει κανείς πως τα batched είναι μία τάξη (και παραπαπάνω, όπως στο update) μεγέθους πιο γρήγορα, τόσο στα μικρά όσο και στα μεγάλα datasets. Αυτός είναι και ο λόγος που χρησιμοποιήσαμε τα Batched versions στα μεγάλα datasets (όπως έχουμε προαναφέρει, τα κανονικά CRUD operations στα μεγάλα datasets είχαν απαγορευτικούς συνολικούς χρόνους, μεγαλύτερους της μίας ημέρας, ενώ τα batched μολίς κάποια λεπτά – ώρες). Επίσης αξιοσημείωτο είναι το γεγονός πως κατά μέσο όρο τα read operations είναι και τα πιο γρήγορα, κάτι το οποίο και περιμέναμε. Επιπροσθέτως, μπορεί να παρατηρήσει κανείς πως όλα τα operations χρησιμοποιώντας sync gremlin requests είναι πιο αργά από το απλό HTTP request και αυτό μπορεί να αιτιολογηθεί από τη φωτογραφία που παραθέσαμε παραπάνω όσον αφορά το architecture overview, καθώς με το HTTP request “χτυπάμε” απλά το Hugegraph-Server και στη συνέχεια πάει το request μας απευθείας στο Huge Core ενώ στην άλλη περίπτωση το request μας πηγαίνει στο Hugegraph-Server από εκεί στο Gremlin-Server και στη συνέχεια στο Huge Core. Επιπλέον, ένα από τα πιο ενδιαφέροντα αποτελέσματα που μπορεί κανείς να παρατηρήσει είναι εάν κοιτάξει τον πίνακα των deletes. Πιο αναλυτικά, εάν και το async delete request με gremlin δείχνει στους 10 κόμβους να τα πηγαίνει αρκετά καλύτερα, όσο οι κόμβοι αυξάνονται κάτι τέτοιο δεν ισχύει. Διαβάζοντας async gremlin requests, εμείς θεωρούσαμε αρκετά σίγουρο πως τα ασύγχρονα αυτά requests θα έτρεχαν πολύ πιο γρήγορα, ωστόσο αυτή ήταν μία λανθασμένη θεωρία και υπάρχει αιτολόγηση του γιατί συμβαίνει αυτό. Εάν και το ασύγχρονο σε απελευθερώνει από το overhead του να τελειώσει το request (non blocking), πρώτον περιμένεις για τη δημιουργία του αντίστοιχου task (καθώς αφού δημιουργηθεί το task θα επιστρέψει το request), δεύτερον εν τέλη όλα τα requests σειριοποιούνται εσωτερικά στο gremlin server σε μία ουρά και τρίτον πρέπει να επαναληφθούν HTTP requests στο task API που παρέχει ο Hugegraph-Server (TaskAPI) προκειμένου να είμαστε σίγουροι για την εγκυρότητα των requests (άμα το Gremlin request τελείωνε με επιτυχία το task status μαρκαριζόταν ως success και από εκεί καταλαβαίναμε πως τα requests έχουν εκτελεστεί σωστά), και αυτός ήταν ο κύριος παράγοντας της αναποτελεσματικής απόδοσης, καθώς όταν αφαιρούσαμε τον έλεγχο αυτό τότε προφανώς η διαδικασία τέλειωνε πολύ γρήγορα. Ενδιαφέρον επίσης έχει και η δεύτερη παρατήρηση που αναφέραμε, την οποία βρήκαμε διαβάζοντας το https://tinkerpop.apache.org/docs/current/reference/ το οποίο είναι το official documentation της Apache TinkerPop για το Gremlin. Πιο συγκεκριμένα, σε ένα σημείο αναφέρει:

Figure 2. Σχόλιο για τα ασύχρονα requests με gremlin

- While you may submit parallel asynchronous requests to a session, it may not make sense to do so because they are simply executed serially as they arrive to the session. A failed asynchronous request could leave an invalid state in the session which may not allow later requests to succeed. Either use synchronous requests only or carefully consider error conditions with asynchronous requests.

Το οποίο σε προτρέπει μάλιστα να μη χρησιμοποιείς async requests. Ωστόσο η σημασία των ασύγχρονων request είνα αδιαμφισβήτητη και δεν χρειάζεται να την αναλύσουμε εδώ.

**Neo4j:**

**Hyperparameters:**

- batch_vertex size: 1000
- batch_edge_size: 1000

Table 2.1. Παρουσίαση χρόνου σε: Insert vertices - Neo4j

| | Insert | | | Batch Insert | | |
|---|---|---|---|---|---|---|
| #nodes | min | mean | max | min | mean | max |
| 10 | 0.0222 | 0.0487 | 0.20 | 0.0218 | 0.0218 | 0.0218 |
| 100 | 0.0068 | 0.0158 | 0.193 | 0.0012 | 0.0012 | 0.0012 |
| 1K | 0.0024 | 0.0084 | 0.484 | 0.0001 | 0.0001 | 0.0001 |
| 10K | NA | NA | NA | 6.5e-5 | 0.0001 | 0.0002 |
| 100K | NA | NA | NA | 6.8e-5 | 0.0001 | 0.0012 |
| 1M | NA | NA | NA | 2.8e-5 | 4.0e-5 | 0.0005 |

