Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Optimizing SQL-based unbounded regular path queries in a relational database

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΚΟΣΜΑΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Optimizing SQL-based unbounded regular path queries in a relational database

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΚΟΣΜΑΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Μαρτίου 2024.

Δημήτριος Τσουμάκος	Νεκτάριος Κοζύρης	Γεώργιος Γκούμας
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2024

Γεώργιος Κοσμάς

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η επεξεργασία γραφημάτων μέσα σε σχεσιακές βάσεις δεδομένων κερδίζει ολοένα και περισσότερο προσοχή, καθώς οι επιστήμονες δεδομένων προσπαθούν να ανακτήσουν πληροφορίες από γραφήματα που είναι αποθηκευμένα σε μορφή πίνακα μέσα σε RDBMS. Μια ιδιαίτερα απαιτητική κατηγορία ερωτημάτων που μπορεί να εφαρμοστεί σε πολλές περιπτώσεις, όπως για παράδειγμα στην ανίχνευση νομιμοποίησης εσόδων από παράνομες δραστηριότητες, είναι τα ερωτήματα κανονικής διαδρομής χωρίς περιορισμούς (RPQ). Ωστόσο, παρά τη σημασία των απεριόριστων RPQ, οι υπάρχουσες τεχνικές δεν καταφέρνουν να παρέχουν επαρκή απόδοση, κυρίως λόγω έλλειψης ενοποίησης με το εσωτερικό της βάσης δεδομένων. Για την αντιμετώπιση αυτού του προβλήματος, αναπτύχθηκε ένας νέος εξειδικευμένος πυρήνας που είναι ενσωματωμένος σε μια σχεσιακή βάση δεδομένων. Ο πυρήνας αυτός λειτουργεί με την έκδοση SQL statements κατά τη διάρκεια του χρόνου εκτέλεσης του. Αυτά τα SQL statements είναι κρίσιμα για την απόδοσή του, επομένως είναι επιτακτική ανάγκη να τα χειριστούμε αποτελεσματικά. Σε αυτή τη διπλωματική προτείνουμε και εφαρμόζουμε διάφορες τεχνικές για τη βελτιστοποίηση της εκτέλεσης αυτού του πυρήνα. Οι βελτιστοποιήσεις μας αποδίδουν αξιοσημείωτα αποτελέσματα, επιτυγχάνοντας έως και 67 φορές βελτίωση στην απόδοση από άκρο σε άκρο σε σύγκριση με την αρχική έκδοση του πυρήνα.

Λέξεις κλειδιά

βάσεις δεδομένων γράφων, επερωτήματα γράφων, κανονικά ερωτήματα διαδρομής, ανάλυση γράφων

Abstract

Graph processing within relational databases is increasingly gaining attention, as data scientists seek to retrieve information from graphs that are stored in tabular format inside RDBMSs. One particularly challenging class of graph queries that is applicable to a number of scenarios, such as money laundering detection, is unbounded regular path queries (RPQs). However, despite the importance of unbounded RPQs, existing techniques fail to provide sufficient performance, mainly for a lack of integration with the internals of the database engine. To address this problem, a new specialized kernel that is embedded in a relational database was developed. The kernel operates by issuing internal SQL statements during its runtime. This SQL statements are critical for its performance, it is thus imperative to handle them efficiently. We propose and implement several techniques to optimize the SQL-based execution of the kernel. Our optimizations yield remarkable results, achieving up to a 67-fold improvement in end-to-end performance compared to the original version of the kernel.

Key words

graph databases, graph queries, regular path queries, graph analytics

Ευχαριστίες

Η εργασία αυτή σηματοδοτεί και το τέλος ενός σημαντικού κεφαλαίου της ακαδημαϊκής και προσωπικής μου πορείας. Αναλογιζόμενος την εώς τώρα διαδρομή μου, αισθάνομαι την ανάγκη να ευχαριστήσω πρώτα όλους όσοι πίστεψαν σε εμένα και αφιέρωσαν το χρόνο και την ενέργεια τους για να με διδάξουν.

Για την εκπόνηση αυτής της εργασίας, πολλά ευχαριστώ οφείλω στον Hugo Kapp, ο οποίος ήταν δίπλα μου από την πρώτη μέχρι και την τελευταία ημέρα. Τον ευχαριστώ θερμά για την καθοδήγηση και την υποστήριξη του. Ακόμα, ευχαριστώ τους Vlad Ioan Haprian και Martin Brugnara για τα σχόλια τους σχετικά με το τελικό κείμενο. Ευχαριστώ, επίσης, τα μέλη των ομάδων PGX και Graph-in-DB που με φιλοξένησαν σε ένα ευχάριστο περιβάλλον εργασίας.

Επιπλέον, θα ήθελα να ευχαριστήσω τον κ. Δημήτριο Τσουμάκο, τα μαθήματα του οποίου μου έδωσαν τόσο το κίνητρο όσο και τα εφόδια για να ασχοληθώ με την περιοχή της διαχείρισης μεγάλων δεδομένων. Τον ευχαριστώ ιδιαιτέρως που δέχτηκε να συνεπιβλέψει αυτήν τη διπλωματική εργασία.

Κλείνοντας, ευχαριστώ τους φίλους μου για τη συμπαράσταση και τις συμβουλές τους καθ'όλη τη διάρκεια των σπουδών μου και την οικογένεια μου για τη συνεχή στήριξη, την ατελείωτη υπομονή και την αγάπη της.

Γεώργιος Κοσμάς,

Αθήνα, 26η Μαρτίου 2024

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 11

Κατάλογος πινάκων 15

Κατάλογος σχημάτων 17

0. Εκτενής Περίληψη 19
- 0.1 Ανάλυση γράφων σε σχεσιακές βάσεις δεδομένων 19
- 0.2 Τύποι υπολογιστικών φόρτων στην ανάλυση γραφημάτων 21
- 0.3 Γλώσσες ερωτημάτων 22
- 0.4 Μη-φραγμένα κανονικά ερωτήματα μονοπατιών 22
- 0.5 Υπάρχουσες προσεγγίσεις 23
- 0.6 Αλγόριθμος υπολογισμού μη-φραγμένων κανονικών ερωτημάτων μονοπατιών . . . 24
  - 0.6.1 Μοντέλο δεδομενων 24
  - 0.6.2 Περιγραφή αλγορίθμου 25
  - 0.6.3 Εκτέλεση αλγορίθμου 26
- 0.7 Συνεισφορά 27

Κείμενο στα αγγλικά 31

1. Introduction 31
- 1.1 Context 31
- 1.2 Motivation 32
- 1.3 Contribution 33
- 1.4 Thesis outline 33

2. Related Work 35
- 2.1 Relational databases 35
- 2.2 Graph databases 36
- 2.3 Graph processing frameworks 36

2.4 Graph data models . . . 38
2.4.1 Property graphs . . . 38
2.4.2 RDF graphs . . . 38
2.5 Graph workloads . . . 38
2.6 Graph query languages . . . 38
2.6.1 SPARQL . . . 39
2.6.2 Cypher . . . 39
2.6.3 PGQL . . . 39
2.6.4 SQL/PGQ . . . 39
2.6.5 GQL . . . 40
2.7 Regular path queries . . . 40
2.8 Performing graph analytics inside a relational database . . . 40

3. Background . . . 43
3.1 Data model . . . 43
3.2 Execution of graph algorithms in a relational database . . . 44
3.3 Evaluation of unbounded recursive path queries . . . 44
3.3.1 Design overview . . . 44
3.3.2 Algorithm . . . 45
3.3.3 Neighbor expansion . . . 47
3.3.4 Output . . . 47
3.3.5 Heterogeneous graphs . . . 48
3.3.6 Path storage . . . 50

4. Optimizations . . . 51
4.1 Batching for path insertion . . . 51
4.2 Optimizer hints . . . 53
4.3 Prepared statements . . . 54
4.4 Batching for neighbor fetching . . . 55

5. Experiments . . . 57
5.1 Setup . . . 57
5.1.1 Hardware . . . 57
5.1.2 Datasets . . . 57
5.1.3 Queries . . . 58
5.2 Impact of each optimization . . . 58
5.3 Impact of maximum path length . . . 59
5.4 Performance comparison . . . 60
5.5 Insertion batch size . . . 63

6. Future Directions . . . 69
6.1 Skip side table inserts . . . 69
6.2 Many-to-many query optimization . . . 69
6.3 Multi-source breadth-first search . . . 70

6.4 Batching for neighbor expansion 70

7. Summary 71

Bibliography 73

Κατάλογος πινάκων

Πίνακες στο αγγλικό κείμενο

5.1 Dataset sizes 57
5.2 Compound effect of optimizations 59
5.3 Maximum path length experiment (normalized results) 59

Κατάλογος σχημάτων

0.1 Γράφημα που αναπαριστά σχέσεις μεταξύ ανθρώπων . . . 19
0.2 Φραγμένα και μη-φραγμένα κανονικά ερωτήματα μονοπατίων [43] . . . 23
0.3 Παράδειγμα ενος schema μιας τριάδας πηγή-ακμή-προορισμός . . . 25

Σχήματα στο αγγλικό κείμενο

1.1 Unbounded recursive path queries illustration in the context of fraud detection [43] . 32

2.1 Architecture of Grail [16] . . . 42

3.1 Example schema of a source-edge-destination triplet . . . 43
3.2 Unbounded RPQs are splitted in three parts: prefix, suffix and recursive part . . . 45
3.3 Execution flow of the unbounded RPQ evaluation . . . 45
3.4 Kernel receives a vertex s and returns paths to every vertex that is reachable by s. . 46
3.5 Template for finding the neighbors of a vertex . . . 47
3.6 Neighbour expansion for a given vertex . . . 48
3.7 Output of unbounded RPQ . . . 48
3.8 Heterogeneous graph . . . 49
3.9 Multiple joins for neighbor retrieval in heterogeneous graphs . . . 49
3.10 Path sharing versus path copy . . . 50

4.1 Multiple separate inserts versus inserting in batches . . . 52
4.2 Illustration of the ”batching for path insertions” optimization . . . 53
4.3 Design of the buffer used for the batch insertions . . . 53
4.4 Illustration of the ”batching for path insertions” optimization . . . 56

5.1 Queries used for experimental evaluation . . . 58
5.2 Optimizations impact . . . 59
5.3 Maximum path length experiment . . . 60
5.4 Performance comparison - tree subgraph of LDBC . . . 61
5.5 Performance comparison - social subgraph of LDBC . . . 62
5.6 Insertion batch size memory-performance trade-off . . . 63
5.7 Paths per insertion batch for buffer memory limit = 1KB . . . 65
5.8 Paths per insertion batch for buffer memory limit = 10KB . . . 65
5.9 Paths per insertion batch for buffer memory limit = 100K . . . 66
5.10 Paths per insertion batch for buffer memory limit = 1MB . . . 66

5.11 Paths per insertion batch for buffer memory limit = 10MB 67

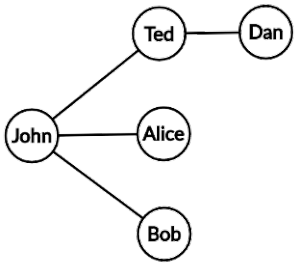
6.1 Example for multi-source BFS 69

Κεφάλαιο 0

Εκτενής Περίληψη

0.1 Ανάλυση γράφων σε σχεσιακές βάσεις δεδομένων

Ένα γράφημα είναι μια δομή δεδομένων που αναπαριστά σχέσεις μεταξύ οντοτήτων. Λόγω της ικανότητάς τους να αναπαριστούν αποτελεσματικά συνδέσεις, τα γραφήματα χρησιμοποιούνται σε ένα ευρύ φάσμα εφαρμογών. Για παράδειγμα, κάποια πεδία στα οποία τα γραφήματα βρίσκουν εφαρμογή είναι τα συστήματα συστάσεων, οι μηχανές αναζήτησης και τα μέσα κοινωνικής δικτύωσης.



Σχήμα 0.1: Γράφημα που αναπαριστά σχέσεις μεταξύ ανθρώπων

Η δημοτικότητα των γραφημάτων οδηγεί σε αύξηση της ζήτησης για λογισμικό που επιτρέπει την αποτελεσματική εξαγωγή συμπερασμάτων από γραφήματα. Οι λύσεις λογισμικού που προσφέρουν δυνατότητες ανάλυσης γραφημάτων πρέπει να είναι σε θέση να επεξεργάζονται τεράστιες ποσότητες δεδομένων, καθώς συχνά καλούνται να ανακαλύψουν μοτίβα που είναι κρυμμένα εντός γραφημάτων τεράστιου μεγέθους. Τέτοια γραφήματα εμφανίζονται σε τομείς όπως το λιανικό εμπόριο, τα οικονομικά και τα μέσα κοινωνικής δικτύωσης.

Από όταν η ανάλυση γραφημάτων πρωτοεμφανίστηκε ως μια πολλά υποσχόμενη τάση στην περιοχή της επιστήμης δεδομένων, αναπτύχθηκαν πολλά εξειδικευμένα συστήματα που έδιναν έμφαση στην αποτελεσματική εκτέλεση εργασιών επεξεργασίας γραφημάτων. Το γεγονός ότι τα υπολογιστικά φορτία που σχετίζονται με την ανάλυση γραφημάτων περιλαμβάνουν επαναληπτικούς υπολογισμούς, σε συνδυασμό με το ότι η γλώσσα SQL δεν ενδείκνυται για να εκφραστούν μέσω αυτής αλγόριθμοι για γραφήματα, οδήγησε στην πεποίθηση ότι η ανάλυση γραφημάτων έχει σημαντικές διαφορές από τα παραδοσιακά OLAP φορτία [24]. Για αυτό το λόγο, η χρήση παραδοσιακών σχεσιακών βάσεων δεδομένων για την εκτέλεση εργασιών επεξεργασίας γραφημάτων δεν έλαβε ιδιαίτερη προσοχή [16]. Ωστόσο, επειδή οι σχεσιακές βάσεις είναι ο ακρογωνιαίος λίθος των συστημάτων διαχείρισης δεδομένων, θα ήταν ιδιαίτερα ωφέλιμο αν οι υπάρχουσες σχεσιακές βάσεις μπορούσαν να αξιοποιηθούν για να φέρουν εις πέρας εργασίες που αφορούν την ανάλυση γράφημάτων, καθώς είναι σύνηθες γραφήματα μεγάλης κλίμακας να βρίσκονται αποθηκευμένα ως πίνακες μέσα σε ένα RDBMS.

Αναμφίβολα, μια σχεσιακή βάση δεδομένων που είναι σε θέση να εξυπηρετεί υπολογιστικά φορτία που σχετίζονται με την ανάλυση γράφημάτων θα ήταν ιδιαίτερα πρακτική, καθώς σήμερα οι περισσότερες επιχειρήσεις έχουν κάποιο RDBMS ως κέντρο της υποδομής τους. Σε πολλές περιπτώσεις η εγκατάσταση ενός δεύτερου εξειδικευμένου συστήματος απαιτεί την μετατροπή δεδομένων από μορφή πίνακα σε μορφή γραφήματος και τη φόρτωσή τους στο καινούργιο σύστημα [24]. Αυτή η εργασία συχνά είναι χρονοβόρα και επιρρεπής σε σφάλματα. Επιπρόσθετα, προκαλεί αύξηση του κόστους αποθήκευσης, καθώς τα αρχικά δεδομένα αποθηκεύονται πλέον δύο φορές. Επιπλέον, επειδή τα δεδομένα αποθηκεύονται σε δύο διαφορετικά συστήματα, ο οργανισμός πρέπει να εγκαθιδρύσει διαδικασίες ETL για να συγχρονίζει το εξειδικευμένο σύστημα με τις αλλαγές που πραγματοποιούνται στο RDBMS [28]. Αυτό είναι ένα τεράστιο μειονέκτημα για μια επιχείρηση, καθώς αυξάνει σημαντικά την πολυπλοκότητα των πληροφοριακών της συστημάτων [16]. Επιπλέον, οι περισσότεροι χρήστες έχουν συνηθίσει να χειρίζονται τα δεδομένα τους μέσω ενός RDBMS. Η εγκατάσταση ενός εξειδικευμένου συστήματος τους εξαναγκάζει να χρησιμοποιήσουν μια πλατφόρμα με την οποία δεν είναι εξοικειωμένοι. Τέλος, πολλές φορές η εγκατάσταση ενός εξειδικευμένου συστήματος δεν επιτρέπεται λόγω νομικών περιορισμών. Για παράδειγμα, όταν καλούμαστε να χειριστούμε ευαίσθητα προσωπικά δεδομένα, όπως οικονομικά ή ιατρικά δεδομένα, είναι απαραίτητο τα πληροφοριακά συστήματα που χρησιμοποιούνται να ευθυγραμμίζονται με το ανάλογο κανονιστικό πλαίσιο. Οι κανονισμοί αυτοί ενδέχεται να απαγορεύουν τη χρήση συστημάτων που δεν διαθέτουν τις απαιτούμενες πιστοποιήσεις [28].

Ωστόσο, η πρακτικότητα δεν είναι το μόνο επιχείρημα υπέρ της χρήσης σχεσιακών βάσεων δεδομένων για την εκτέλεση εργασιών επεξεργασίας γραφημάτων. Σημαντικό ρόλο διαδραματίζει και το γεγονός πως η ερευνητική κοινότητα έχει μελετήσει εκτενώς τις σχεσιακές βάσεις δεδομένων και έχει συσσωρεύσει τεράστια γνώση γύρω από αυτές. Κατά συνέπεια, η ανάπτυξη ενός συστήματος επεξεργασίας γραφημάτων που βασίζεται σε μια βάση δεδομένων σχέσεων παρέχει τη δυνατότητα να επωφεληθεί άμεσα από τα ερευνητικά αποτελέσματα που έχουν προκύψει μετά από δεκαετίες έρευνας. Συναλλαγές (transactions), ανοχή σε σφάλματα (fault tolerance), παραλληλία (concurrency), ακεραιότητα (integrity constraints), ασφάλεια και βελτιστοποίηση ερωτημάτων (query optimization) είναι μόνο μερικά από τα πιο αξιοσημείωτα χαρακτηριστικά που προσφέρουν οι υπάρχουσες σχεσιακές βάσεις δεδομένων [24][28]. Αυτά τα χαρακτηριστικά είναι εξαιρετικά πολύτιμα για τους εταιρικούς πελάτες που έχουν την ευθύνη για την ανάπτυξη κρίσιμων εφαρμογών. Τα συστήματα επεξεργασίας γραφημάτων που έχουν χτιστεί πάνω σε μια σχεσιακή βάση δεδομένων μπορούν να χρησιμοποιήσουν αυτές τις δυνατότητες αμέσως. Αντιθέτως, τα συστήματα που αναπτύσσονται από την αρχή είναι υποχρεωμένα να τις υλοποιήσουν από την αρχή, αν θέλουν να παραμείνουν ανταγωνιστικά στο εμπόριο. Φυσικά, η υλοποίηση όλων αυτών των χαρακτηριστικών απαιτεί πολύ κόπο και μεγάλο κόστος.