Table 2.2. Παρουσίαση χρόνου σε: Insert edges - Neo4j

| | Insert | | | Batch Insert | | |
|---|---|---|---|---|---|---|
| #nodes | min | mean | max | min | mean | max |
| 10 | 0.0079 | 0.0164 | 0.122 | 0.0050 | 0.0050 | 0.0050 |
| 100 | 0.0033 | 0.0072 | 0.098 | 0.0003 | 0.0003 | 0.0003 |
| 1K | 0.0011 | 0.0046 | 0.111 | 5.2e-5 | 6.7e-5 | 0.0002 |
| 10K | NA | NA | NA | 5.8e-5 | 0.0001 | 0.0004 |
| 100K | NA | NA | NA | 5.2e-5 | 6.4e-5 | 0.0003 |
| 1M | NA | NA | NA | 2e-5 | 6.7e-5 | 0.0004 |

Table 2.3. Παρουσίαση χρόνου σε: Update vertices - Neo4j

| | Update | | | Batch Update | | |
|---|---|---|---|---|---|---|
| #nodes | min | mean | max | min | mean | max |
| 10 | 0.0110 | 0.0246 | 0.123 | 0.0122 | 0.0122 | 0.0122 |
| 100 | 0.0043 | 0.0084 | 0.010 | 0.0018 | 0.0018 | 0.0018 |
| 1K | 0.0020 | 0.0074 | 0.203 | 0.0006 | 0.0006 | 0.0006 |
| 10K | 0.0027 | 0.0042 | 0.008 | 0.0001 | 0.0001 | 0.0004 |
| 100K | 0.0031 | 0.0046 | 0.008 | 9.8e-5 | 0.0001 | 0.0003 |
| 1M | 0.0069 | 0.0075 | 0.015 | 8.8e-5 | 0.0001 | 0.0005 |

Table 2.4. Παρουσίαση χρόνου σε: Update edges - Neo4j

| | Update | | | Batch Update | | |
|---|---|---|---|---|---|---|
| #nodes | **min** | **mean** | **max** | **min** | **mean** | **max** |
| 10 | 0.0020 | 0.0047 | 0.020 | 0.0062 | 0.0062 | 0.0062 |
| 100 | 0.0004 | 0.0010 | 0.007 | 0.0004 | 0.0004 | 0.0004 |
| 1K | 0.0001 | 0.0002 | 0.004 | 6.6e-5 | 8.5e-5 | 0.0003 |
| 10K | 0.0001 | 0.0007 | 0.008 | 5.7e-5 | 6.8e-5 | 0.0002 |
| 100K | 0.0001 | 0.0005 | 0.004 | 4.8e-5 | 5.9e-5 | 0.0002 |
| 1M | 0.0002 | 0.0006 | 0.005 | 6.2e-5 | 0.0001 | 0.0003 |

Table 2.5. Παρουσίαση χρόνου σε: Read vertices - Neo4j

| | Read | | |
|---|---|---|---|
| #nodes | **min** | **mean** | **max** |
| 10 | 0.00731 | 0.01701 | 0.07453 |
| 100 | 0.00367 | 0.00670 | 0.06378 |
| 1K | 0.00150 | 0.00303 | 0.05189 |
| 10K | 0.00167 | 0.00329 | 0.04671 |
| 100K | 0.00178 | 0.00386 | 0.03822 |
| 1M | 0.00157 | 0.00345 | 0.06605 |

Table 2.6. Παρουσίαση χρόνου σε: Read edges - Neo4j

| | Read | | |
|---|---|---|---|
| #nodes | **min** | **mean** | **max** |
| 10 | 0.00143 | 0.00312 | 0.01203 |
| 100 | 0.00035 | 0.00070 | 0.00635 |
| 1K | 8.8e-5 | 0.00013 | 0.00196 |
| 10K | 0.00010 | 0.00011 | 0.00371 |
| 100K | 8.9e-5 | 0.00016 | 0.00257 |
| 1M | 8.6e-5 | 0.00019 | 0.01641 |

Table 2.7. Παρουσίαση χρόνου σε: Delete vertices - Neo4j

| | Delete | | |
|---|---|---|---|
| #nodes | **min** | **mean** | **max** |
| 10 | 0.01024 | 0.01823 | 0.05832 |
| 100 | 0.00413 | 0.00809 | 0.04909 |
| 1K | 0.00239 | 0.00529 | 0.07015 |
| 10K | 0.00301 | 0.00503 | 0.04264 |
| 100K | 0.00389 | 0.00602 | 0.05784 |
| 1M | 0.00266 | 0.00526 | 0.06499 |

Table 2.8. Παρουσίαση χρόνου σε: Insert vertices - Neo4j

| | Delete | | |
|---|---|---|---|
| #nodes | **min** | **mean** | **max** |
| 10 | 0.00328 | 0.00788 | 0.01551 |
| 100 | 0.00069 | 0.00205 | 0.01131 |
| 1K | 0.00011 | 0.00029 | 0.01312 |
| 10K | 0.00010 | 0.00022 | 0.01221 |
| 100K | 0.00010 | 0.00020 | 0.00549 |
| 1M | 0.00010 | 0.00024 | 0.02509 |

Να σημειωθεί ότι το Delete και το Read των κόμβων και των σχέσεων – ακμών στις περιπτώσεις όπου #κόμβων > 10000 έγινε πάνω στους πρώτους 100 κόμβους και τις αντίστοιχες ακμές του γιατί οι χρόνοι ήταν απαγορευτικοί. Όλες οι υπόλοιπες ενέργειες και οι αντίστοιχες μετρήσεις τους έχουν γίνει πάνω σε όλο το γράφο για πιο αντικειμενικά αποτελέσματα!

**Σύγκριση αποτελεσμάτων:**

Αρχικά, πρέπει να αναφέρουμε ότι επιλέξαμε αυτό το batch size (1000 τόσο σε vertices όσο και edges) καθώς με μεγαλύτερα batch sizes το performance των batched requests άρχισε να χειροτερεύει. Αυτό πιστεύουμε πως οφείλεται στο γεγονός ότι δεν έφτανε (στα μεγαλύτερα batch sizes) το Heap Memory που δίναμε στον JVM.

Συγκρίνοντας και πάλι τα απλά CRUD operations σε σχέση με τα αντίστοιχα batched versions τους παρατηρούμε ότι σε πλήθος κόμβων 10 και 100 δεν πετυχαίνουμε πολύ καλύτερο perfomance (speedup 1-2.5), ωστόσο παρατηρούμε ότι στους 1000 κόμβους το speedup εκτοξέυεται στο 8 (πέρα από το update edges που είναι 2.5) και βλέποντας το χρόνο που απαιτούσε πχ για να κάνει insert τα δεδομένα με απλά CRUD operations (10-12 ώρες) σε σχέση με τα αντίστοιχα batched (40-45 λεπτά) σε περισσότερους κόμβους (1M) το speedup εκεί θα είναι αρκετά μεγαλύτερο! Άμα είχαμε ακόμα πιο δυνατό μηχάνημα και μπορούσαμε να δώσουμε ακόμα περισσότερο Heap Memory στον JVM για να έχουμε ακόμα μεγαλύτερα batches τότε είμαστε σίγουροι πως το speedup θα ήταν αρκετά μεγαλύτερο. Μάλιστα σε τάξεις των εκατομμυρίων δεδομένων διαβάσαμε πως το προτεινόμενο batch size είναι της τάξης κάποιων δεκάδων χιλιάδων (πχ 30000). Τέλος αξιοσημείωτο είναι το γεγονός ότι τόσο τα delete και read operations είναι ανθεκτικά ως προς το μέγεθος των δεδομένων αναδεικνύοντας έτσι το scalability του Neo4j, εάν και αυτό θα συζητηθεί περαιτέρω αργότερα, όταν συγκρίνουμε το Hugegraph με το Neo4j.

## Neo4j vs Hugegraph:

Καθώς η εξαγωγή των αποτελεσμάτων είναι δύσκολη μέσω των παραπάνω πινάκων, ακολουθεί σύγκριση σε διαγράμματα, με κάθε βασικό CRUD operation insert, read, update, delete. Επειδή για κάθε διαφορετική βάση παράγονται 8 διαγράμματα, επιλέγουμε να παρουσιάσουμε στην αναφορά μόνο για τις βάσεις node_100 και node_1000000. Στο αρχείο comparisons.ipynb φαίνονται όλα τα επι μέρους αποτελέσματα.