Επιπλέον, αξίζει να αναφερθούν οι διαφορές ανάμεσα στην επιστράτευση ενός RDBMS για ανάλυση γραφημάτων έναντι ενός εξειδικευμένου in-memory συστήματος. Αν και τα in-memory συστήματα επεξεργασίας γραφημάτων επιτυγχάνουν καλύτερη απόδοση για σύνολα δεδομένων μέτριου μεγέθους, αποτυγχάνουν να επεξεργαστούν μεγαλύτερα σύνολα δεδομένων, καθώς τα ενδιάμεσα αποτελέσματα που παράγονται δεν χωράνε στη μνήμη ενός υπολογιστή [16]. Για να ξεπεραστεί αυτό το πρόβλημα, οι λύσεις αυτές πρέπει να εκτελούνται με κατανεμημένο (distributed) τρόπο, χρησιμοποιώντας πολλούς υπολογιστές. Ερευνητικά αποτελέσματα δείχνουν ότι μία ομάδα 32 υπολογιστών εκτελεί μια εργασία επεξεργασίας γραφήματος κατά μια τάξη μεγέθους πιο γρήγορα από ένα RDBMS

που εκτελείται σε έναν μόνο κόμβο [16]. Παρόλο που αυτή είναι αναμφίβολα μια σημαντική διαφορά στην επίδοση, αν λάβουμε υπόψη όλα τα έξοδα, συμπεριλαμβανομένων των εξόδων για υπολογιστικούς πόρους και του κόστους διαχείρισης, είναι πιθανό η εναλλακτική του RDBMS να είναι πιο οικονομική. Με άλλα λόγια, παρά το γεγονός ότι τα κατανεμημένα συστήματα επεξεργασίας γράφημάτων υπερτερούν ως προς τις επιδόσεις τους, ένα RDBMS μπορεί να είναι η καλύτερη επιλογή για εφαρμογές όπου οι επιδόσεις δεν είναι η πρώτη προτεραιότητα, καθώς ένα κατανεμημένο σύστημα μπορεί να είναι πολύ δαπανηρό.

Η κυριαρχία των σχεσιακών βάσεων δεδομένων, σε συνδυασμό με τη δημοτικότητα που έχει αποκτήσει η ανάλυση γραφημάτων τα τελευταία χρόνια, οδήγησε σε πολλές ερευνητικές προσπάθειες που σχετίζονται με την αποδοτική ανάλυση γραφημάτων με τη χρήση σχεσιακών βάσεων.

Το Grail [16] είναι ένα σύστημα που αποθηκεύει κορυφές και ακμές σε μορφή πίνακα και παρέχει μια φιλική προς τον προγραμματιστή διεπαφή που του επιτρέπει να εκφράσει ερωτήματα για γραφήματα με βάση την vertex-centric φιλοσοφία. Στη συνέχεια, ένας ”μεταφραστής” (translator) αντιστοιχίζει αυτά τα ερωτήματα σε ισοδύναμα ερωτήματα SQL και ένας optimizer βελτιώνει τα παραγόμενα SQL statements. Στη συνέχεια, η βελτιστοποιημένη SQL εκτελείται από μια σχεσιακή βάση δεδομένων. Παρομοίως, το Vertexica [24] είναι ένα εργαλείο ανάλυσης γραφημάτων που λαμβάνει ως είσοδο vertex-centric ερωτήματα που είναι παρεμφερή με εκείνα του Pregel και χρησιμοποιεί user-defined functions για να τα μετατρέψει σε κλασσικά ερωτήματα SQL που θα εκτελεστούν από ένα RDBMS. Το SQLGraph [38] μεταφράζει ερωτήματα για γρ σε SQL και χρησιμοποιεί ένα σχεσιακό schema για την αποθήκευση πληροφοριών γειτνίασης. Το GRFusion [20] είναι ένα σύστημα που τροποποιεί το εσωτερικό μιας σχεσιακής βάσης δεδομένων προκειμένου να ενσωματώσει τόσο το σχεσιακό όσο και το graph data model σε ένα ενοποιημένο query engine. Για να το πετύχει αυτό, ορίζει γραφήματα μέσω προβολών (views) και στη συνέχεια υλοποιεί ένα ευρετήριο (graph index) το οποίο αντικατοπτρίζει την τοπολογία του γραφήματος εντός της μνήμης. Οι κόμβοι και οι άκρες του ευρετηρίου περιέχουν δείκτες προς τα αντίστοιχα σχεσιακά δεδομένα, δίνοντάς μας τη δυνατότητα να εκτελούμε αποτελεσματικά τελεστές μέσω του in-memory ευρετηρίου και να ανακτούμε τα σχεσιακά δεδομένα μόνο όταν τα χρειαζόμαστε. Το GraphGen [46] είναι ένα εργαλείο που καθιστά εφικτή την εξαγωγή γραφημάτων μεγάλης κλίμακας που είναι αποθηκευμένα σε μορφή πίνακα μέσα σε μια σχεσιακή βάση δεδομένων. Το GRainDB [23] επεκτείνει το DuckDB [33] με δυνατότητες επεξεργασίας γραφημάτων, εφαρμόζοντας τεχνικές που επιταχύνουν τους τελεστές `JOIN` με αποτέλεσμα να εξυπηρετούνται καλύτερα τα υπολογιστικά φορτία που αφορούν επεξεργασία γράφων.

0.2 Τύποι υπολογιστικών φόρτων στην ανάλυση γραφημάτων

Κατ’ αναλογία με τον χαρακτηρισμό των υπολογιστικών φορτίων των σχεσιακών βάσεων σε transactional και σε analytical, διακρίνουμε επίσης δύο διαφορετικά υπολογιστικά φορτία για την επεξεργασία γραφημάτων. Συγκεκριμένα, διαχωρίζουμε τα υπολογιστικά φορτία για την επεξεργασία γράφων σε *ερωτήματα* (graph queries) και *αλγορίθμους* (graph algorithms).

Τα ερωτήματα αναφέρονται σε αναζήτηση υπογράφων που ταιριάζουν σε κάποιο δοθέν μοτίβων [42]. Αυτά τα αιτήματα αφορούν μόνο με μια μικρή περιοχή του γραφήματος [42]. Η εύρεση της συντομότερης διαδρομής μεταξύ δύο κορυφών είναι ένα αντιπροσωπευτικό παράδειγμα ερώτημα. Οι αλγόριθμοι περιλαμβάνουν επαναληπτικές υπολογισμούς που εκτελούνται σε ολόκληρο το γράφημα

[42]. Παραδείγματα αλγορίθμων είναι οι αλγόριθμοι PageRank και Bellman-Ford.

Λόγω του διαρκώς αυξανόμενου ενδιαφέροντος για εφαρμογές μηχανικής μάθησης, μεγάλη έμφαση δίνεται πλέον και στον υπολογισμό graph embeddings, τα οποία αξιοποιούνται κατά την εκπαίδευση νευρωνικών δικτύων για γραφήματα (GNN) [42] . Τέτοιου είδους υπολογιστικά φορτία συνιστούν μια ειδική κατηγορία αλγορίθμων, η οποία ονομάζεται *graph ML*.

0.3 Γλώσσες ερωτημάτων

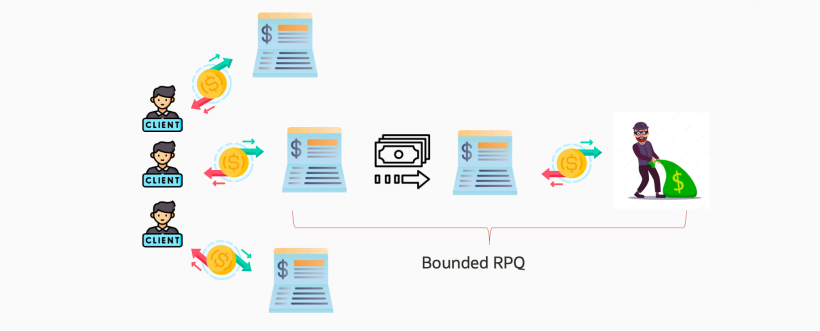
Μια *γλώσσα ερωτημάτων* (query language) είναι μια γλώσσα μέσω της οποίας οι χρήστες μπορούν να εκφράσουν ερωτήματα σε μια βάση δεδομένων [37]. Οι γλώσσες ερωτημάτων τείνουν να είναι γλώσσες υψηλού επιπέδου συγκριτικά με το επίπεδο αφαίρεσης που χρησιμοποιούν οι γλώσσες προγραμματισμού. Μπορούν να κατηγοριοποιηθούν σε διαδικαστικές και μη-διαδικαστικές γλώσσες. Στις διαδικαστικές γλώσσες, το ερώτημα καθοδηγεί τη βάση δεδομένων σχετικά με τα βήματα που πρέπει να ακολουθήσει για να παράξει τα επιθυμητά αποτελέσματα [37]. Στις μη-διαδικαστικές γλώσσες, το ερώτημα απλώς περιγράφει το επιθυμητό αποτέλεσμα του ερωτήματος χωρίς να προσδιορίζει συγκεκριμένα βήματα εκτέλεσης. Στις μη-διαδικαστικές γλώσσες οι αποφάσεις που σχετίζονται με την εκτέλεση ανατίθενται στη βάση δεδομένων [37]. Υπάρχουν πολλές γλώσσες ερωτημάτων που έχουν φτιαχτεί με στόχο να διευκολύνουν τη συγγραφή ερωτημάτων για γραφήματα. Κάποιες εκ των πιο σημαντικών είναι οι SPARQL[1], Cypher[18], PGQL[44], SQL/PGQ[45] και GQL[17].

0.4 Μη-φραγμένα κανονικά ερωτήματα μονοπατιών

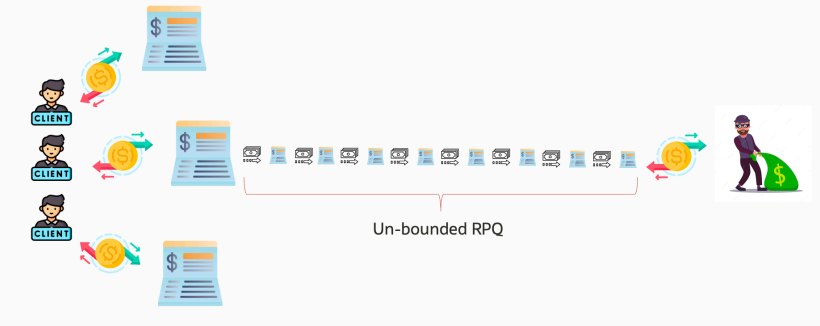
Μία από τις πιο συνηθισμένες εργασίες που εκτελούνται από χρήστες που αναλύουν δεδομένα γραφήματος είναι η εύρεση μονοπατιών που ταυτίζονται με κάποιο μοτίβο (pattern matching). Οι χρήστες μπορούν να υποβάλουν ερωτήματα που αφορούν το γράφημα συντάσσοντας ερωτήματα σε κάποια γλώσσα ερωτημάτων για γραφήματα όπως η PGQL [44]. Τα μοτίβα μπορεί να είναι είτε μοτίβα σταθερού μήκους (για παράδειγμα¨ `(p1:Person)-[:Knows]->(p2:Person)`), είτε μοτίβα μεταβλητού μήκους, τα οποία εκφράζονται με τη χρήση κανονικών εκφράσεων (regular expressions). Αυτά τα ερωτήματα ονομάζονται ερωτήματα κανονικής διαδρομής (regular path queries - RPQ) και αποτελούν ένα πολύ ισχυρό εργαλείο στα χέρια των αναλυτών δεδομένων, καθώς τους επιτρέπουν να εκφράσουν σύνθετα ερωτήματα με απλό τρόπο. Ωστόσο, η εκτέλεση τους είναι μια υπολογιστική εργασία που συνοδεύεται από πολλές προκλήσεις, ιδιαίτερα λόγω του κινδύνου έκρηξης των ενδιάμεσων αποτελεσμάτων που παράγονται.

Τα ερωτήματα κανονικής διαδρομής (RPQ) μπορούν να ταξινομηθούν σε φραγμένα RPQ και μη-φραγμένα RPQ, ανάλογα με το αν υπάρχει άνω φράγμα στο μήκος του μονοπατιού που αναζητούμε. Για παράδειγμα, το `(p1:Person)-[:Knows]->{,2}(p2:Person)` αναζητά τις συνδέσεις μεταξύ ατόμων που είτε γνωρίζονται μεταξύ τους είτε έχουν μία κοινή γνωριμία (”φίλος φίλου”). Αυτό το ερώτημα είναι ένα φραγμένο RPQ, επειδή το μέγιστο μήκος ενός μονοπατιού που ταιριάζει με αυτό το μοτίβο είναι 2. Από την άλλη πλευρά, το `(p1:Person)-[:Knows]->*(p2:Person)` ακολουθεί άπειρες συνδέσεις ατόμων (”Α ξέρει κάποιον Β που ξέρει κάποιον C που ξέρει κάποιον ...”). Αυτό το ερώτημα είναι ένα μη-φραγμένο RPQ, επειδή το μέγιστο μήκος του μονοπατιού δεν περιορίζεται από κάποιο άνω φράγμα.

Τα μη-φραγμένα RPQ αξιοποιούνται σε ένα ευρύ φάσμα εφαρμογών. Ένα πρακτικό παράδειγμα που αναδεικνύει την ισχύ των μη-φραγμένων RPQ είναι η ανίχνευση ύποπτων συναλλαγών στον χρηματοπιστωτικό κλάδο. Για παράδειγμα, σε υποθέσεις ξεπλύματος μαύρου χρήματος, οι εγκληματίες συχνά πραγματοποιούν πολύπλοκες μεταφορές χρημάτων προκειμένου να εμποδίσουν τις αρχές από το να εντοπίσουν την πραγματική προέλευση κεφαλαίων που έχουν αποκτηθεί από παράνομες δραστηριότητες. Για την παρακολούθηση των χρηματικών ροών, τα χρηματοπιστωτικά ιδρύματα δημιουργούν τεράστια γραφήματα στα οποία οι τραπεζικοί λογαριασμοί αναπαρίστανται ως κορυφές και οι συναλλαγές μεταξύ λογαριασμών αναπαρίστανται ως κατευθυνόμενες ακμές. Με τη χρήση μη-φραγμένων RPQ, ερωτήματα που βοηθούν στον εντοπισμό μοτίβων, όπως για παράδειγμα στον εντοπισμό κύκλων στο γράφημα, μπορούν να εκφραστούν με απλό τρόπο, καθιστώντας το ευκολότερο για τους ερευνητές να ανιχνεύσουν ύποπτες ακολουθίες συναλλαγών.



(a) Παράδειγμα: φραγμένα κανονικά ερωτήματα μονοπατίων



(b) Μη-φραγμένα κανονικά ερωτήματα μονοπατίων

Σχήμα 0.2: Φραγμένα και μη-φραγμένα κανονικά ερωτήματα μονοπατίων [43]

0.5 Υπάρχουσες προσεγγίσεις

Παρά τη σημαντικότητα των μη-φραγμένων RPQs, οι υπάρχουσες τεχνικές για την εκτέλεση τους μέσα σε σχεσιακές βάσεις δεδομένων δεν αποδίδουν επαρκώς.

Μια προσέγγιση αναφορικά με την εκτέλεση των μη-φραγμένων RPQ είναι η σύνταξη ερωτημάτων `RECURSIVE WITH`. Ωστόσο, τέτοιου είδους ερωτήματα είναι δύσκολο να γραφτούν. Επιπλέον, εξαρτώνται από το schema του γραφήματος. Αυτό σημαίνει ότι πρέπει να γράφονται εξαρχής κάθε φορά που προστίθεται ένας νέος πίνακας στον ορισμό του γραφήματος, για τον οποίο εκτελείται το μη-φραγμένο RPQ. Ακόμα, το `RECURSIVE WITH` παράγει πιο γενικά αποτελέσματα από αυτά που απαιτούνται για το μη-φραγμένο RPQ. Αυτό επηρεάζει αρνητικά την απόδοσή τους, καθώς δαπανάται

υπολογιστικός χρόνος για την ανακάλυψη λύσεων που τελικά θα απορριφθούν. Τέλος, το `RECURSIVE WITH` είναι ένας πολύπλοκος και όχι καλά βελτιστοποιημένος τελεστής.

Μια δεύτερη προσέγγιση για την εκτέλεση των μη-φραγμένων RPQ είναι η σύνταξη ενός PL/SQL script που θα διασχίζει το γράφημα. Το PL/SQL script θα εφαρμόζει στην ουσία έναν αλγόριθμο αναζήτησης κατά πλάτος. Η απόδοση αυτής της λύσης πάσχει από το κόστος του interpretation που χαρακτηρίζει την PL/SQL. Επιπλέον, χρησιμοποιεί μόνο τις υπάρχουσες δυνατότητες SQL, πριν ενταχθούν σε αυτή δυνατότητες που αφορούν την επεξεργασία γραφημάτων.

Όταν η διάμετρος του γραφήματος είναι γνωστή εκ των προτέρων, μπορούμε να εκτελέσουμε τα μη-φραγμένα RPQs μετατρέποντάς τα σε φραγμένα RPQ των οποίων το άνω φράγμα είναι ίσο με το μεγαλύτερο μονοπάτι στο γράφημα. Μια προσέγγιση για την εκτέλεση των φραγμένων RPQs είναι η μετάφραση τους σε ισοδύναμη SQL που περιλαμβάνει πολλά `JOIN` και `UNION ALL`. Ωστόσο, αυτή η τεχνική πάσχει λόγω της πιθανής έκρηξης των ενδιάμεσων αποτελεσμάτων για διαδρομές με μεγάλα μήκη, εξαιτίας των διαδοχικών τελεστών `JOIN`.

0.6 Αλγόριθμος υπολογισμού μη-φραγμένων κανονικών ερωτημάτων μονοπατιών

Η απόδοση αυτών των προσεγγίσεων δεν είναι επαρκής. Τα μειονεκτήματά τους οφείλονται στο γεγονός ότι εκτελούνται σε προϋπάρχουσες SQL engines που δεν παρέχουν εξειδικευμένη υποστήριξη για μη-φραγμένα RPQs. Βελτιωμένη απόδοση θα μπορούσε να επιτευχθεί εάν ο μηχανισμός που εκτελεί τα μη-φραγμένα RPQs είχε πρόσβαση στους εσωτερικούς μηχανισμούς της σχεσιακής βάσης δεδομένων. Για να καλυφθεί αυτό το κενό, έχει αναπτυχθεί ένας εξειδικευμένος πυρήνας που είναι ενσωματωμένος σε μια σχεσιακή βάση δεδομένων.