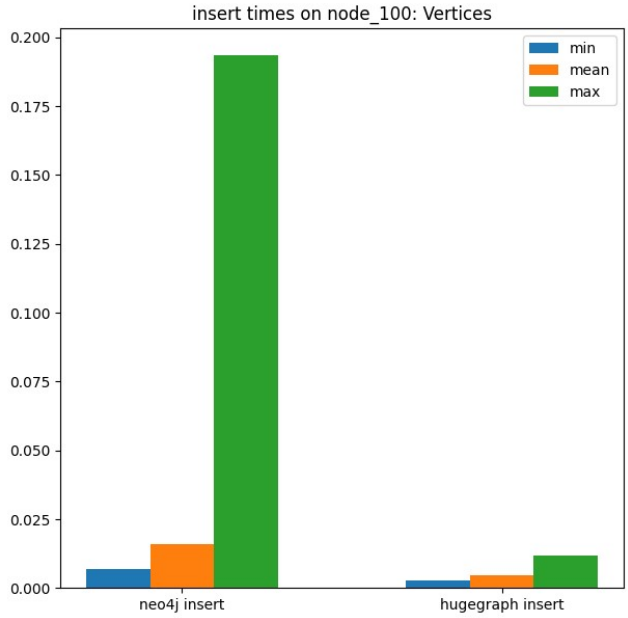


Figure 3.1: Συγκριση σε Insert operations στα 100 στοιχεία για vertices

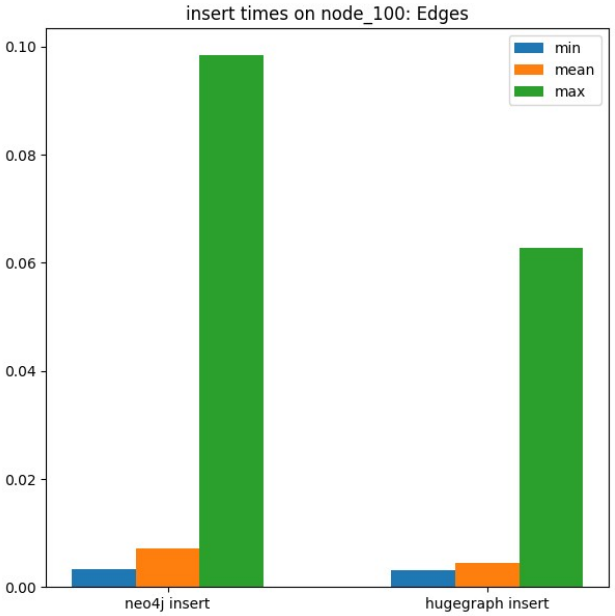


Figure 3.2: Σύγκριση σε Insert operations στα 100 στοιχεία για edges

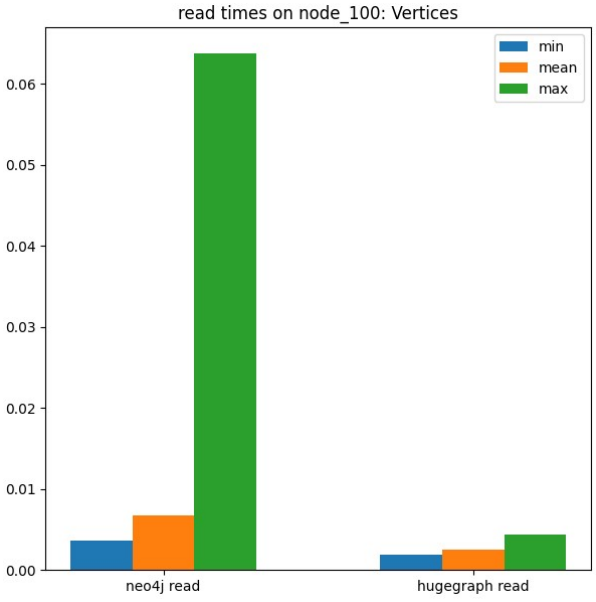


Figure 3.3: Σύγκριση σε Read operations στα 100 στοιχεία για vertices

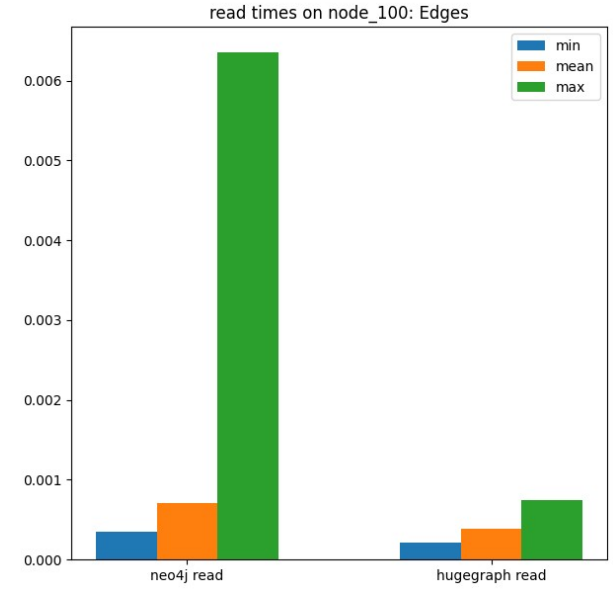


Figure 3.4: Σύγκριση σε Read operations για 100 στοιχεία για edges

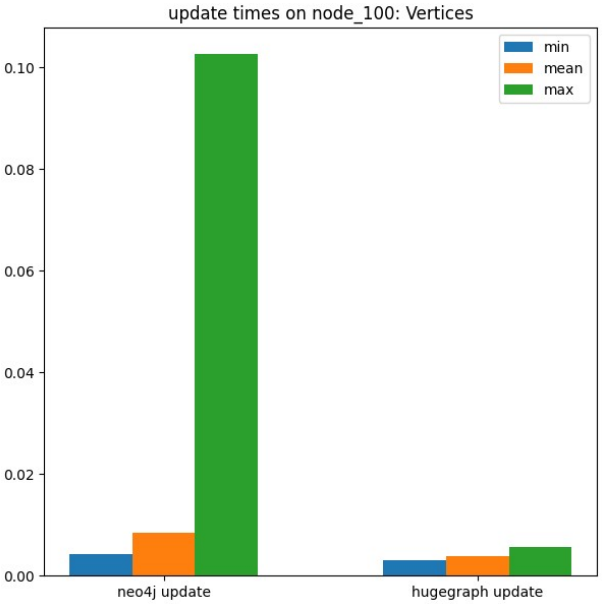


Figure 3.5: Σύγκριση σε Update operations στα 100 στοιχεία για vertices

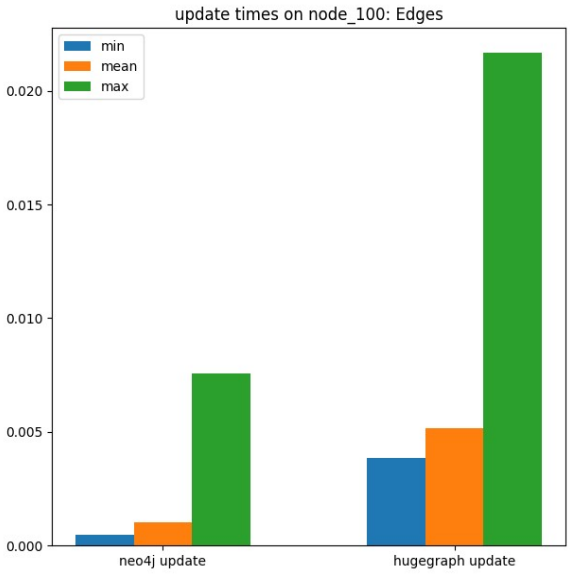


Figure 3.6: Σύγκριση σε Update operations στα 100 στοιχεία για edges

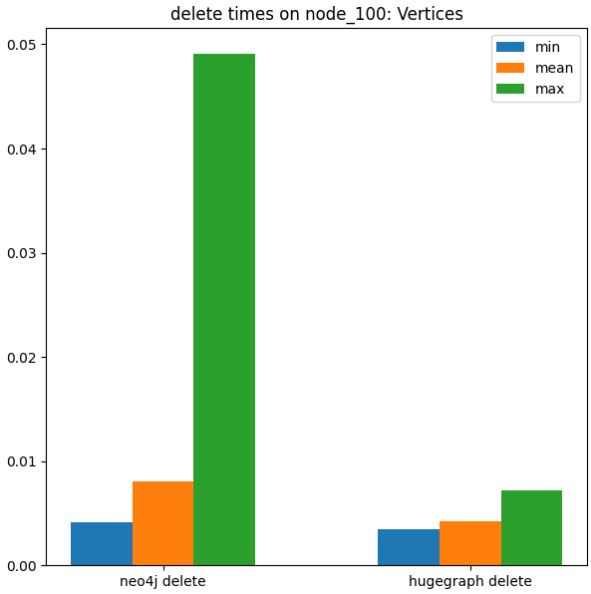


Figure 3.7: Σύγκριση σε Delete operations στα 100 στοιχεία για vertices

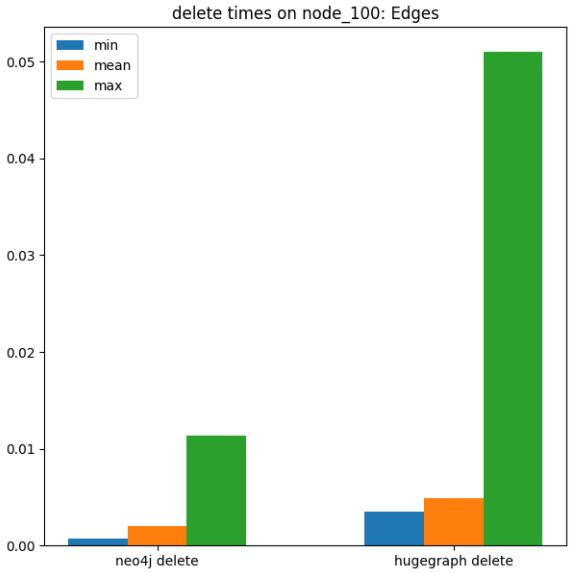


Figure 3.8: Σύγκριση σε Delete operations στα 100 στοιχεία για edges

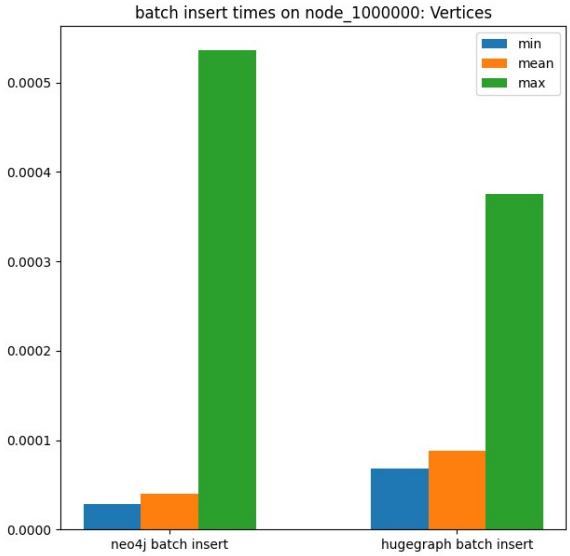


Figure 4.1: Σύγκριση σε batch insert operations στα 1M στοιχεία για vertices

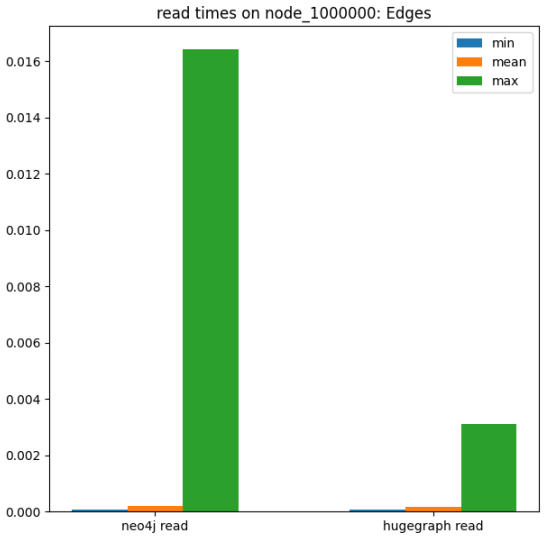


Figure 4.2: Σύγκριση σε batch insert operations στα 1M στοιχεία για edges

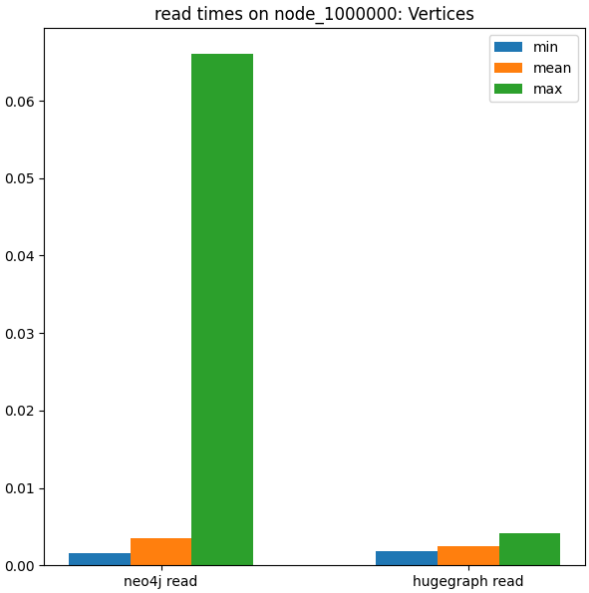


Figure 4.3: Σύγκριση σε read operations στα 1M στοιχεία για vertices

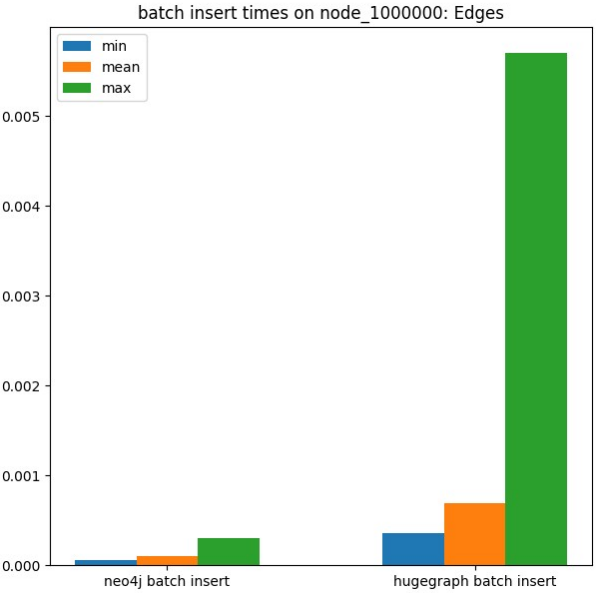


Figure 4.4: Σύγκριση σε read operations στα 1M στοιχεία για edges

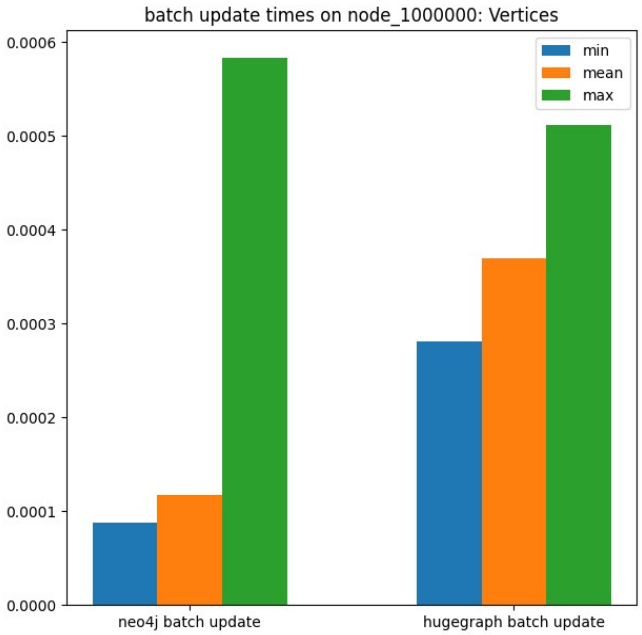


Figure 4.5: Σύγκριση σε batch update operations στα 1M στοιχεία για vertices

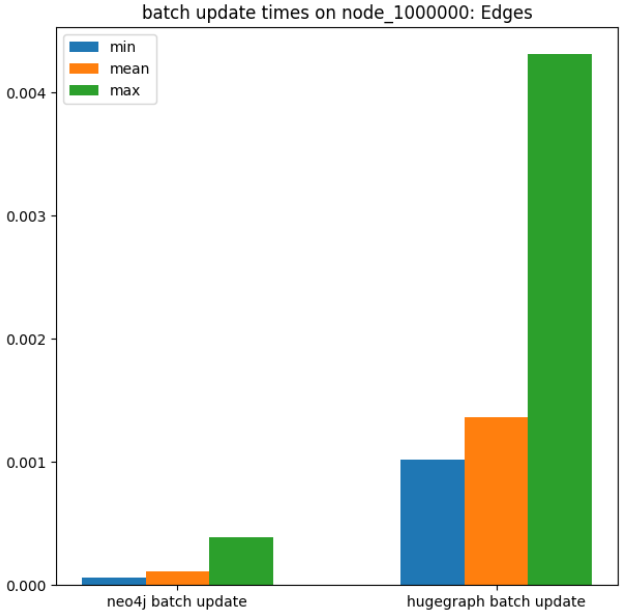


Figure 4.6: Σύγκριση σε batch update operations στα 1M στοιχεία για edges

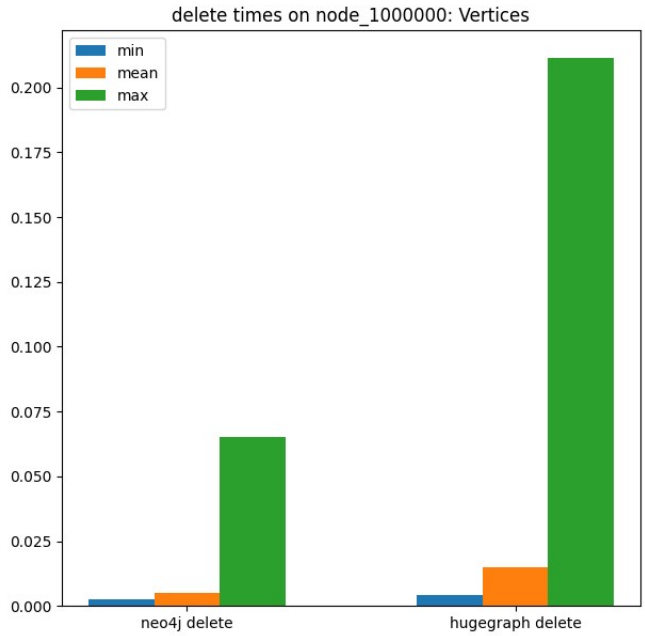


Figure 4.7: Σύγκριση σε delete operations στα 1M στοιχεία για vertices

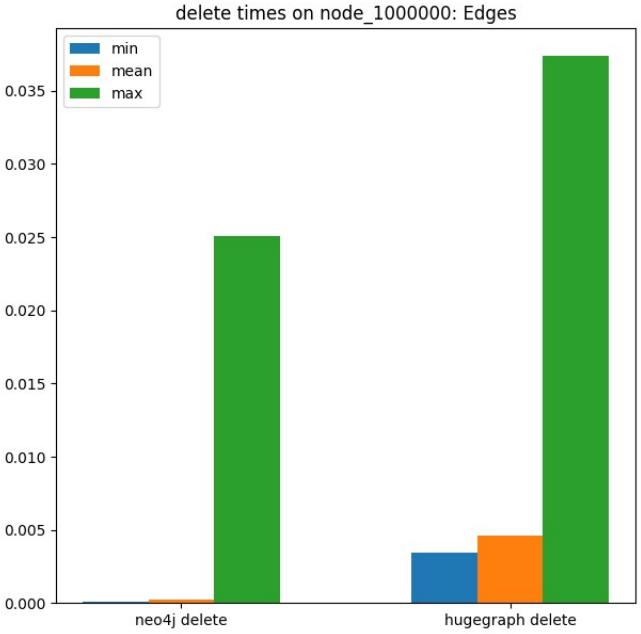


Figure 4.8: Σύγκριση σε delete operations στα 1M για edges

Αρχικά παρατηρούμε πως το Hugegraph είναι scalable καθώς οι χρόνοι παραμένουν πολύ παρόμοιοι με την άυξηση των κόμβων – dataset. Δηλαδή στα inserts, updates, read και updates οι χρόνοι είναι στη ίδια τάξη μεγέθους με μικρές διακυμάνσεις, πέρα από το απλό delete των vertices και των update των edges (τόσο το απλό όσο και με Gremlin) όπου εκεί παρατηρούμε αύξηση των μέσων τιμών εκτέλεσης