0.6.1 Μοντέλο δεδομενων

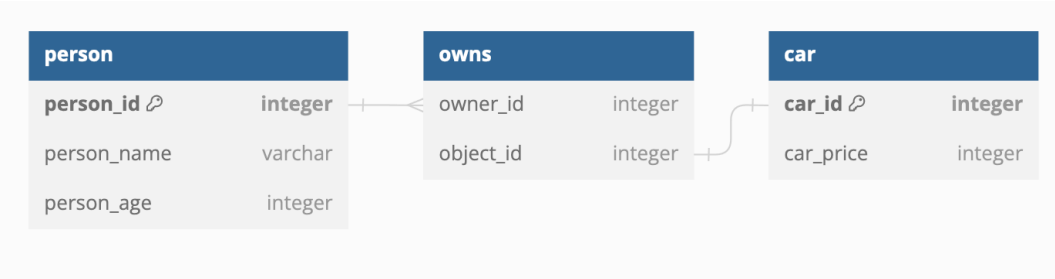
Το *μοντέλο δεδομένων* (data model) μιας βάσης δεδομένων είναι ένα σύνολο εννοιών που μας επιτρέπουν να περιγράψουμε τα δεδομένα, τις σχέσεις μεταξύ τους, τη σημασιολογία τους και τους περιορισμούς που ισχύουν για αυτά [37]. Ένα από τα πιο διαδεδομένα μοντέλα δεδομένων για γραφήματα είναι οι *γραφήματα ιδιοτήτων* (property graphs). Ένα γράφημα ιδιοτήτων είναι ένα κατευθυνόμενο γράφημα στο οποίο τόσο οι κορυφές όσο και οι ακμές μπορούν να έχουν απεριόριστο αριθμό ιδιοτήτων (properties) [42]. Επιπλέον, στις κορυφές και τις ακμές μπορούν να ανατεθούν ετικέτες (labels) που καθιστούν εφικτό τον διαχωρισμό μεταξύ διαφορετικών τύπων αντικειμένων και σχέσεων μέσα στο γράφημα [42].

Σε αυτή την υποενότητα θα περιγράψουμε το μοντέλο δεδομένων που χρησιμοποιείται για τον ορισμό γραφημάτων ιδιοτήτων εντός μιας σχεσιακής βάσης. Ας υποθέσουμε ότι έχουμε τρεις πίνακες T_s, T_e, T_d τέτοιους ώστε το T_e να δηλώνει foreign keys για αμφότερους τους T_s και T_d. Η τριάδα (triplet) $<T_s, T_e, T_d>$ μπορεί να χρησιμοποιηθεί για τον ορισμό συνδέσεων σε ένα γράφημα ιδιοτήτων. Ειδικότερα:

- κάθε σειρά στους T_s και T_d μπορεί να ερμηνευθεί ως κορυφή του γραφήματος
- κάθε σειρά στον T_e μπορεί να ερμηνευθεί ως μια κατευθυνόμενη ακμή που συνδέει δύο κορυφές που αντιστοιχούν στις σειρές των T_s και T_d, οι οποίες προσδιορίζονται μέσω των foreign keys

- οι στήλες του T_e μπορούν να ερμηνευθούν ως ιδιότητες των ακμών και οι στήλες των T_s και T_d μπορούν να ερμηνευτούν ως ιδιότητες των κόμβων

Οι T_s, οι T_e και οι T_d ονομάζονται *πίνακες στοιχείων γράφου* (graph element tables). Συγκεκριμένα, οι T_s και T_d είναι *πίνακες κόμβων* (vertex table) και ο T_e είναι *πίνακας ακμών* (edge table). Κάθε πίνακας ακμών ορίζει μια κατευθυνόμενη σχέση μεταξύ των δύο πινάκων κόμβων. Αυτοί οι πίνακες κόμβων αποκαλούνται πίνακας κόμβων πηγής (source vertex table) και πίνακας κόμβων προορισμού (destination vertex table) αντίστοιχα. Ένας πίνακας μπορεί να εμφανιστεί περισσότερες από μία φορές στην ίδια τριάδα.



Σχήμα 0.3: Παράδειγμα ενος schema μιας τριάδας πηγή-ακμή-προορισμός

Ένα σύνολο τέτοιων τριάδων μπορεί να ορίσει ένα γράφημα ιδιοτήτων σε μια σχεσιακή βάση δεδομένων. Τα γραφήματα ιδιοτήτων αναπαρίστανται ως SQL schema objects που αντιπροσωπεύουν γραφήματα μέσω πινάκων που ήδη υπάρχουν μέσα σε μια βάση δεδομένων. Αυτοί οι πίνακες ονομάζονται επίσης *underlying objects*.

Τα γραφήματα ιδιοτήτων δεν αποθηκεύουν δεδομένα. Τα δεδομένα βρίσκονται αποθηκευμένα στα υποκείμενα αντικείμενα που χρησιμοποιήθηκαν για τον ορισμό του γραφήματος. Υπό αυτή την έννοια, τα γραφήματα ιδιοτήτων λειτουργούν σαν προβολές που παρέχει ένα επίπεδο αφαίρεσης για τη διαχείριση γραφημάτων που έχουν αποθηκευτεί σε μια σχεσιακή βάση δεδομένων.

Τα γραφήματα ιδιοτήτων μπορούν να αναζητηθούν με τη χρήση επεκτάσεων SQL για την επεξεργασία γραφημάτων. Οι τροποποιήσεις στα υποκείμενα αντικείμενα που χρησιμοποιούνται για τον ορισμό του γραφήματος ιδιοτήτων είναι άμεσα ορατές στα ερωτήματα του γραφήματος ιδιοτήτων.

Οι χρήστες μπορούν να γράψουν ερωτήματα που αφορούν γραφήματα ιδιοτήτων εκμεταλλευόμενοι τις σχετικές επεκτάσεις της SQL.

0.6.2 Περιγραφή αλγορίθμου

Η εκτέλεση αλγορίθμων για την ανάλυση γραφήματα ιδιοτήτων επιτρέπει στους χρήστες να αποκτήσουν πληροφορίες από γραφήματα που είναι αποθηκευμένα μέσα σε μια σχεσιακή βάση δεδομένων. Οι αλγόριθμοι γραφημάτων εκτελούν εργασίες ανάλυσης γραφημάτων που λαμβάνουν υπόψη τόσο τη δομή του γραφήματος όσο και τις ιδιότητες των κορυφών ή/και των άκρων του.

Οι αλγόριθμοι γραφημάτων καλούνται από ένα ερώτημα σχετικά με το γράφημα. Προκειμένου να εκτελεστούν οι υπολογισμοί που προσδιορίζονται από τα βήματα του αλγορίθμου στο πλαίσιο της σχεσιακής βάσης δεδομένων, η υλοποίηση εκτελεί ερωτήματα SQL δυναμικά. Συνεπώς, η εκτέλεση του αλγορίθμου βασίζεται στην SQL. Για παράδειγμα, εάν ο αλγόριθμος χρειάζεται να βρει τους γείτονες μιας κορυφής, εκτελεί ένα ερώτημα `SELECT` για να τους ανακτήσει, αφού πρώτα ενώσει τους

κατάλληλους πίνακες (βλέπε §3.3.3). Αυτός ο σχεδιασμός επιτρέπει τη χρήση καθιερωμένων τεχνικών βελτιστοποίησης ερωτημάτων (query optimization), των οποίων η υλοποίηση υπάρχει ήδη στο RDBMS. Αξίζει να σημειωθεί ότι η συντριπτική πλειοψηφία του υπολογιστικού χρόνου του αλγορίθμου αφιερώνεται στην εκτέλεση αυτών των SQL ερωτημάτων.

Για να κάνει τα αποτελέσματά του προσβάσιμα, ο αλγόριθμος δημιουργεί νέες ιδιότητες και τις χρησιμοποιεί για να αποθηκεύσει τα αποτελέσματα του. Αυτές οι ιδιότητες ονομάζονται *ιδιότητες εξόδου*. Για παράδειγμα, ο αλγόριθμος PageRank [10] δημιουργεί μια ιδιότητα εξόδου για κάθε κορυφή, ώστε για να αποθηκεύει την κατάταξη που υπολόγισε για αυτήν.

Επειδή οι αλγόριθμοι γραφημάτων καλούνται μέσω ερωτημάτων, η έξοδός τους πρέπει να είναι διαθέσιμη μέσω SQL. Επιπρόσθετα, οι αλγόριθμοι γραφημάτων παράγουν τεράστιους όγκους ενδιάμεσων αποτελεσμάτων, τα οποία πρέπει να αποθηκεύονται αποτελεσματικά και να μπορούν να ανακτηθούν από τα ερωτήματα SQL που αποτελούν τον κορμό του αλγορίθμου. Για την ικανοποίηση αυτών των απαιτήσεων, δημιουργούνται πίνακες στους οποίους αποθηκεύονται τόσο τα ενδιάμεσα όσο και τα τελικά αποτελέσματα. Η χρήση πινάκων παρέχει έναν φυσικό τρόπο ανάκτησης της εξόδου του αλγορίθμου μέσω SQL.

Οι προαναφερθέντες πίνακες δημιουργούνται κατά το compilation του ερωτήματος που προκάλεσε την εκτέλεση του αλγορίθμου. Συγκεκριμένα, δημιουργείται ένας πίνακας ανά πίνακα κόμβων. Για παράδειγμα, εάν το γράφημα αποτελείται από δύο πίνακες κορυφών και τα αποτελέσματα του ερωτήματος εκφράζονται μέσω μιας ιδιότητας εξόδου, θα δημιουργηθούν δύο πίνακες. Καθένας από αυτούς τους πίνακες ονομάζεται *πλευρικός πίνακας* (side table) και ο αντίστοιχος πίνακας κόμβων ονομάζεται *πρωτεύων πίνακας* (primary table). Οι κύριοι πίνακες περιέχουν τις αρχικές ιδιότητες του γραφήματος και οι πλευρικοί πίνακες αποθηκεύουν τις ιδιότητες που δημιουργούνται από τον αλγόριθμο. Κάθε πλευρικός πίνακας έχει στήλες που λειτουργούν ως foreign key προς του κύριους πίνακες, καθώς και στήλες που χρησιμοποιούνται για την αποθήκευση των ιδιοτήτων εξόδου του αλγορίθμου. Όταν ένα ερώτημα γραφήματος επιχειρεί να ανακτήσει ταυτόχρονα ιδιότητες εξόδου όσο και ιδιότητες εισόδου, εκτελείται ένα `JOIN` μεταξύ του πρωτεύοντος πίνακα και του πλευρικού πίνακα.

0.6.3 Εκτέλεση αλγορίθμου

Τα μη-φραγμένα RPQs προσπαθούν να αντιστοιχίσουν ένα μοτίβο της μορφής `(a)-[e]->*(b)`, όπου το `*` αναφέρεται στο αστέρι του Kleene και τα `a` και `b` μπορεί να είναι σύνθετα ερωτήματα. Για να εκτελεστεί ένα μη-φραγμένο RPQ, χωρίζεται σε τρία μέρη: το *πρόθεμα*, την *κατάληξη* και το *αναδρομικό μέρος*. Αυτός ο διαχωρισμός λαμβάνει χώρα κατά το query compilation, αφού πρώτα το ερώτημα έχει αναλυθεί συντακτικά και σημασιολογικά.

Το πρόθεμα και το επίθημα μπορούν να εκτελεστούν χρησιμοποιώντας τις υπάρχουσες δυνατότητες του συστήματος. Το αναδρομικό τμήμα του ερωτήματος λαμβάνει ως είσοδο μια κορυφή s, η οποία είναι το τέλος του προθέματος. Ο αλγόριθμος έχει ως στόχο την εύρεση της συντομότερης διαδρομής προς κάθε κορυφή που είναι προσβάσιμη από την δοθείσα κορυφή s. Επιπλέον, ο αλγόριθμος θα πρέπει να αποθηκεύει τις απαιτούμενες πληροφορίες ώστε να μπορούν να απαντηθούν ερωτήματα και συναθροίσεις (aggregations) που αφορούν τις κορυφές που αποτελούν κάθε διαδρομή. Για να επιτύχει το στόχο αυτό, εκτελεί μια **αναζήτηση κατά πλάτος** (breadth-first search) που διασχίζει το γράφημα ξεκινώντας από s.

Κατά την εκτέλεση του αλγορίθμου:

- ένα αποτέλεσμα του προθέματος δίνεται ως είσοδος στο αναδρομικό μέρος του ερωτήματος
- το αναδρομικό τμήμα του ερωτήματος εκτελείται μέσω μιας αναζήτησης κατά πλάτος του οποίου η έξοδος αποθηκεύεται σε έναν πλευρικό πίνακα
- το επίθημα διαβάζει τα αποτελέσματα του αναδρομικού μέρους από τους πλευρικούς πίνακες για να τα χρησιμοποιήσει ως είσοδο
- το πρόθεμα και το επίθημα ενώνονται μέσω ενός `JOIN`

0.7 Συνεισφορά

Οι επιδόσεις του πυρήνα χρήζουν βελτίωσης. Ένα σημαντικό πρόβλημα με την υλοποίηση του πυρήνα είναι ότι χειρίζεται τις εντολές SQL με υποβέλτιστο τρόπο. Σε αυτή την εργασία βελτιώνουμε τις επιδόσεις του εξειδικευμένου πυρήνα που χρησιμοποιείται από τη σχεσιακή βάση δεδομένων για την εκτέλεση μη-φραγμένων RPQs. Εντοπίζουμε τα κρισιμα προβληματα που σχετίζονται με την εκτέλεση του αλγορίθμου και τα αντιμετωπίζουμε εφαρμόζοντας διάφορες βελτιστοποιήσεις. Η εργασία μας περιορίζεται στα μη-φραγμένα RPQs που αναζητούν μόνο ένα συντομότερο μονοπάτι, αλλά οι προτεινόμενες λύσεις γενικεύονται και σε άλλες σημασιολογίες. Ο βελτιστοποιημένος πυρήνας επιτυγχάνει έως και x67 βελτιωμένη επίδοση χωρίς να αυξάνει την κατανάλωση μνήμης του συστήματος.

Κείμενο στα αγγλικά

Chapter 1

Introduction

1.1 Context

A graph is a data structure that represents relationships between entities. Due to their ability to efficiently represent connections, graphs are used in a broad spectrum of applications. For example, graphs empower recommendation systems, search engines and fraud detection.

The popularity of graph applications drives a demand for software that enables the efficient analysis of graphs. Graph analytics solutions are in great demand by data scientists that are interested in extracting valuable insights from graph-structured data. These solutions must be capable of processing huge amounts of data, as industry practitioners in fields like retail, finance and social media are often tasked with discovering patterns that are hidden in graphs that are massive in size.

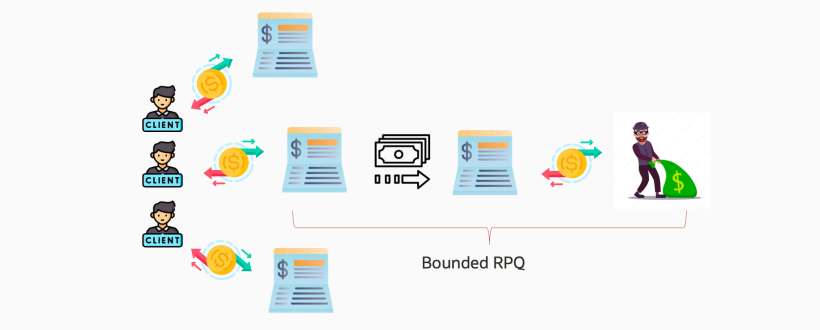
Even though specialized graph processing systems have been developed, for many organizations relational databases are the de-facto solution when it comes to data storage and processing. Because of the wide spread adoption of relational databases, it is common that large-scale graphs are stored in tabular format inside a RDBMS. In these cases, in order to deploy a graph analytics framework, it is required to import the graph data from the relational database and to set up ETL processes to ensure data freshness. For many scenarios, the increase in system complexity and maintenance costs is not justified by the advantages that a specialized system offers. To fill this gap, relational databases began offering graph processing functionalities in order to allow data scientists to perform graph processing tasks inside the database.

One of the most common tasks performed by users that analyze graph data is the retrieval of paths that match a specific pattern. Users are able to query the graph by writing graph queries in a graph query language such as PGQL [44]. The patterns can be either fixed-length patterns, such as `(p1:Person)-[:Knows]->(p2:Person)`, or variable-length patterns, which are expressed with the use of regular expressions. These queries are called regular path queries (RPQs) and they are a very powerful tool as they allow users to easily express complex queries. However, their evaluation is a computational task that comes with many challenges, particularly due to the danger of intermediate result explosion when querying large datasets.

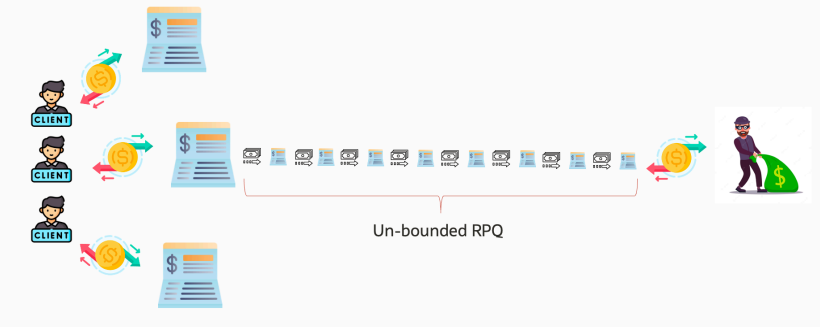
Regular path queries (RPQs) can be classified into bounded RPQs and unbounded RPQs, according to whether there is an upper bound to the length of the path we aim to retrieve. For example, `(p1:Person)-[:Knows]->{,2}(p2:Person)` retrieves connections between people that either know each other or have a common acquaintance (”friend-of-a-friend” relationship). This query is a bounded RPQ, because the maximum length of a path that matches this pattern is 2. On the other hand `(p1:Person)-[:Knows]->*(p2:Person)` follows connections of people across an unrestricted

number of friendships ("A knows B who knows C who knows ..."). This query is an unbounded RPQ, because the maximum length of the pattern is not restricted by an upper bound.

Unbounded RPQs are utilized in a diverse range of use cases. A practical scenario that showcases the power of unbounded RPQs is fraud detection in the financial industry. For example, in money laundering cases, criminals carry out series of complex money transfers in order to prevent the authorities from tracking the origin of funds that have been acquired through illegal activities. To monitor money flows, financial institutions generate huge graphs in which bank accounts are represented as vertices and transactions between accounts are represented as directed edges. With unbounded RPQs, queries that identify patterns such as cycles in the graph can be expressed in a simple manner, making it easier for investigators to detect suspicious transaction patterns.



(a) Bounded recursive path queries example



(b) Unbounded recursive path queries example

Figure 1.1: Unbounded recursive path queries illustration in the context of fraud detection [43]

1.2 Motivation

Despite the fact that unbounded RPQs are valuable in numerous real-life scenarios, existing techniques to evaluate them in the context of a relational database do not perform well enough.

One approach to evaluate unbounded RPQs is to write `RECURSIVE WITH` queries. However, these queries are hard to write. Moreover, they are schema-dependent. This means that they must be rewritten when a new table is added to the definition of the property graph over which the unbounded RPQ is executed. Additionally, the `RECURSIVE WITH` produce more generic results than those that are needed by the unbounded RPQ. This impacts their performance negatively, as computational time is spent on discovering solutions that will be eventually thrown away. Lastly, `RECURSIVE WITH` is a complex and not well-optimized operator.

A second approach is to execute a PL/SQL script that will perform graph traversal. This algorithm will essentially implement a breadth-first search algorithm. The performance of this solution suffers from the interpretation overhead that characterizes PL/SQL. Furthermore, it only utilizes existing SQL capabilities, before the introduction of graph processing features.

In cases where the diameter of the graph is known, we can evaluate unbounded RPQs by converting them to bounded RPQs whose bound is equal to the longest path in the graph. A common approach to evaluate bounded RPQs is to translate the graph query to equivalent SQL that involves many `JOIN`s and `UNION ALL`. Still, this technique suffers from intermediate result explosion for paths with large lengths due to the consecutive `JOIN` operations.

The performance of these approaches is not sufficient. Their disadvantages come from the fact that they are executed over existing SQL engines that do not provide specialized support for unbounded RPQs. Improved performance could be achieved if the mechanism that evaluates unbounded RPQs had access to the internals of the relational database. To fill this gap, a specialized kernel that is tightly integrated with a relational database has been developed.

Nevertheless, the performance of the aforementioned kernel is not good enough. For instance, the execution time for an unbounded RPQ that explores a graph of approximately 60K vertices and 150K edges exceeds 15 minutes. A major problem with the implementation of the kernel is that it handles SQL statements in a suboptimal way.

1.3 Contribution

In this work we enhance the performance of the specialized kernel that is used by the relational database to evaluate unbounded RPQs. We identify performance bottlenecks related to the SQL-based execution of the existing algorithm and we address these bottlenecks by implementing optimizations. Specifically, we implement insertion batching for the algorithm's output, prepared statements, and batching for vertex neighbour fetching. Our work targets unbounded regular path queries that search for a single shortest path, but the proposed solutions generalize to broader semantics as well. The optimized kernel achieves up to x67 end-to-end performance improvement without increasing the system's memory consumption.

1.4 Thesis outline

This thesis includes eight chapters:

- In **Chapter 2** we review existing literature relevant to this thesis
- In **Chapter 3** we explain the design of the original system
- In **Chapter 4** we present the optimizations we implemented
- In **Chapter 5** we demonstrate the experimental results of this work
- In **Chapter 6** we discuss future directions
- In **Chapter 7** we provide a summary of our work

Chapter 2

Related Work

2.1 Relational databases

A *database-management system* (DBMS) is a collection of interrelated data and a set of programs to access those data [37]. The DBMS specifies the data structures in which the data will be stored, so that the users can access and manipulate them in an efficient manner. At the same time, the database system makes sure that the data are not lost in case of a system crash, secures the data from unauthorized accesses and guarantees that the data will remain consistent when multiple users are handling them simultaneously [37].

The early history of database systems goes side-by-side with the progress of the storage mediums: During the 1950s, magnetic tapes were used for data-storage and computer users could input data from punched card decks. These early systems were used by companies in order to automate simple business processes such as salary raises. However, these systems were characterized by severe technical limitations, since the magnetic tapes could only be read sequentially and the data often were too large to fit in the available memory [37]. During the late 1960s, hard disks began being widely used for data storage [37]. Hard disks supported direct access to data, a technical capability that gave rise to new data models. Network and hierarchical models were developed during that time [37].

In 1970 Codd published a seminal paper [13] in which he defined the so-called *relational model*. In the relational model, *tables* are used to describe both data and the relationships between them. Tables consist of multiple *columns* and contain multiple *rows*. Each table represents entities of a particular type, each row of the table represents one particular instance of this entity and each column represents one attribute of this entity [37]. For example, we could have an `EMPLOYEES` table, with columns `employee_id`, `first_name`, `last_name`, `department_id` and `salary`. In this table, each row would represent one employee.

In order to refer to a specific row in a table, we need a way to uniquely identify it. Attribute values are used to achieve this. In more detail, a *super-key* is a set of one or more attributes that allow us to identify a row inside a table. A super-key that does not have a proper subset that is also a super-key is called a *candidate key*. The candidate key that the database designer chooses to use to identify rows in a table is called a *primary key*. In the previous example, the `employee_id` column would be the primary key of the `EMPLOYEES` table since it uniquely identifies each employee and, by extension, every row.

The relational model also captures relationships between entities. This is also achieved with the use of columns. In order to describe the connection of a row that belongs to table A to a row that belongs to table B, we use a set of attributes of the first row that uniquely identifies the second row.

This set of attributes is called *foreign key* table A referencing table B. The aforementioned tables may be the same table. In the previous example, the `department_id` column would be a foreign key from the table `EMPLOYEES` referencing the table `DEPARTMENT`.

A database based on the relational model is called a relational database and the system used to manage a relational database is called a *relational database-management system* (RDBMS). Decades of research in the field of relational databases have enabled the scientific community to acquire a vast amount of knowledge regarding how to optimize various aspects of a RDBMS. Right now, relational databases are the most widely used technology in the field of data management, with numerous massive organizations utilizing them to manage their data. Most RDBMSs use SQL to access and manipulate data, which is the standard relational database language [37].

2.2 Graph databases

A *graph database* is a database that specializes in storing and manipulating graphs. A graph is a collection of *nodes* and *edges*. Nodes store the data objects and edges represent the relationship between these nodes [2]. Both nodes and edges can have attribute fields that store extra information about the data. Instead of exposing tables, a graph database exposes nodes and edges [3].

A categorization of graph databases according to the way they are built separates them in *native* and *hybrid* graph databases. Native graph databases implement a graph data model in the storage layer, while hybrid graph databases only expose a graph data model to the users but utilize a different implementation internally [3]. We will dig deeper in the trade-offs behind these two approaches in §2.8.

Graph databases provide the users with the capability to execute traversal queries that correspond to paths in the queried graph. They also enable the user to run graph algorithms such as Breadth-First Search, PageRank [10] and Bellman-Ford in order to gain deeper insights [4].

Graphs appear in several aspects of computer science and help engineers solve a considerable number of significant computing problems. Consequently, graph databases are used in a broad spectrum of applications. Typical applications of graph databases in the real world include fraud detection in the financial industry, route planning in the logistics industry and recommendation engines in the social media industry. According to Gartner, a leading technology consulting firm, graph technologies were one of the top 10 data and analytics trends in 2021 [31]. Inkwood Research also forecasts that the global market for graph databases will reach $4.6 billion by 2027, growing at an astonishing 21.7% rate from 2019 [34].

2.3 Graph processing frameworks

A number of graph processing frameworks has been developed to facilitate the execution of graph algorithms and the extraction of insights from graph data. In contrast to graph databases, these frameworks do not support data storage. Instead, the user has to set up a data loading pipeline from a data source (e.g., HDFS tables, HBase tables, Hive tables) and create a new graph structure to work on.

Processing large-scale graphs in an efficient manner poses many technical challenges. The most significant challenge is that graph algorithms have irregular memory access patterns that do not allow

systems to utilize locality [27]. Executing graph algorithms in a distributed setting amplifies this problem and raises issues related to fault-tolerance [27].

Many single-machine systems exist for the purpose of executing graph analytics over large graph datasets. Prominent examples of single-machine graph analytics frameworks include Ligra [36], GraphMat [39] and GraphPhi [32]. However, these frameworks do not scale for datasets that do not fit in-memory. In order to support even larger datasets, disk-based approaches have been developed. GraphChi [25] and X-Stream [35] are notable examples of graph processing systems that rely on secondary storage to process graphs with billions of edges using a single machine.

At the same time, many distributed systems have been developed to accommodate graph processing for graph datasets that cannot fit in the memory of a single machine. In 2010, Google published Pregel [27], a seminal paper that described the implementation of a system that performs efficient computations over graphs with billions of vertices. Pregel proposed a vertex-centric programming paradigm for graph processing.

According to this approach, a graph algorithm is executed in iterative rounds that are called ”supersteps”. At the start of each superstep, every vertex receives messages from the vertices with which it is connected. Next, every vertex performs a computation based on the messages it received at the beginning of the computation. After its computation is done, each vertex can send messages to other vertices that are connected with it. Finally, every vertex decides if it wants to ”sleep”. Vertices that sleep are woken up only if they receive a message. The algorithm terminates when all vertices are asleep [5]. The vertex-centric approach has many similarities with the classical Batch Synchronous Parallel (BSP) model [11]. When using this programming paradigm, the complexity of distributed systems is abstracted away from the user, in a manner similar to the MapReduce [14] programming model [16].

When the user designs a graph algorithm that will be executed in Pregel, he/she has to ”think like a vertex” and define a function that will define how each vertex will behave during each superstep. To put it in another way, the developer writes a user-defined function that will read the messages received by a vertex, perform a computation and then generate messages to send to some of the neighbors of the vertex. The vertex-centric model may seem limiting at first. However, it comes with many advantages, since it allows the programmer to develop algorithms that are guaranteed to be decentralized and scalable without dealing with the complexity of concurrent and distributed systems. In more detail, each vertex performs independent computations based on its own data. As a result the algorithm can exploit the benefits of parallelization. At the same time, the algorithms developed under the vertex-centric approach are simple and intuitive. [5].

To this day, several distributed graph processing systems have been developed. Apache Giraph [12] is Hadoop-based open-source implementation of Pregel [27]. Similarly, GraphX [19] is a Spark-based open-source implementation of Pregel [27]. GraphLab [26] is a distributed framework that is widely considered one of the fastest alternatives [22]. PGX.D [22] is a commercial distributed graph processing system that manages to perform better than a single-machine when running in a cluster with a small number of machines [22]. It achieves that by utilizing techniques to reduce network traffic and to balance the workload between machines.

2.4 Graph data models

The *data model* of a database is a set of concepts that enable us to describe the data, the relationships between them, their semantics and the constraints that hold true for them [37]. The two most widespread data models for graphs are property graphs and RDF graphs.

2.4.1 Property graphs

A property graph is a directed graph in which both vertices and edges can have an unlimited number of properties [42]. Additionally, labels can be assigned to vertices and edges in order to allow differentiation between various types of objects and relationships within the graph [42].

2.4.2 RDF graphs

The Resource Description Framework (RDF) is a data model that has been initially created as a way to represent information in the semantic web [30]. In RDF, data are represented as a collection of (subject, predicate, object) triples, also known as spo triplets. The combination of all the RDF triples can be treated as a graph [29]. RDF is a flexible data model in the sense that it enables building the schema that represents the data gradually, using a so-called "pay-as-you-go" dataspace paradigm [30]. This paradigm is contrary to the classical schema-first design used in a classical data model representation. The standard language to query RDF graphs is SPARQL [29].

2.5 Graph workloads

Analogous to the transactional/analytical characterization of workloads in relational databases, we also distinguish between two different types of workload for graph processing. Specifically, we separate graph workloads into *graph queries* and *graph algorithms*. Graph queries refer to low-latency graph traversal and pattern-matching [42]. These requests are only relevant to a small local region of the graph [42]. Finding the shortest path between two vertices is a representative example of a graph query workload. Graph algorithms include iterative, long-running processes that execute over the whole graph [42]. Examples of graph algorithm workloads include Pagerank and Bellman-Ford algorithms.

With the ever-growing adaptation of machine learning techniques across every industry, a rising graph processing workload is related to the computation of graph or vector embedding so that they can be used during the training of Graph Neural Networks (GNNs) [42]. This type of workload is called *graph ML* and it is considered by many as a special case of the graph algorithms workload.

2.6 Graph query languages

A *query language* is a language in which a database user expresses requests to the database [37]. Query languages tend to be high-level languages if we compare them to the abstraction level used by programming languages. They can be categorized in procedural and non-procedural languages. In procedural languages, the query guides the database regarding the steps it should follow to generate the expected results [37]. In non-procedural languages, the query just describes the intended outcome

of the operation without specifying particular steps and the decisions related to the exact execution are being delegated to the database [37].

2.6.1 SPARQL

SPARQL [1] is a declarative query language and protocol designed for querying and manipulating data stored in Resource Description Framework (RDF) format. SPARQL allows users to express queries that retrieve information from RDF datasets by specifying patterns to match against RDF graphs, enabling the extraction of relevant data and relationships. SPARQL is an official recommendation of the W3C Consortium for handling RDF data [21]. SPARQL queries are formed by a set of triples that are analogous to RDF (subject, property, object) triples, with the difference that the elements of the triple might also be variables [1].

2.6.2 Cypher

Cypher [18] is a graph query language that originated from the neo4j graph database. Cypher is based on the property graph data model. It allows the user to query and modify data, along with defining schemas. It also offers formal semantics for its core constructs, enabling query equivalence proofs and correctness proofs for query optimizations [18]. In Cypher queries, the user expresses a pattern visually with the use of "ASCII art" [18], for example (a)-[r]->(b). The query is processed linearly from the beginning to the end of the query text, following the "path" defined by the pattern. Generally, the language philosophy is purposely similar to SQL in order to feel familiar to SQL users that try to handle graph data with Cypher [18].

2.6.3 PGQL

PGQL [44] is a graph query language that was born in Oracle Labs. It is based on the property graph data model. Just like Cypher, PGQL queries heavily utilize pattern-matching to express graph-specific concepts. Moreover, PGQL not only follows the philosophy of SQL, but it also adopts SQL-like syntax and query functionality in order to make it easier for SQL users to transition to PGQL [44]. To be more precise, PGQL includes GROUP BY, MIN/MAX/AVG/SUM, ORDER BY, LIMIT, UNION and NOT EXISTS operators. Additionally, PGQL queries return their results in tabular format, a feature that allows them to be nested inside SQL queries [44]. This empowers the user to compose more complex queries and paves the way for PGQL's integration to existing databases [44].

2.6.4 SQL/PGQ

SQL/PGQ (Property Graph Queries) is an extension of SQL that aims to introduce new capabilities regarding accessing property graphs. Essentially, SQL/PGQ makes it possible for SQL users to handle tabular data as graphs [45].

SQL/PGQ is part of SQL:2023, which is the most recent SQL standard at the time of writing. SQL/PGQ is the first property graph language that got endorsed by a standards organization (ISO) [45]. This is a positive step that encourages graph database adoption by the data science community, since the current fragmented state of graph query languages slows down the growth of the field [45].

The standardization of graph query languages is an ongoing effort that involves both experts from academia and stakeholders from the industry.

2.6.5 GQL

GQL is a new graph query language that is being developed by the same committee that drove the development of SQL/PGQ and will be a standalone language for handling property graphs [17]. GQL brings together different elements from existing approaches, but at the same time proposes new features. Compared to SQL/PGQ, GQL will provide create, insert, update and delete operation. It will also treat graphs as first-class citizens by introducing graph-only data types such as nodes, edges and paths [6]. GQL is expected to be officially published by May 2024 [6].

2.7 Regular path queries

Graph queries can express both fixed-length path patterns and variable-length path patterns. A special case of variable-length path patterns that significantly improves the expressiveness of graph query languages is the case of *regular path queries* (RPQs) [15]. RPQs are queries that retrieve paths between two vertices such that the traversed edges match a regular expression [15]. The evaluation of Regular Path Queries (RPQ) typically involves a many-to-many search [40]. Starting from a set of initial vertices, the goal is to reach a set of target vertices. Unidirectional searches initiate from one end of the path, progressively matching adjacent edges and vertices until they reach a target vertex or exhaust all exploration possibilities [40].

2.8 Performing graph analytics inside a relational database

When graph analytics initially surfaced as a promising data science trend, many new specialized graph processing systems emerged. The fact that graph analytics workloads involve iterative computations, combined with the perceived challenge of expressing graph algorithms in SQL, led to the belief that graph analytics workloads are significantly different than traditional OLAP workloads [24]. Because of that, the use of a traditional RDBMS as a platform for graph analytics tasks did not receive much attention [16]. Nevertheless, the fact that relational databases are the cornerstone of data management systems motivated both academics and industry practitioners to question whether existing relational databases could be used to efficiently handle graph processing tasks.

Without a doubt, a relational database that is capable of serving graph processing workloads would be convenient in many real-life scenarios, since nowadays most enterprises use a RDBMS as the center of their data management infrastructure. In many cases, deploying a graph database means transforming data from a tabular format to a graph format and loading them in the new graph database [24]. This task is often time-consuming, prone to errors and causes an increase in storage costs, since the original data are now duplicated. Moreover, because the data are stored in two different locations, the organization has to setup ETL processes to keep the graph database synchronized with updates and inserts that take place in the RDBMS [28]. This is a huge overhead for an enterprise, since it substantially increases complexity and IT administrative costs [16]. In addition to this, most users are accustomed to handling their data with a RDBMS and the deployment of a specialized system forces

them to learn how to use the new system. Finally, it is often the case that using a graph database is not possible due to legal limitations. When handling sensitive data like financial or medical information, it is essential to adhere to numerous regulations. These regulations may restrict the use of external engines that lack the required certifications [28].

However, convenience is not the only argument in favor of using a relational database to perform graph processing tasks. The data management community has extensively studied relational databases and has acquired vast knowledge around them. Consequently, developing a graph processing system that is based on a relation database enables it to benefit from the outcomes that have emerged after decades of research. Transactions, fault tolerance, concurrency, integrity constraints, security guarantees and powerful query optimizers are some of the most notable features that existing relational databases offer [24][28]. These features are immensely valuable to enterprise users who are in charge of building business-critical applications. Graph databases that are built on top of a relational database can utilize these capabilities immediately. On the other hand, graph databases that are built from the ground up are obligated to implement them from scratch in order to remain competitive in an enterprise setting. Naturally, this comes with a considerable engineering effort.

Furthermore, it is worth mentioning the trade-offs between using a RDBMS-based graph processing solution versus an in-memory graph processing framework. Although in-memory graph processing systems achieve better performance for datasets of moderate data size, they fail to process larger datasets, since intermediate results cannot fit in the memory of a single machine [16]. To overcome this scalability issue, these solutions have to run in a distributed setting using multiple machines. Research results show that a cluster of 32 nodes performs graph processing an order of magnitude faster than a single-node RDBMS [16]. Even though this is undoubtedly an important difference in performance, if we take into account all the expenses, including hardware amortization, power, and administration costs, it is possible that performing graph-processing inside a RDBMS is more cost-effective. In other words, despite the fact that distributed graph processing frameworks outperform RDBMS-based graph analytics, the latter may be the best choice for scenarios where performance is not the priority, since the scale-out approach might be too costly.

The extensive adoption of relational databases, combined with the popularity that graph processing has gained in recent years, turned RDBMS-based graph analytics into an approach that has attracted the attention of both industry and academia.