με την αύξηση των κόμβων. Από την άλλη το Neo4j είναι scalable σε όλα τα operations, καθώς όλοι οι χρόνοι είναι στην ίδια τάξη μεγέθους ανεξάρτητα από το μέγεθος των κόμβων (στους 10 κόμβους μόνο είναι αργό, αρκετά πιο αργό από το Hugegraph, αλλά αυτό πιστεύουμε πως έχει μικρή σημασία). Όσον αφορά τη σύγκριση των δύο Graph DBMSs το Neo4j γενικά πετυχαίνει αρκετά πιο γρήγορα απλά CRUD operations (όσον αφορά τις μέσες τιμές τους), της τάξης του 2-3 speedup στην γενική περίπτωση (αλλά και παραπάνω σε μερικές περιπτώσεις όπου και πετυχαίνει speedup της τάξης του 10). Ωστόσο η πιο σημαντική παρατήρηση είναι ότι τα batched CRUD operations του Neo4j πετυχαίνουν μία τάξης μεγέθους μικρότερους χρόνους (στα edges καθώς αυτά χρειάζονται το περισσότερο χρόνο, στα μεγάλα datasets) σε σχέση με αυτού του Hugegraph, καθώς με ένα πχ batched insert request κάνουν insert 1000 κόμβους και 1000 edges σε σύγκριση με αυτού το Hugegraph που κάνει insert 200 και 100 αντίστοιχα. Για το λόγο αυτό το Neo4j τελειώνει πχ στους 1M κόμβους το insertion σε λεπτά (45 λεπτά για την ακρίβεια) ενώ το Hugegraph σε ώρες (4 ώρες και 45 λεπτά για την ακρίβεια). Από αυτό και μόνο το αποτέλεσμα φαίνεται η ανωτερότητα του Neo4j (να υπενθυμίσουμε ότι το Neo4j είναι rank 1, με score 53.51, στα Graph DBMSs ενώ το hugegraph rank 22, με score 0.46 που είναι μία αρκετά μεγάλη διαφορά). Με βάση τις παραπάνω παρατηρήσεις γενικά θα επιλέγαμε ως Graph DBMS το Neo4j, ωστόσο αυτή η επιλογή έχει και μερικά μειοντεκτήματα. Πρώτον το Neo4j δεν επιτρέπει το να έχουμε πολλαπλά instances up and running ταυτόχρονα, κατί που από μόνο του είναι ένας σημαντικός περιορισμός και κατά δεύτερον αντιμετωπίσαμε αρκετά προβλήματα στην εγκατάσταση μερικών plug ins (όπως το Gremlin plug in) και για αυτό το λόγο δεν τα χρησιμοποιήσαμε. Όσον αφορά το ease of use, γενικά το Neo4j ήταν αρκετά πιο απλό τόσο στην εγκατάσταση όσο και στη χρήση σε σχέση με το Hugegraph (στο οποίο εάν δεν είχαμε τον έτοιμο PyHugegraph Driver θα έπρεπε να κάνουμε πολύ περισσότερο develop). Επιπροσθέτως όσον αφορά τα Languages που χρησιμοποιήσαμε, τόσο το Gemlin (Hugegraph) όσο και το Cypher (Neo4j) είχαν αρκετό ενδιαφέρον και ήταν και τα δύο αρκετά εύκολα στη χρήση τους.