From a research perspective, numerous works related to performing graph analytics inside a relational database have been published. Grail [16] is a system that stores vertices and edges as tables and provides a programmer-friendly interface to query these graphs in a vertex-centric manner. A translator then maps these graph queries to equivalent SQL queries and an optimizer improves the produced SQL queries. The improved SQL is then executed by a relational database. Likewise, Vertexica [24] is a graph analytics tool that receives Pregel-like vertex-centric queries as input and makes use of UDFs to convert them to standard SQL queries to be executed by a RDBMS. SQLGraph [38] translated graph queries to SQL and uses a relational schema to store adjacency information and JSON to store the vertex/edge attributes. GRFusion [20] is a system that modifies the internals of a relational database in order to integrate both the relational and the graph data model within a unified query engine. To achieve this, it defines a graph using views over tables and then materializes an in-memory graph index that mirrors the topology of the graph. The nodes and the edges of the graph index contain

pointers to the corresponding relational data, empowering us to efficiently perform graph operations over the in-memory index and only retrieve relational data when we need them. GraphGen [46] is a tool that allows scalable extraction of large-scale graphs that are stored in tabular form in a relational database. GRainDB [23] enhances DuckDB [33] with graph processing capabilities by implementing fast join mechanisms that better serve graph workloads.

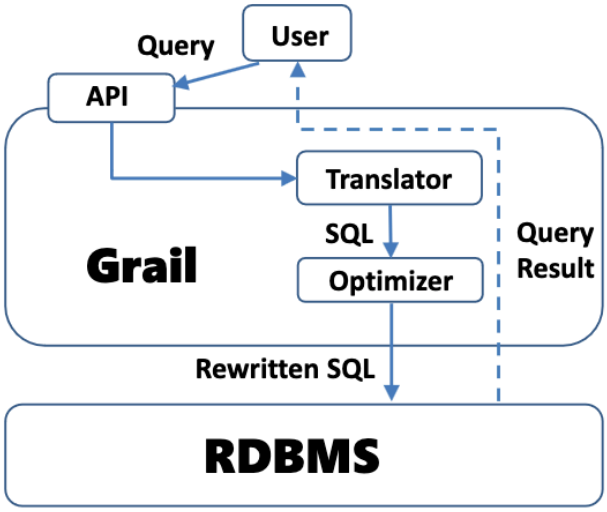


Figure 2.1: Architecture of Grail [16]

Chapter 3

Background

3.1 Data model

In this section we will describe the data model that is used to define property graphs in a relational schema. Let's suppose that we have three tables/views T_s, T_e, T_d such that T_e declares some foreign key relations to both T_s and T_d. The triplet <T_s, T_e, T_d> can be used to define connections in a property graph. Specifically:

- each row in T_s and T_d can be interpreted as a vertex
- each row in T_e can be interpreted as a directed edge that connects the two vertices that correspond to the referenced rows in T_s and T_d
- columns of T_e can be interpreted as edge properties and columns of T_s and T_d can be interpreted as vertex properties

T_s, T_e and T_d are called *graph element tables*. Specifically, T_s and T_d are *vertex tables* and T_e is an *edge table*. Every edge table defines a directed association between the two vertex tables, referred to as the source table and the destination table. A table/view can appear more than once in the triplet.

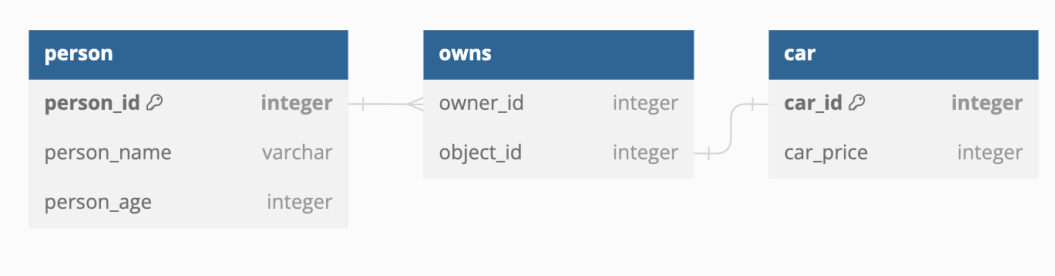


Figure 3.1: Example schema of a source-edge-destination triplet

A collection of such triplets can be used to define a *property graph* in a relational database. Property graphs are SQL-schema objects that represent graphs from existing database objects. These database objects are also called *underlying objects*. Graph element tables can be characterized by labels that expose columns of the underlying objects as properties.

A property graph does not store data itself. The data reside in the underlying objects that were used to define the graph. In that sense, a property graph can be thought of as a view that provides an abstraction for graph data that are stored in a relational database.

Property graphs can be queried with the use of SQL extensions for graph processing. Modifications to the underlying objects that are used to define the property graph are instantly visible to the property graph queries.

3.2 Execution of graph algorithms in a relational database

Executing graph algorithms over property graphs allows users to gain insights from graph data that are stored inside a relational database. Graph algorithms perform graph analytics tasks that take into account both the structure of the graph and the properties of its vertices and/or edges.

Graph algorithms are invoked as part of a graph query. In order to perform the necessary graph processing computations in the context of a relational database, the implementation dynamically issues large SQL queries. This leads to an SQL-based execution of the graph algorithm. For example, if the algorithm needs to find the neighbours of a vertex, it issues a `SELECT` query to retrieve them after joining the appropriate tables (see §3.3.3). This design enables the utilization of established query optimization techniques that are already implemented by the RDBMS. It's worth mentioning that the vast majority of the computational time is dedicated to the execution of these queries.

To make its results accessible, a graph algorithm creates a new property and uses it to store its outputs. These properties are called *output properties*. As an example, PageRank [10] creates an output property for every vertex to store the rank it calculated for it.

Because graph algorithms are invoked through queries, their output must be available through SQL. Moreover, graph algorithms generate huge volumes of intermediate results. These results must be efficiently stored and retrieved by the SQL queries that consist the main part of the algorithm. To address these requirements, tables are created in which both the intermediate and the final results are stored. Using tables provides a natural way to retrieve the output of the algorithm in a SQL-based manner and enables the SQL queries to conveniently manage the algorithm's data.

The aforementioned tables are created during the compilation of the graph query that invoked the graph algorithm. Specifically, a table per graph element table is created. For example, if the graph consists of two vertex table and the query's results expose an output property, two tables will be created. Each of these tables is called a *side table* and the associated graph element table is called a *primary table*. The primary tables hold the graph's original properties and the side tables store additional properties calculated by the graph algorithm. Each side table has columns that implicitly act as a foreign key to the primary tables and columns that are used to store the algorithm's output properties. When a graph query attempts to retrieve both an output property and an input property, a join between the primary table and the side table is performed.

3.3 Evaluation of unbounded recursive path queries

3.3.1 Design overview

An unbounded regular path query (RPQ) is a regular path query that attempts to retrieve paths that match a regular expression that involves unlimited hops. Equivalently, they are graph queries that attempt to match a pattern of the form `(a)-[e]->*(b)`, where `*` refers to Kleene's star and both `a` and `b` can be complex graph queries.

To process an unbounded RPQ, the design splits it in three parts: the *prefix*, the *suffix* and the *recursive part*. This split takes place during query compilation, after the query has been parsed and semantically analyzed. In the previous query, `a` is the prefix, and `b` is the suffix. Both the prefix and the suffix can be evaluated using the existing capabilities of the system. The recursive part is evaluated by a specialized kernel that receives a vertex a as input and finds every vertex b that can be reached from vertex a.

$$\underbrace{(v_1) \rightarrow (a)}_{\text{prefix}} \underbrace{\longrightarrow *}_{\text{recursive part}} \underbrace{(b) \rightarrow (v_2)}_{\text{suffix}}$$

Figure 3.2: Unbounded RPQs are splitted in three parts: prefix, suffix and recursive part

During the execution:

- one result of the prefix is given as input to the recursive part of the query
- the recursive part of the query is being evaluated through the execution of a graph algorithm whose outputs are stored in a side table
- the suffix reads the results of the recursive part from the side tables in order to use them as input
- the prefix and the suffix are joined

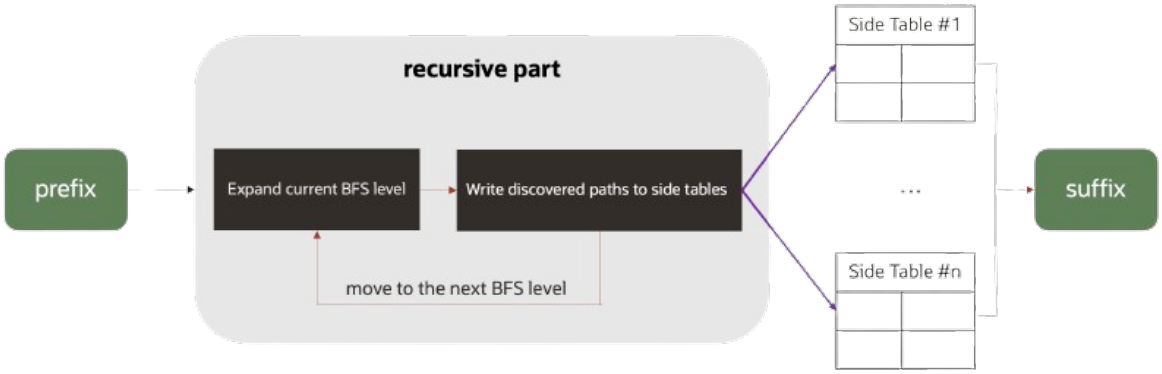


Figure 3.3: Execution flow of the unbounded RPQ evaluation

3.3.2 Algorithm

Description

The recursive part of the query receives as input a vertex s at which the prefix ends. The algorithm is tasked with finding shortest paths to every vertex that is reachable from s. Additionally, the algorithm should keep the required information to support queries and aggregations over the vertices of each path. To achieves its task, it executes a **breadth-first search** that traverses the graph starting from s.

Figure 3.4: Kernel receives a vertex s and returns paths to every vertex that is reachable by s.

As a reminder, the most common implementation of the breadth-first search algorithm uses:

- a queue of vertices to store the frontier of the traversal
- a set to keep track of the visited vertices

The data structures employed to implement the specialized kernel are not conventional data structures that operate in-memory. These data structures are not sufficient for an implementation that is tasked with processing massive graph datasets, because the intermediate state of the algorithm may not fit in-memory. To address this issue, specialized *disk-spilling data structures* are used. These data structures flush part of their content to the disk, allowing the management of larger data volumes. This design takes advantage of pre-existing data structures that are implemented internally by the relational database. Specifically, it uses:

- a disk-spilling hash-map
- a disk-spilling buffer that is used as a queue

Disk-spilling hash-map

A hash-map is used instead of a set in order to keep track of the number of times a vertex has been reached. This information is necessary to extend the breadth-first search so that it can efficiently answer "top-k" queries such as "find the 5 shortest paths". The disk-spilling hash-map stores its entries in different partitions. When the system's available memory is limited, it flushes one or more of its partitions to the disk. The partitioning and flushing policy of the disk-spilling hash-map is beyond the scope of this thesis.

For the disk-spilling hash-map to be used efficiently, look-up and insert operations should not be randomly ordered. Rather, they should be grouped by partition, in order to reduce disk I/O by ensuring that the accessed partition currently lies in-memory. To achieve that, the breadth-first search algorithm is slightly modified to enforce partition-friendly data access patterns. These modifications are not relevant to the work presented in this thesis.

Disk-spilling buffer

Due to technical limitations, the disk-spilling buffer that is used as a queue for the breadth-first search has two phases, a "write-only" phase and a "read-only" phase. During the "write-only" phase, writes are allowed and reads are prohibited, while during the "read-only" phase, reads are allowed and writes are prohibited. The initial phase is the "write-only". After the user decides to switch to the "read-only" phase, returning to the "write-only" phase requires erasing the contents of the buffer. As a consequence, using a single disk-spilling buffer as a queue is not possible, since the algorithm

iteratively reads a vertex that belongs to the current BFS level from the top of the queue and then writes its neighbours to the end of the queue.

To overcome this problem, the design uses two disk-spilling buffers:

- one ”read-only” disk-spilling buffer that contains the vertices of the current BFS level
- one ”write-only” disk-spilling buffer that is being filled with the vertices of the next BFS level

When the end of a breadth-first search level is reached:

- the next BFS level will become the current BFS level, so its disk-spilling buffer is switched to ”read-only”
- the BFS level that just ended is no longer useful, so its disk-spilling buffer can be erased and switched to ”write-only” in order to accommodate the next BFS level

3.3.3 Neighbor expansion

According to the design that was described in §3.2, the algorithm is executed via SQL queries that are issued dynamically by the algorithm's kernel. In more detail, to find the next level of the breadth-first search, the algorithm has to discover the neighbours of each vertex in the current level by issuing appropriate `SELECT` statements.

In particular, let's assume a vertex v that belongs to a vertex table $VT1$. To retrieve the neighbours of v, for every triplet $< VT1, E, VT2 >$ of the property graph for which $VT1$ is a source vertex table, a `SELECT` query over a source-edge-destination join is issued.

```
1 SELECT dst.id
2 FROM vt1 src, et12 e, vt2 dst
3 WHERE src.id = :x
4   AND src.join_key = e.src_key
5   AND dst.join_key = e.dst_key;
```

Figure 3.5: Template for finding the neighbors of a vertex

The results of the `SELECT` queries are then processed inside the kernel. Specifically, for every neighbour that was found, a lookup in the hash-map is performed to determine whether the neighbour has been already visited. If the neighbour has not been visited, it is added to the queue.

3.3.4 Output

The algorithm kernel writes its output to side tables, from where it will be read from the SQL that evaluates the suffix of the RPQ. To write its output to the side tables, the kernel issues `INSERT` statements. The results are written in one-row-per-step form. This means that for every path that reaches vertex d, the algorithm inserts n rows to the side tables, where n is the length of that path. As a consequence, the total number of inserted rows is equal to the sum of the number of hops for each path that reaches vertex d, for every vertex v that was reached during the algorithm's execution. This means that the system has to handle an insert-heavy workload.

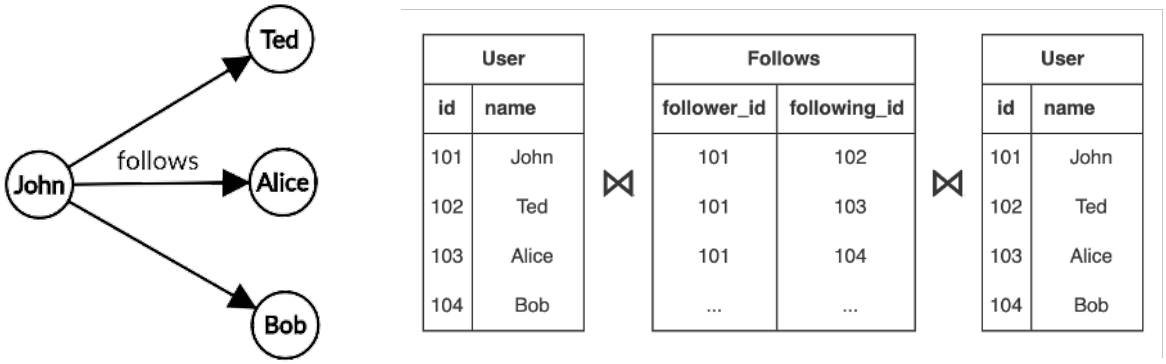


Figure 3.6: Neighbour expansion for a given vertex

It is crucial to remember that the goal of the specialized kernel is not to just find vertices that are reachable from a given vertex, but to also retrieve and manipulate the paths that lead to those reachable vertices. Writing the results in one-row-per-step form facilitates the calculation of aggregations over the path's vertices by utilizing the well-optimized aggregation capabilities that the database provides. For example, we can calculate the maximum value of a property across every vertex of the path.

The output is written per destination vertex. This implies that if a path's final vertex is vertex d, the algorithm kernel will insert the output rows to the side table that is associated with the vertex table to which vertex d belongs. The rows will have the primary key value of vertex d.

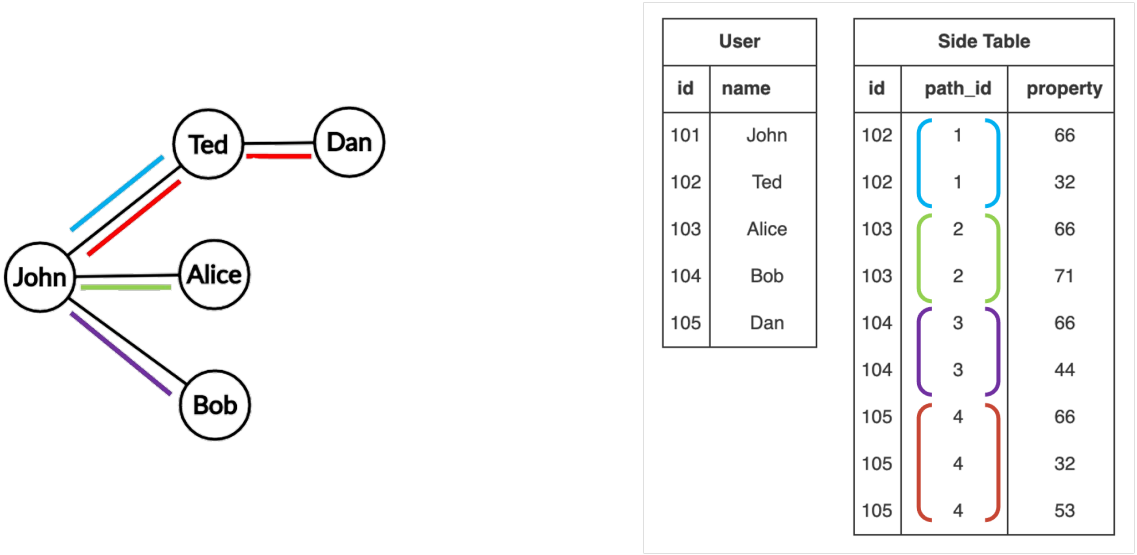


Figure 3.7: Output of unbounded RPQ

3.3.5 Heterogeneous graphs

The data model that was described in 3.1 allows property graphs with multiple vertex/edge tables in their definition. These graphs are called *heterogeneous* graphs. For example, the graph that is visualized in 3.8 involves three vertex tables (with `Person`, `EU_City` and `US_City` as underlying objects) and four edge tables (with `EU_Born`, `EU_Lives`, `US_Born` and `US_Lives` as underlying objects). In this section we briefly describe some details related to supporting heterogeneous graphs.

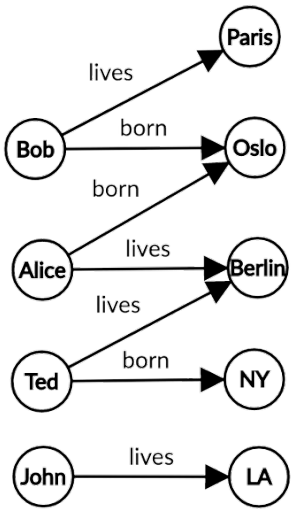


Figure 3.8: Heterogeneous graph

To retrieve the neighbors of a vertex, we have to take into account the connections that are defined by multiple edge tables. In more detail, when we want to retrieve the neighbours of a given vertex v that belongs to a vertex table VT, we have to consider every table ET that has VT as its source vertex table. For each edge table ET, we execute one `SELECT` statement according to the template presented in 3.5. Vertex tables and edges tables are processed in sequence. As an example, in 3.9 we visualize the `JOIN` operations that must be performed in order to retrieve the neighbours of "Ted" from 3.8.

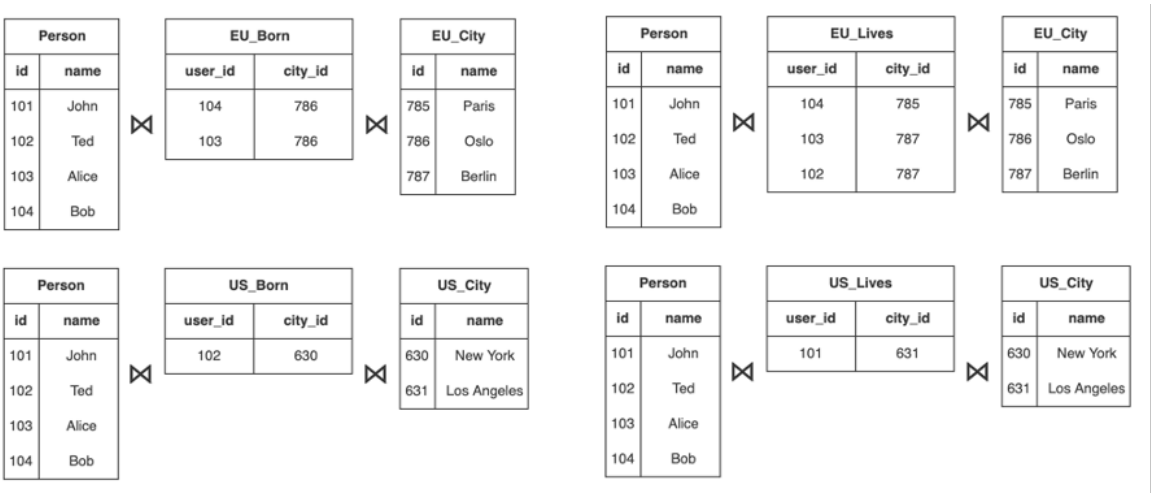


Figure 3.9: Multiple joins for neighbor retrieval in heterogeneous graphs

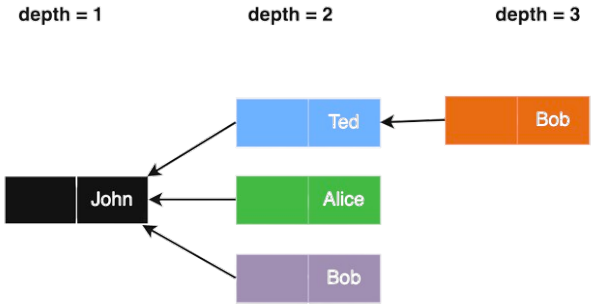
In heterogeneous graph, the vertices of the graph may be defined in different vertex tables. Since different tables have different schemas, vertices in the same graph might be identified by different primary keys. For example, $VT1$ might use two numbers as its primary keys but $VT2$ might use a string. This is problematic because the primary keys of the vertices are used as keys for the hash-map that keeps track of the vertices that have been visited by the graph traversal. This issue is solved by using a distinct hash-map per vertex table. When a look-up/insert is performed, the operation is directed to the correct hash-map.

3.3.6 Path storage

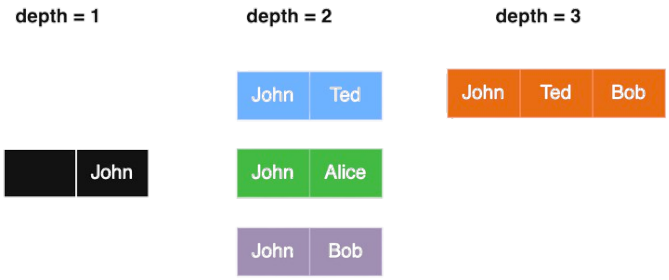
An approach to storing discovered paths is *path sharing*. According to this approach, when a new vertex v is discovered during the graph traversal, v will be added to the queue together with a pointer to the previous vertex in the path towards v. This means that subpath prefixes are shared across all the paths that extend them. Under this design, reconstructing a path requires a lot of random memory accesses, since it involves following the pointers that lead from the end of the path to the starting vertex. While these accesses are acceptable in an in-memory scenario, they come with an intolerable performance overhead for out-of-core execution, since disk storage does not support efficient disk I/O. For this reason, the *path copy* design is chosen instead. Under this design, when a a new vertex v is discovered during the graph traversal, the whole path to v will be added to the queue.

With the prefix sharing design, it is not possible to only use two disk-spilling buffers, since each queue uses pointers to previous BFS levels to define paths. On the contrary, the prefix copy design enables the use two disk-spilling buffers, since every queue stores self-contained paths. Of course, this design decision comes with a space-time trade-off, as the prefix copy approach needs more memory. Because disk storage is used in this design, the extra memory requirement is considered acceptable.

In 3.10 the two approaches are visualized for the the graph that was used in 3.6.



(a) Path sharing



(b) Path copy

Figure 3.10: Path sharing versus path copy

Chapter 4

Optimizations

Even though the algorithm described in chapter 3 manages to evaluate unbounded RPQs, its performance is not sufficient. Performance analysis suggests that the algorithm's runtime deals with SQL statements inefficiently. In this work we use profiling tools to recognize bottlenecks in the algorithm's runtime and we implement optimizations to improve its SQL-based execution.

4.1 Batching for path insertion

Batch insertion is a suitable technique when we aim to perform a large number of insertions to the database. Instead of performing each insertion independently, we combine all the insertions in a group. Batch insertion allows the database engine to process all the inserts at once. This has the potential to perform significantly better in comparison to multiple single-row insertions [37]. This is explained by the fact that there is a constant overhead per statement execution. As a consequence, when the inserts are performed separately, this overhead occurs for every statement. For a large enough volume of `INSERT` statements, the accumulated impact on performance is substantial. On the other hand, when insertions are performed in batches, the execution of the statement is initialized once for the whole batch. As a result, the cost of the initialization is amortized and can be neglected. This results in a notable increase in the number of inserts that can be executed per second. Due to its usefulness, many database drivers such as JDBC provide native support for batch insertion. This can be better understood from the pseudocode below.

During the evaluation of unbounded recursive path queries, we have to insert one row to a side table for every hop within each path generated by the query. This leads to an insert-heavy workload. Our profiling analysis indicated that inserting rows to the side tables was the major bottleneck of the algorithm. To reduce the impact of inserts, we applied insert batching.

To implement this, we allocate an in-memory buffer before we start the graph traversal. This buffer will be used to temporarily keep the path hops that will be eventually written in the side tables. We set a fixed upper limit for the buffer's size to prevent it from consuming too much memory. During the algorithm's execution, whenever we discover a new path, we do not immediately insert its hops to the side table. Instead, we store the path in-memory by adding its hops to the buffer. This allows us to postpone the execution of the respective `INSERT` statements until a large enough number of inserts has been queued.

When we decide to write the buffer's content to the side tables, we iterate over the paths in the buffer and for every path hop we generate an `INSERT` statement to write it to the appropriate side table. At the end we end up with a group of insertions that includes an `INSERT` statements for every hop of

```
begin stmt;
execute insert row_1;
end stmt;

begin stmt;
execute insert row_2;
end stmt;

...

begin stmt;
execute insert row_n;
end stmt;
```

(a) Pseudo-code for multiple separate inserts

```
begin stmt;
execute insert row_1;
execute insert row_2;
...
execute insert row_n;
end stmt;
```

(b) Pseudo-code for batch insertion

Figure 4.1: Multiple separate inserts versus inserting in batches

every path that was found in the buffer. To avoid unnecessary memory allocations, after flushing each batch, we clear and reuse the buffer in which the batch was stored.

We trigger a flush of the buffer to the side tables when:

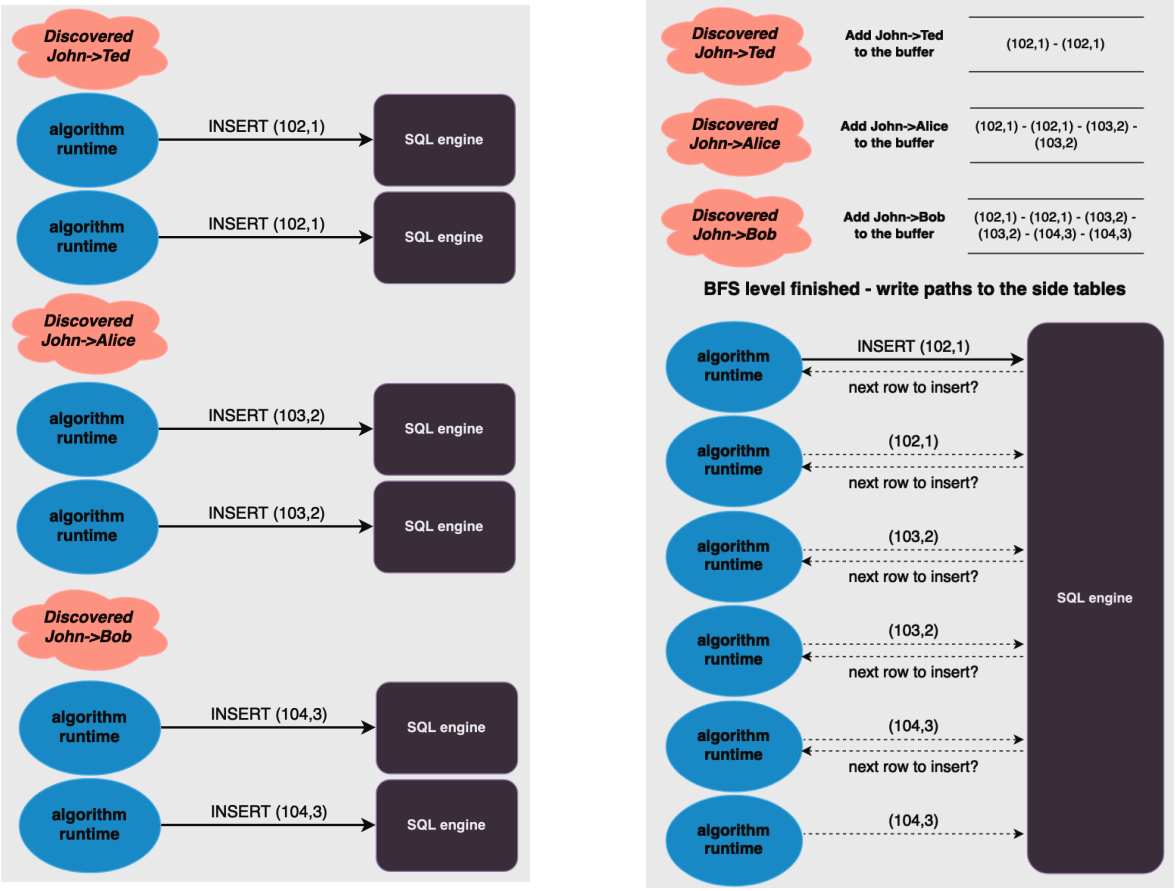
- its memory size has surpassed the fixed upper limit
- the end of a breadth-first search (BFS) level has been reached

Regarding the first flushing criterion, the maximum memory size of the buffer is required because otherwise the buffer would end up consuming excessive memory for queries that search for paths with large lengths or for queries over very dense graphs. Moreover, our experiments show that after a certain point further increasing the upper value demonstrates diminishing returns, as the performance improvement we gain is insignificant. We provide an experimental analysis on how the value of this parameter affects the execution of the algorithm in §5.5.

Regarding the second flushing criterion, we would ideally allow the buffer to store paths that were found at different BFS levels. However, this is not possible, because the buffer stores pointers to the paths and not deep copies of them. Since the paths are stored in the disk-spilling queues, these pointers become invalid after the end of each BFS level, when the queues are cleared. We prefer to store pointers to the paths in the buffer because these pointers require less memory than the paths they point to. This lets us include more paths in a buffer without changing the buffer's memory limit. By extension, due to this design decision, we are obligated to flush the buffer at the end of each BFS level.

The events we use to decide when the buffer flushing will be triggered ensure that all paths belonging to the same batch were discovered at the same BFS level. equal to the depth of the BFS level at which they were discovered.

Note that paths that belong to the same insertion batch have the same length. This is guaranteed by the criteria according to which we determine when the batch will be written to the side tables. To be more precise, because the batch is flushed at the end of each BFS level, it is ensured that every batch will contain only paths that were discovered at the same BFS depth. Because of this, storing the length of each path is redundant, as all the paths in a batch have identical length.



(a) Insertions without batching

(b) Insertions with batching

Figure 4.2: Illustration of the ”batching for path insertions” optimization

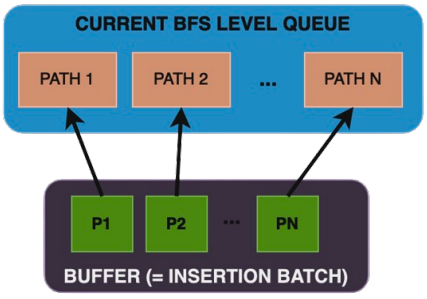


Figure 4.3: Design of the buffer used for the batch insertions

4.2 Optimizer hints

It is common that for a given query there are many equivalent ways to compute its result. The different ways to compute the result of a query are called *query execution plans*. While query execution plans calculate the same result, they do not necessarily exhibit the same performance. *Query optimization* is the phase of query processing during which the database engine selects which one of the query execution plans will be executed. In doing so, it aims to choose the most efficient execution plan by utilizing available information about the data in order to make a prediction [37].

Often the query writer has some insight regarding which execution plan will perform better. This

information can be passed to the optimizer through the use of *optimizer hints*. Optimizer hints are directives provided by a query to make suggestions to the query optimizer regarding which execution plan should be preferred. The optimizer may or may not follow the hint.

To further optimize insert operations to the side tables, we include optimizer hints [7] to the queries we issue. To be more specific, we guide the optimizer to insert the rows in the side tables using *direct-path insert*. Direct-path insert is an inserting method that skips parts of the conventional insert operation, in order to facilitate efficient inserting. In more detail:

- Direct-path inserts do not attempt to reuse free-space in the table into which the rows will be inserted. Rather, it allocates space at the end of the table and it appends the rows.
- Direct-path insert bypasses buffer cache. Instead it writes directly to the end of the datafile.
- Integrity constraints are not checked and trigger processing is not executed. However, this is also not an issue for us, since we perform the inserts from inside the database and we take responsibility for the values that we will insert.
- Direct-path insert locks the table in which we will insert the rows. This is not a problem in our scenario, because we created the table we are writing to with the sole purpose of storing the results of the graph algorithm we are running and only we are reading/modifying the table.
- Direct-path does not keep any redo logs and keeps minimal undo logs. This is also not concern, because in case of failure during the execution of the graph query we will just re-execute it, since this is an analytical task.

To summarise, by using direct-path inserts in our case, we manage to avoid much of the overhead that comes with the conventional insert and achieve better performance.

4.3 Prepared statements

Query parsing is the phase of query processing during which the database engine parses the given query and converts it to an internal representation that is more useful for query optimization and query execution [37]. Query parsing produces a parse tree that will subsequently be translated to a relation algebra expression and later to an execution plan [37]. The whole process is very similar to the task performed by the parser of a compiler. Query parsing involves syntax checking, privilege validation, semantics verification and execution plan generation.

For statements that are relatively simple to execute, query parsing may be a non-negligible performance overhead, because execution plan generation is a CPU-intensive task. If a statement is repeatedly used, we can avoid regenerating the same execution plan over an over again by keeping a handle to the compiled version of the statement after compiling it for the first time. To do this, we utilize *bind variables*. In more detail, we can generate the execution plan and leave some values unspecified on purpose. The unspecified values will act as parameters that will allow us to use the stored execution plan for different queries that perform the exact same task for different data. For example, we will be able to use the same execution plan to insert different values to a specific table, just be

replacing the bind variables with the actual values we intend to insert each time. It is worth mentioning that, internally, we use a special type of bind variables that can supports re-assigning values. The pre-compiled execution plans we store are called *prepared statements* [37].
In our case, we use prepared statements for:

- the `INSERT` statements that inserts hops in the side tables of vertex tables
- the `SELECT` statements that retrieves the neighbours of a vertex during neighbour expansion

Both statements are relatively simple to execute so parsing constitutes a considerable cost in terms of performance. Furthermore, both statements are executed repeatedly, so the overhead of compilation is considerable. As a result, this optimization leads to end-to-end performance improvement by compiling each query only once and then running it with different parameter values as many times as needed [37].

Since execution plans are generated for a specific table, we have to keep one execution plan for each vertex table to which we want to insert a hop and one execution plan for each edge table from which we want to retrieve vertex connections. We do not generate the execution plans before-hand. Rather, the first time we attempt to execute a statement for a specific table, we compile the statement once and store the produced execution plan. After that, we reuse the stored execution plan for the other executions by just supplying the appropriate values for the parameters of the statement. This is made possible by the fact that the algorithm executes the exact same SQL statements over and over due to its iterative nature. Since execution plans do not consume substantial memory and the number of vertex/edge tables in relatively small, caching execution plan comes with virtually no memory overhead.

4.4 Batching for neighbor fetching

Query execution plans can be seen as a sequence of basic database operations such as selects, projects and sorting operations. Some of these operations, for example joins, receive more than one input. Query execution plans can be visualized as *operations trees*, which are trees that represent the steps required to actually execute a query, with the first steps being the leafs and the final step being the root of the tree. [37]

The naive method to evaluate the results of an execution plan is to sequentially execute each operation and *materialize* (temporarily store) its results, so that they can be given as input to another operator at a later stage. However, this process suffers from excessive disk I/O [37]. An alternative would be to concurrently evaluate multiple operations in a pipelined fashion, with the products of one operation being directly passed to the next, eliminating the need to materialize intermediate results [37]. Pipelines can be demand-driven or producer-driven.