**Neo4j vs Hugegraph traversers:**

Τέλος, όσον αφορά τους traversers στο Hugegraph πειραματιστήκαμε, όπως έχουμε προαναφέρει, με τα K-out, K-neighbor και shortest path ενώ με το Neo4j μόνο με το shortest path καθώς για τα άλλα δύο χρειαζόμασταν κάποια plug ins τα οποία δεν καταφέραμε να κατεβάσουμε. Πιο αναλυτικά:

| | **Hugegraph** | | | **Neo4j** |
|---|---|---|---|---|
| #nodes | K-out | K-neighbor | Shortest Path | Shortest Path |
| 10 | 0.005 | 0.0045 | 0.0573 | 0.0138 |
| 100 | 0.004 | 0.0037 | 0.0836 | 0.3244 |
| 1K | 4.065 | 4.3190 | 0.1576 | 0.3663 |
| 10K | 3.972 | 4.0430 | 0.2985 | 0.2305 |
| 100K | 9.482 | 9.7999 | 0.8544 | 0.4100 |
| 1M | 12.71 | 13.448 | 1.8818 | 0.2019 |

Να σημειωθεί ότι δεν χρησιμοποιήσαμε κάποιο property των edges πχ στο shortest path (δεν χρησιμοποιήσαμε για παράδειγμα το βάρος των ακμών, που στη περίπτωση μας είναι το property years των ακμών – σχέσεων). Επίσης το shortest path μεταξύ Neo4j και Hugegraph έγινε πάνω στους ίδιους κόμβους προφανώς.

Από τα παραπάνω παρατηρούμε ότι στο Hugegraph όσο μεγαλώνει το μέγεθος του dataset τόσο περισσότερο χρόνο απαιτεί και η ολοκήρωσει του κάθε query. Πιο αναλυτικά η μετάβαση από τους 100 κόμβους στους 1000 εκτινάσει το χρόνο εκτέλεσης από κάποια milisecond σε second (3 τάξεις μεγέθους μεγαλύτερο) τόσο στο K-out όσο και στο K-neighbor. Επίσης οι χρόνοι μεταξύ K-out και K-neighbor παρατηρούμε πως είναι σχεδον οι ίδιοι και αυτό είναι απολύτως λογικό εάν το σκεφτεί κανείς καθώς τα δύο αυτά queries εν τέλη εκφυλίζονται στο ίδιο, το μόνο που αλλάζει είναι το μέγεθος του response (το response του K-neighbor είναι superset του response του K-out). Όσον αφορά το Shortest Path στο Hugegraph παρατηρούμε και πάλι ότι όσο αυξάνονται οι κόμβοι τόσο αυξάνεται και ο χρόνος εκτέλεσης του query. Πιο αναλυτικά στους 10 κόμβους απαιτεί κάποιες δεκάδες milisecond (50 milisecond για την ακρίβεια), στους ενδιάμεσους κόμβους αυξάνεται σταδιακά ο χρόνος και τέλος στους 1M κόμβους απαιτεί σχεδόν 2 δευτερόλεπτα (1.88 για την ακρίβεια). Αυτή η συμπεριφορά ήταν και η αναμενόμενη. Αντιθέτως ωστόσο, προς μεγάλη μας έκπληξη παρατηρούμε ότι στο Neo4j ο χρόνος που απαιτείται για το Shortest Path είναι της ίδιας τάξης μεγέθους σε κάθε πλήθος κόμβων (κάποια δεκάδες milisecond). Αυτό αναδυκνύει για ακόμα μια φορά το γιατί το Neo4j είναι το rank 1 Graph DMBS. Για την ακρίβεια το tool που δίνει τη δυνατότητα να είναι τόσο γρήγορο το Neo4j στο Shortest Path είναι το Graph Data Science (GDS) το οποίο τρέχει με optimized τρόπο traversal queries.

**Βιβλιογραφία:**


**Hugegraph:**

**https://hugegraph.apache.org/docs/introduction/readme/**

**https://hugegraph.apache.org/docs/quickstart/hugegraph-server/**

**https://hugegraph.apache.org/docs/clients/restful-api/**

**https://hugegraph.apache.org/docs/config/config-guide/**

**https://hugegraph.apache.org/docs/guides/architectural/**

**https://github.com/tanglion/PyHugeGraph**

**https://hugegraph.apache.org/docs/language/hugegraph-gremlin/**

**https://tinkerpop.apache.org/docs/current/reference/**

**https://hugegraph.apache.org/docs/performance/api-preformance/hugegraph-api-0.5.6-cassandra/**

**https://hugegraph.apache.org/docs/config/config-https/**

**Neo4j:**

**https://neo4j.com/**

**https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/**

**https://neo4j.com/docs/cypher-manual/current/indexes-for-full-text-search/**

**https://neo4j.com/docs/cypher-manual/current/introduction/**

https://neo4j.com/docs/cypher-manual/current/constraints/

https://neo4j.com/developer/python/

https://neo4j.com/docs/cypher-manual/current/introduction/transactions/

https://neo4j.com/docs/cypher-manual/current/syntax/

https://neo4j.com/docs/python-manual/current/