- In *demand-driven pipelines*, each operation only evaluates the next result when it receives requests from its parent operation in the operation tree. When an operator needs input from a child operator, it makes a relevant request. Demand-driven pipelines work with lazy computations. [37]
- In *producer-driven pipelines*, operations do not wait for requests to generate results. In that sense, producer-driven pipelines work with eager computations. [37]

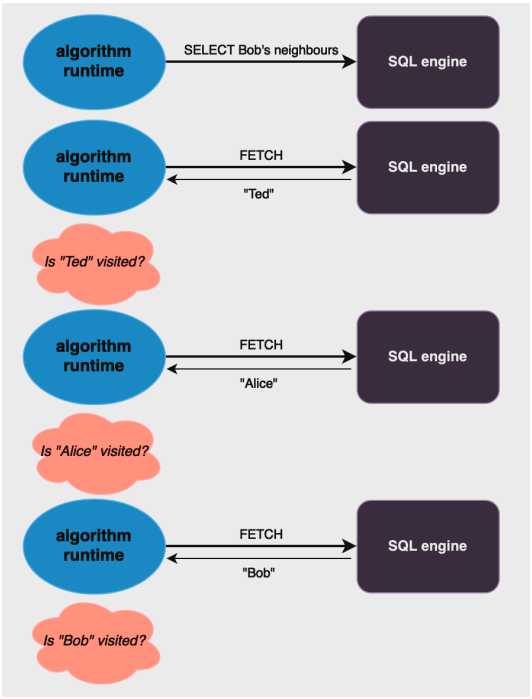
Most centralized RDBMSs prefer demand-driven pipelining [2]. In a demand-driven pipeline, every operation can be implemented as an iterator that provides an interface to its parent operation. The interface exposes the following functions:

- `open()`: initializes the operator
- `fetch()`: returns the next output row of the operation
- `close()`: signals to the iterator that no more rows will be requested

The iterator preserves the state of its execution between calls to ensure that successive next() requests yield consecutive result rows. [37]

During the neighbour expansion phase of our algorithm, each vertex at the frontier of the bread-first search issues one or more `SELECT` statements in order to query the database about its neighbours. We observed that the original version of the prototype fetched neighbours one-by-one and checked if the fetched neighbor should be added to the next BFS level.

In order to accelerate neighbour fetching, we implemented the required modifications in order to allow each vertex to fetch its neighbours in one large batch. This eliminated the overhead of constantly requesting the next available neighbour and resulted in better performance. We then iterate over the fetched neighbors to determine which of them will be added to the next BFS level.



(a) Neighbour fetching without batching

(b) Neighbour fetching with batching

Figure 4.4: Illustration of the ”batching for path insertions” optimization

Chapter 5

Experiments

5.1 Setup

5.1.1 Hardware

We evaluated the system on an Oracle Linux machine with kernel version 5.4.17. The specifications of the machine are:

- AMD EPYC 7742 Processor @ 2.30GHz, with 1 socket, 16 cores, and 32 threads
- 251GB DDR4 RAM @ 2133MHz
- 16MB L3 Cache, 512KB L2 Cache and 64KB L1 Cache with cache line size of 64B

Every system we evaluated was allowed to use at most 150GB of memory.

5.1.2 Datasets

The following datasets were used for the evaluation:

- the San Fransisco dataset
- datasets from LDBC SNB [9]

The San Francisco dataset is a grid-like graph that represents part of San Francisco's road network.

The Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) [9] is designed to evaluate graph database systems in workloads that simulate real-world social network applications. The LDBC SNB dataset includes large-scale graphs that mimic the structure of social networks, with various types of entities such as users, posts and comments. It intends to provide a standardized benchmark for evaluating the performance of graph database systems in handling such workloads.

Dataset	#vertices	#edges
SAN_FRANCISCO	59.813	149.715
LDBC_01	327.588	1.477.965
LDBC_1	3.181.724	17.256.038
LDBC_10	29.987.835	176.623.445

Table 5.1: Dataset sizes

5.1.3 Queries

We will use the following queries to perform experiments:

- $Q1$ is a simple unbounded recursive path query without filters that begins from a fixed starting vertex in SAN_FRANCISCO and explores the graph.
- $Q2$ explores a part of a LDBC SNI [9] graph that has a tree structure, since it follows a chain of replies until the chain reaches the initial post.
- $Q3$ explores a part of a LDBC SNI [9] graph that has the structure of a social network, since it represents connections between persons that know each other.

```
1  // Q1
2  SELECT n.id, m.id FROM MATCH ALL
3      (n) -[e]->{,100} (m)
4  WHERE n.id = <input>;
5
6  // Q2
7  SELECT m.id AS root_id FROM MATCH ALL
8      (n:Comment)
9          -[e:commentReplyOfComment|commentReplyOfPost]->*
10     (m:Post|Comment)
11 WHERE n.id = <input>;
12
13 // Q3
14 SELECT m.id AS friend_id FROM MATCH ALL
15     (n:Person) -[e:personKnowsPerson]->{,3} (m:Person)
16 WHERE n.id = <input>;
```

Figure 5.1: Queries used for experimental evaluation

For $Q1$, the bound is large enough so that the query explores most of the graph. In that sense, its behaviour is identical to that of an unbounded RPQ. For $Q3$, we choose a relatively small bound so that the execution time remains reasonable. However, the measurements we get from this bound are enough to reach to solid conclusions. Of course, for each graph we alter the $n.id$ in the WHERE clause so that it corresponds to an id that exist in the dataset for which we run the query.

5.2 Impact of each optimization

In figure 5.2 we visualize the compound improvement to the end-to-end performance of $Q1$ after applying each optimization. First of all, we immediately notice the impressive performance gains that occur after we apply insert batching. This significant improvement is explained by the huge volume of inserts that take place during the execution of the query (approximately 670K). Since inserting was the main bottleneck of the original system and we managed to eliminate a major part of the overhead from which every `INSERT` statement suffered, we were able to significantly accelerate query execution. This optimization alone improves end-to-end query execution time by a factor of 32. After utilizing prepared statements and batching for neighbor fetching we manage to reduce the execution time by an additional 25%. The total end-to-end performance was improved by a factor of 44.

optimization	extra improvement	end-to-end improvement
baseline	1	x2
+batching for hop insertion	x32	x32
+prepared statements	x1.14	x37.5
+batching for neighbor fetching	x1.19	x44

Table 5.2: Compound effect of optimizations

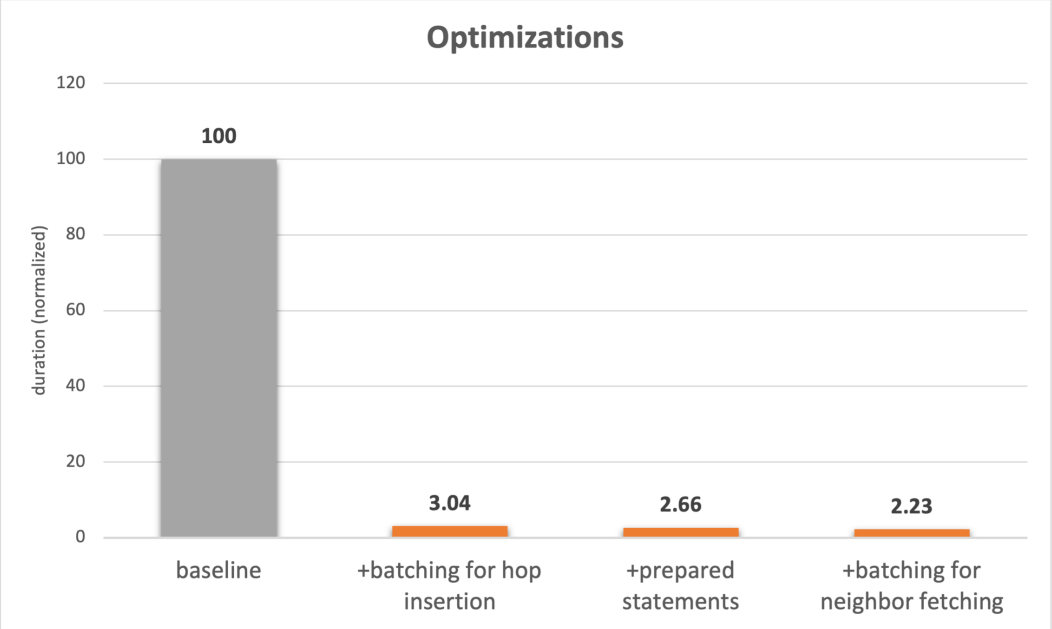


Figure 5.2: Optimizations impact

5.3 Impact of maximum path length

In this experiment, we modify the maximum path length of the path we aim to retrieve from $Q1$. We do this, to understand the system's behaviour as function of the amount of work that needs to be done. We gradually increase the value of this bound and measure the performance of both systems. It's worth noting that the workload does not increase linearly with the path length bound. The workload increase depends heavily on the structure of the part of the graph that is explored by the unbounded RPQ. The normalized results for different values of the maximum length parameter are the following:

maximum path length	optimized system	baseline	improvement	hops inserted
16	1	2	x2	1.763
32	1.31	5.12	x3	6.684
64	2.81	68.75	x24	110.717
128	16.62	900	x48	1.420.041
256	98.5	6556.25	x67	9.134.788

Table 5.3: Maximum path length experiment (normalized results)

According to our experiments, the optimized version performs better for every value of the parameter. As the length of the path we are searching for increases, the optimized version continues to

perform well. On the other hand, the original version fails to scale. To explain this behaviour, we have to take into account that when we increase the maximum path length from k to $k+1$, for every path of length $k+1$ that is retrieved, both systems are forced to perform $k+1$ more inserts, since we need one `INSERT` for every vertex of every path. In other words, larger maximum path length parameters lead to workloads that are characterized by more insertion operations. To provide an intuition regarding the rate at which the workload increases, we include the number of hops inserted to the side tables during the execution of $Q1$ for different values of the path length bound.

Given the fact that our optimizations mainly target the overhead that impacts `INSERT` performance, it is expected that the impact of our optimizations will be more noticeable for queries that search for paths with greater lengths.

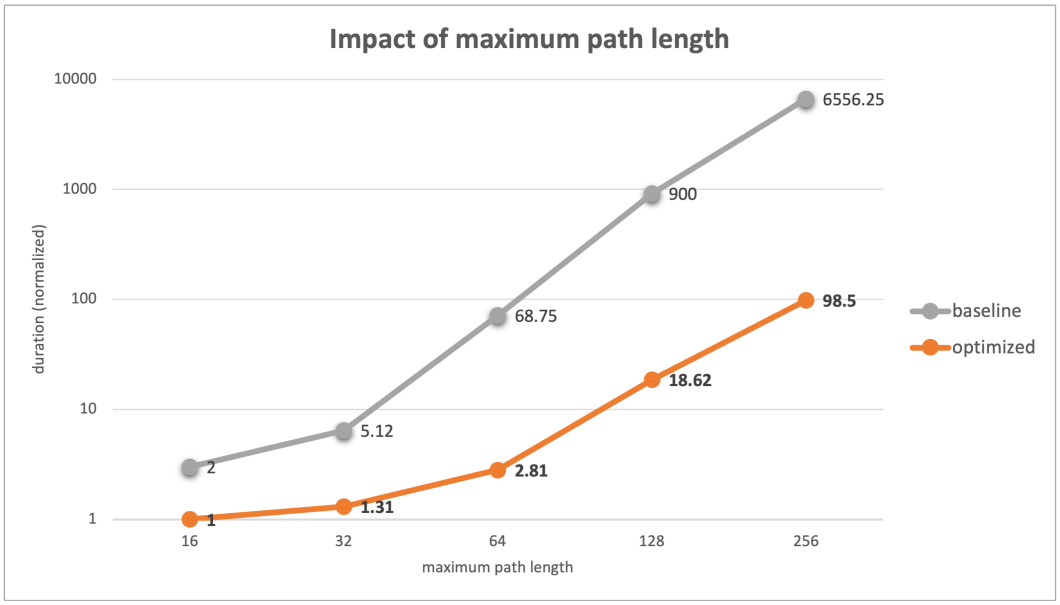


Figure 5.3: Maximum path length experiment

Moreover, our optimizations also target the SQL queries that are invoked during the neighbor expansion phase of the bread-first search algorithm. When the query searches for a path with larger length, the breadth-first search explores more levels and, by extension, more neighbor expansion phases are executed. As a consequence, path queries with greater maximum path length are benefited by these optimizations too.

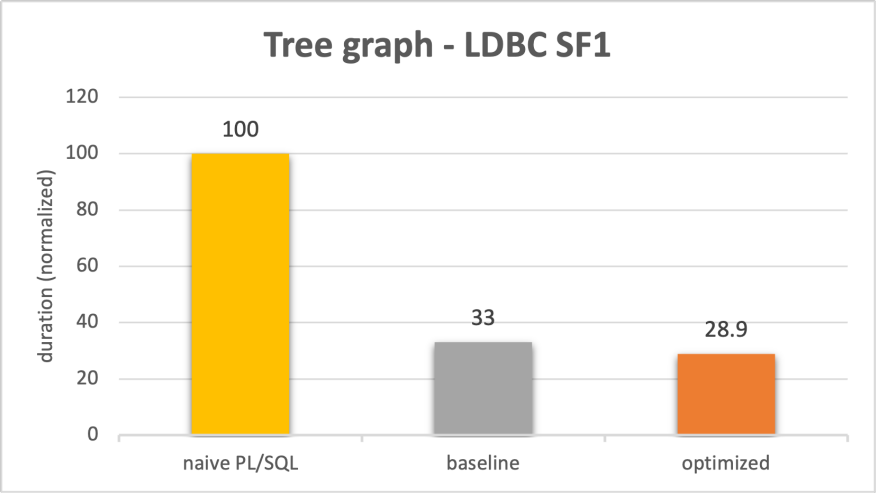
5.4 Performance comparison

We compare against a naive PL/SQL implementation of unbounded RPQs evaluation that is not integrated in the database. To be more precise, we compare against the equivalent PL/SQL program a user would have to run to generate results for the same query. This program would essentially be an implementation of the bread-first search algorithm (BFS). We perform this comparison in order to measure how much better our implementation will perform against what a regular user could do today.

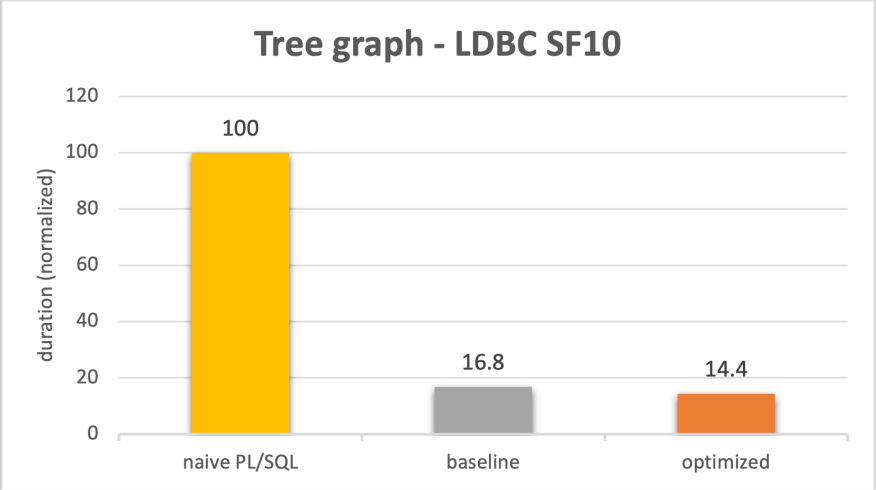
Since our implementation is integrated inside the runtime of the database engine, we eliminate much of the overhead from which the naive implementation suffers. Additionally, we are able to store temporary information in-memory and take advantage of existing database mechanism. As a result,

we expect to perform better than the naive implementation.

For the comparison we use datasets of different sizes. The size difference between each dataset is an order of magnitude. For this experiment we measure the execution time of $Q2$ and $Q3$. $Q2$ operates on a subset of LDBC that is a tree. $Q3$ operates on a subset of LDBC that has the structure of a social network. The subset on which the query will operate on is determined implicitly by the filters that are applied to the vertices and the edges.



(a) Tree subgraph of LDBC with scale factor 10

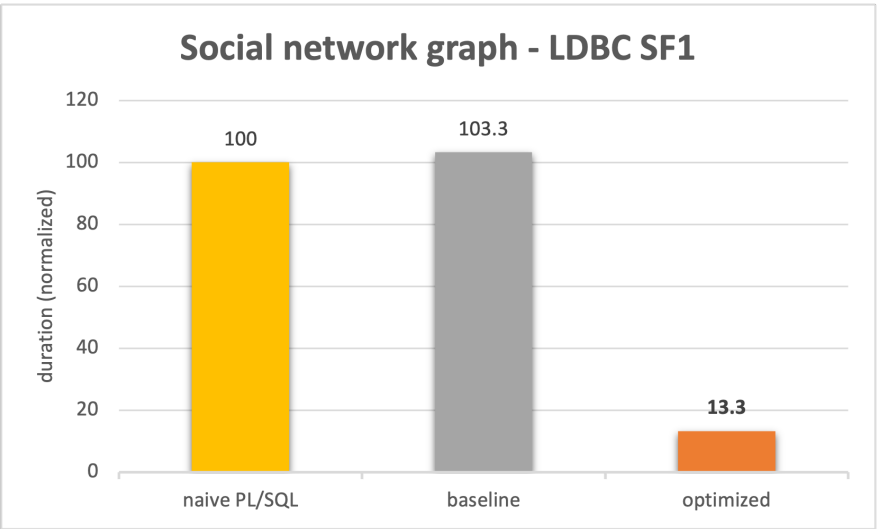


(b) Tree subgraph of LDBC with scale factor 10

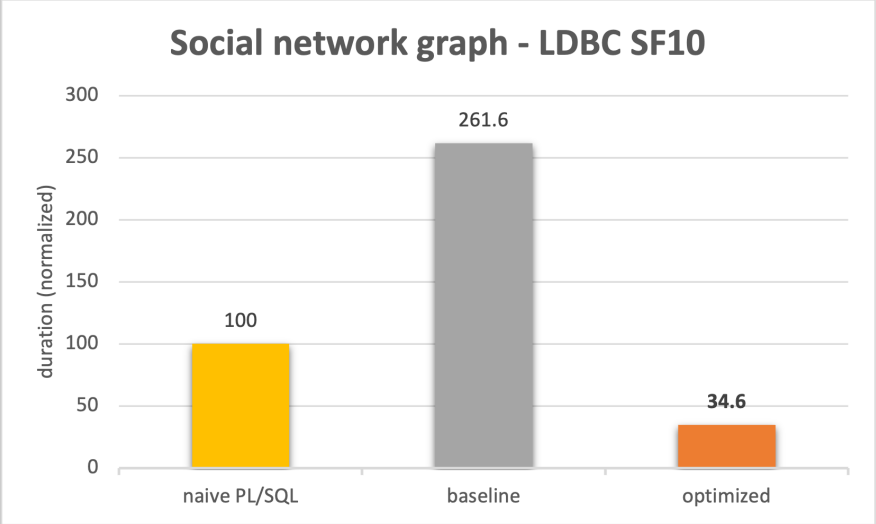
Figure 5.4: Performance comparison - tree subgraph of LDBC

We can see that our system outperforms the user-executed PL/SQL script. This happens because the PL/SQL script involves extra layers of parsing and execution that lead to slower performance. What's more, PL/SQL does not have access to the database internals. By extension, the naive implementation cannot use insertion batching, as the APIs we use to implement it are internal only. On the contrary, our system can access the internals of the database engine directly and is capable of retrieving and manipulating data more efficiently.

Additionally, by looking at the execution times before the normalization, we can observe that the size of the input does not significantly affect the performance of each system. This is expected, as graph queries do not perform computations across the whole dataset. Rather, they begin from a



(a) Social subgraph of LDBC with scale factor 10



(b) Social subgraph of LDBC with scale factor 10

Figure 5.5: Performance comparison - social subgraph of LDBC

starting vertex and then proceed to explore a local region of the graph around this starting vertex. The size of the region that will be explored depends on the maximum path length parameter and is usually small when compared to the size of the whole graph. This means that the execution time of a graph query is not influenced severely by the size of the dataset. However, the execution time is affected by the characteristics of the region that will be explored. These characteristics depends on the maximum path length, the complexity of the graph pattern we aim to match and the graph density of the region, since dense regions will include vertices with larger degrees, which in turn lead to more edges that must be processed.

Finally, it is interesting to observe the differences between how the unoptimized version behaves for the tree and the ”social network” parts of the graph. In the social network sub-graph, there are more opportunities for batching compared to the tree sub-graph, since the social network sub-graph is more dense. This means that the optimizations we implemented have a greater effect on the ”social network” experiment. This can be understood by the difference in the performance gap between the

unoptimized and the optimized version for the two experiments. In the tree graph experiment, the performance gap between the unoptimized and the optimized version is not major. On the other hand, for the ”social network” experiments, the gap is substantial. In fact, for the social network experiment for scale factor 10, the naive PL/SQL solution outperforms the unoptimized version, even though the unoptimized version is integrated inside the database. This is indicative of the bad performance of the unoptimized version for dense graphs.

5.5 Insertion batch size

In this section we examine the trade-off between the algorithm's performance and the upper bound for the size of the insertion batches (§4.1) by presenting relevant experimental results. The following measurements originate from executions of $Q1$.

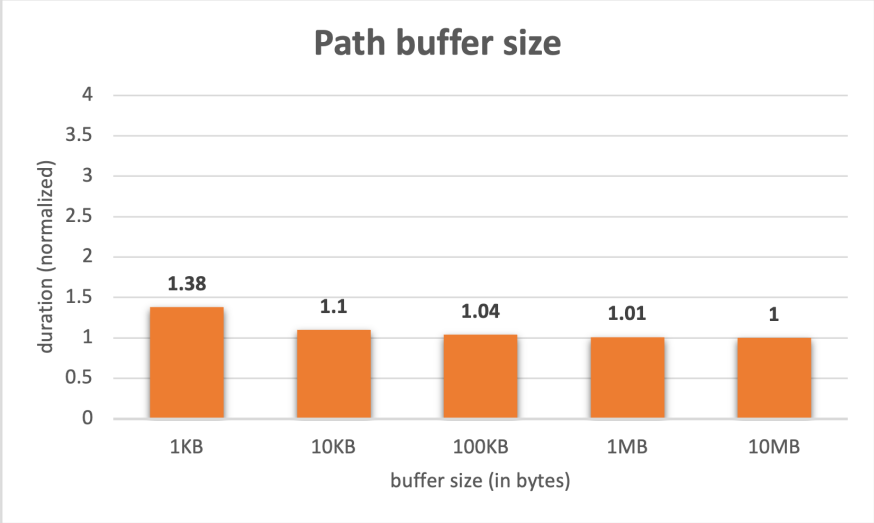


Figure 5.6: Insertion batch size memory-performance trade-off

The size of the batch is determined by the memory limit we set for the buffer in which we store the paths that will be inserted. We expect that if we allow the buffer to consume more memory, the system's performance will improve. This is because a larger buffer can store more paths. As an immediate consequence, a larger buffer enables insert batching with even larger batches. Larger batches are preferred because they reduce the average overhead per inserted row.

The experimental results verify our intuition. Indeed, more memory leads to better performance. However, after a certain point (100K) we have diminishing returns, as providing more memory does not lead to significantly better results. To generalize this conclusion and decide on the optimal value, we will have to run this experiment for a broad range of queries that are executed on graphs with different structures. This would lets us choose the ”sweet” spot regarding the size of the buffer.

The next figures let us dive deeper in the behavior of the buffer for different memory sizes. As a reminder, the length of the paths we want to store in the buffer monotonically increases, since the paths are found during the execution of a breadth-first search (BFS). As a consequence, the memory required to store each path also increases as the depth of the breadth-first search increases. It is worth noting that, when the size of a path reaches a point where it cannot fit in the buffer, we just flush it immediately.

- When we set the size of the buffer to **1KB**, we can see that the size of one path quickly exceeds the size of the buffer. This can be understood by the fact that after the 914th batch, every batch contains exactly one path. This means that the buffer is flushed every time a new path is discovered, because the buffer cannot fit more than one path. This is still better than no batching at all, since the original version performed one flush for every vertex of every path, while this version performs one batch per path.

- When we set the size of the buffer to **10KB**, the increased buffer size leads to a smoother decline in the number of paths-per-batch. The number of paths-per-batch stabilizes to 5 in this experiment. If we allowed the algorithm to continue its execution (by increasing the maximum path length parameter, which acts as a criterion for algorithm termination), the number of paths-per-batch would eventually converge to 1, as the size of the paths would exceed the size of the buffer.

- When we set the size of the buffer to **100KB**, we recognise an increase and then a decrease in the paths-per-batch line:

  – The breadth-first search algorithm begins from a starting vertex and gradually explores the graph. In the first BFS levels, the explored region is small and we do not have many paths to flush.

  – As the algorithm proceeds, the explored region becomes larger and larger. As a consequence, each iteration explores more paths. This drives an increases of paths-per-batch.

  – The aforementioned increase, combined with the increase in the path length, lead to an increment in the memory required to store all the paths of a BFS level in the buffer. After a certain point, the memory limit is exceeded and the buffer must be flushed before the end of the BFS level.

  – This means that, after this point, two or more batches are required to write a BFS level to the side tables. Initially, the first batch contains most of the paths and a second batch contains only the few paths that could not fit in the first path. As the depth of the BFS increases, the length of the paths increases too. Consequently, the memory required to store each path is also increased. As a result, fewer paths can fit in the first batch and more paths end up belonging to the second batch.

  – Finally, as the length of the paths increases even more, the number of paths we can store in the buffer begins to decrease. This explain the decrease in the paths-per-batch line.

- When we set the size of the buffer to **1MB**, we notice that the number of batches is equal to the number of breadth-first search levels. In other words, we only flush the buffer at the end of the BFS level. This means that we were never forced to flush the buffer because its memory consumption exceeded the upper bound.

- When we set the size of the buffer to **10MB**, the behaviour is identical to the behaviour we observed when buffer size was set to 1MB. This was anticipated, since for 1MB the flush buffer was never forced to flush before the end of the BFS level. Further increase in the size of the buffer does not offer any improvements.

If we compare the data when the maximum size of the buffer is 1MB to the data when the maximum size of the buffer is 100KB, we will notice that they are identical until the 59th data point. From that we deduce that when we set the memory limit for the buffer to 100KB, this memory limit is not exceeded until the paths' length becomes 60. Note that this is the point after which the paths-per-batch line starts decreasing when the memory limit is 100KB.

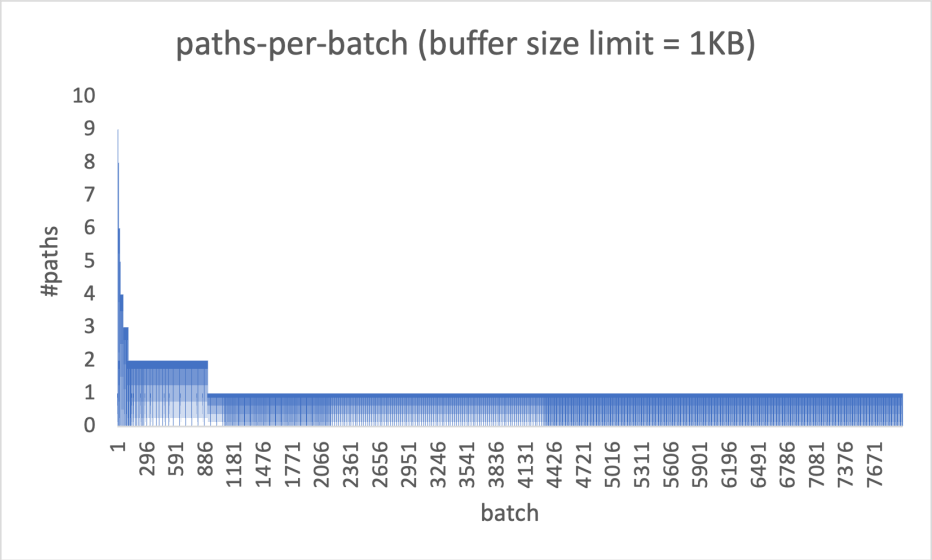


Figure 5.7: Paths per insertion batch for buffer memory limit = 1KB

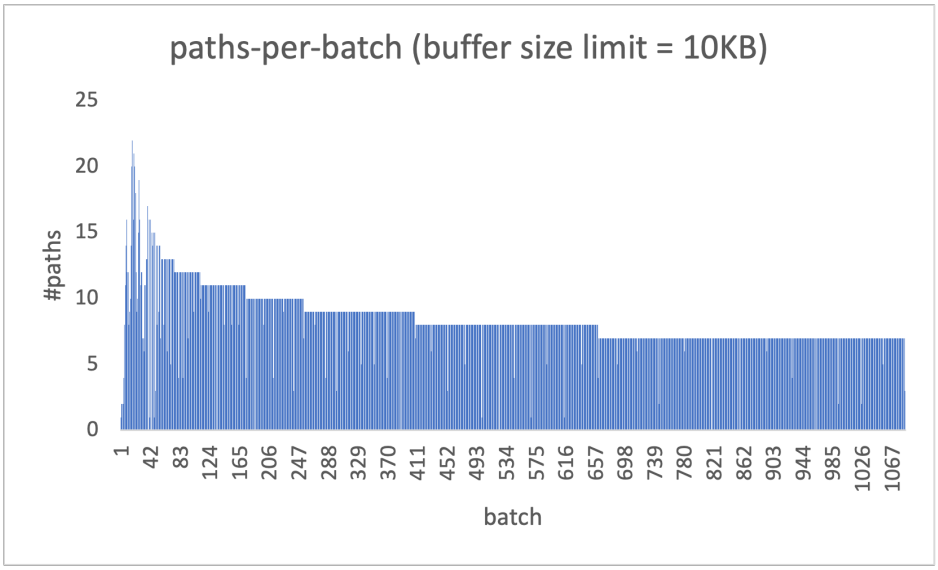


Figure 5.8: Paths per insertion batch for buffer memory limit = 10KB

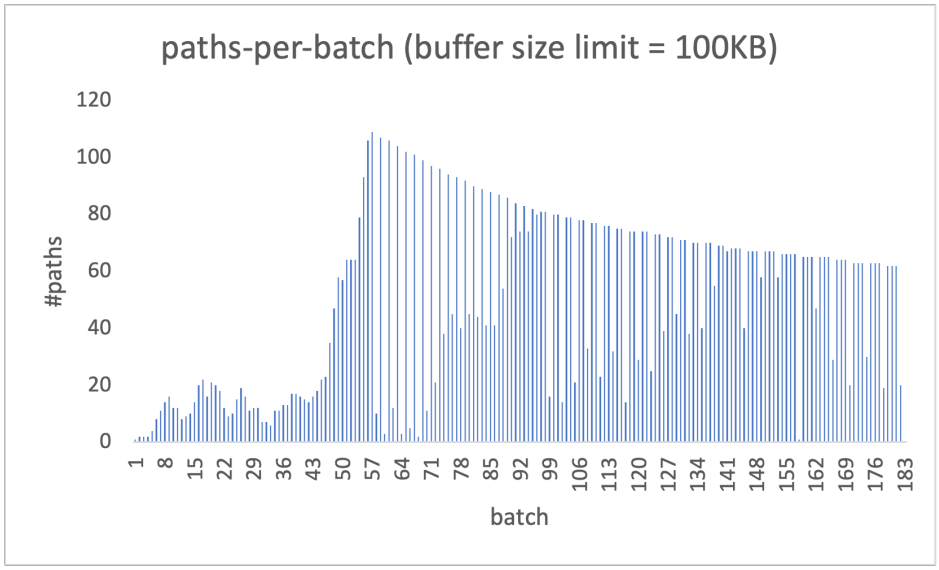


Figure 5.9: Paths per insertion batch for buffer memory limit = 100K

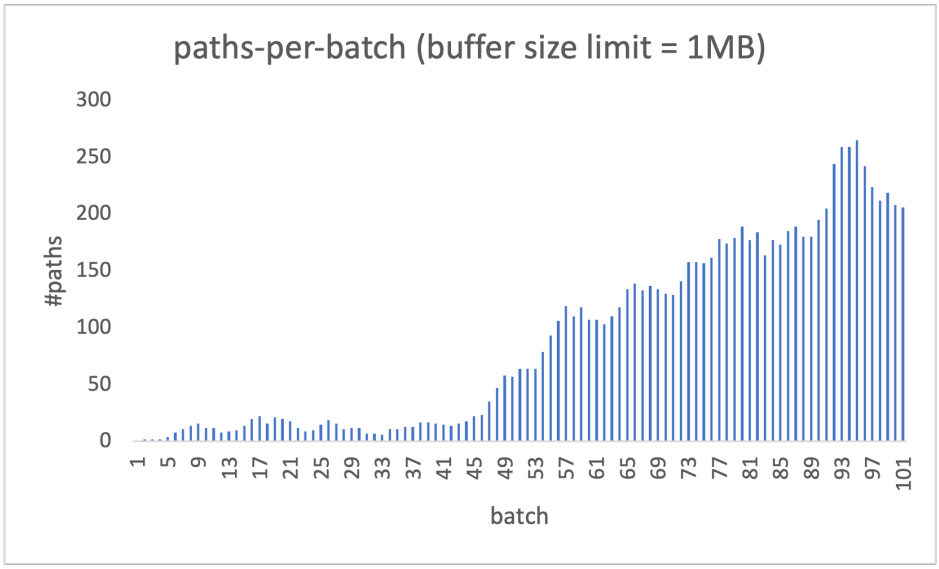


Figure 5.10: Paths per insertion batch for buffer memory limit = 1MB

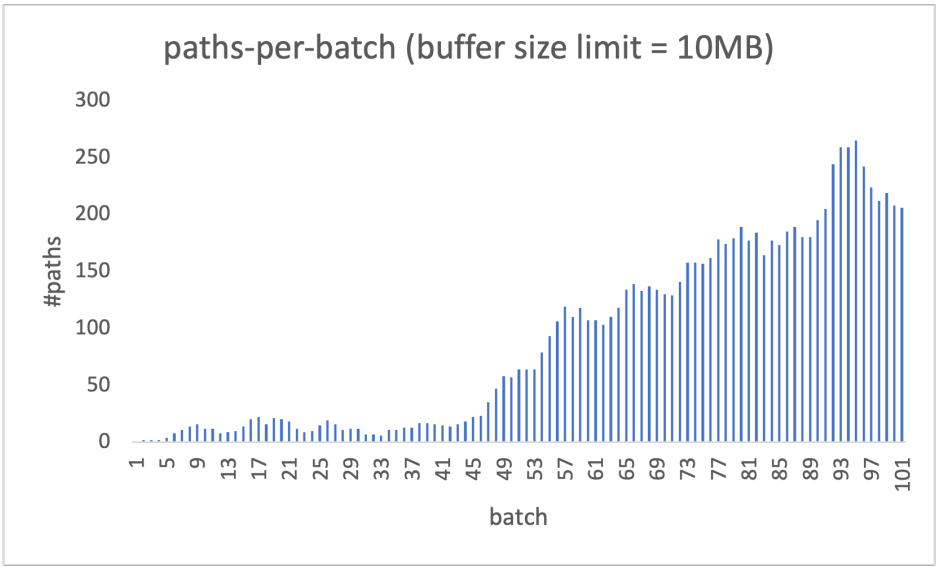


Figure 5.11: Paths per insertion batch for buffer memory limit = 10MB

Chapter 6

Future Directions

6.1 Skip side table inserts

Even after applying batch inserting, performance analysis of the algorithm's runtime with profiling tools shows that writes to the side tables are still a significant part of the total execution time of unbounded RPQs. Since the algorithm kernel generates its results one after the other while it runs, we could push the results directly to the input of the suffix without writing them to the side tables. Due to technical limitations of the system, currently this optimization cannot be applied to queries that include more than one vertex table. However, it is expected that queries that involve only one vertex table will be the common case. For these queries, this optimization will reduce total execution time by 50%, according to profiling results.

6.2 Many-to-many query optimization

Right now, we execute one graph traversal for each vertex v that is a destination vertex for a prefix path. If for a given query multiple paths match the prefix, then we have to traverse the graph multiple times. This can be problematic when there are numerous destination vertices for paths that match the prefix and only few vertices that match the suffix. In such cases, `(a)→*(b)` may be cheaper to compute compared to `(b)←*(a)`.

For example in figure 6.1, when a user runs a query to find all the paths that reach A, it will execute one breadth-first search for every possible starting point. It would be more efficient to just execute a BFS that would traverse the edges reversly, with A as the starting vertex.

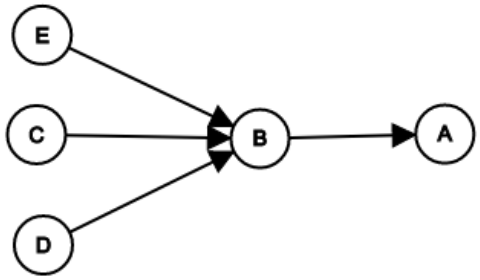


Figure 6.1: Example for multi-source BFS

To implement this optimization, because in our data model the property graph is directed, we have to reverse the direction of the edges in order to process the path in reverse order. It is worth noting

that it is challenging to define a heuristic to decide when evaluating the reverse path is cheaper. The high-level intuition of this heuristic would be that we want the lowest cardinality side to be the prefix of the unbounded RPQ.

6.3 Multi-source breadth-first search

To further improve performance in the many-to-many query scenario, we could traverse the graph with a multi-source breadth-first search [41] instead of executing multiple single-source breadth-first searches. Multi-source breadth first search is an algorithm that explores a graph from multiple starting points simultaneously. The usage of multi-source breadth-first search could lead to performance improvements, however it comes with substantial engineering cost due to intricacies of the system.

6.4 Batching for neighbor expansion

To discover the next BFS level, we have to find the neighbours of every vertex that belongs to the current BFS level. Following the SQL-based philosophy of the system, for every vertex in the current BFS level we dynamically issue `SELECT` queries in order to retrieve its neighbours. For a given vertex v that belongs to a vertex table VT, the number of `SELECT` queries we issue to retrieve its neighbours is equal to the number of edge tables for which VT is a source vertex table.

When the graph we traverse is very large, BFS levels may contain a significant number of vertices. By extension, we issue a significant number of `SELECT` queries. We can improve the performance of neighbour expansion by utilizing the fact that we already know every vertex that belongs to the current BFS level when we begin its expansion. This means that we could retrieve the next BFS level by issuing just one `SELECT` query per edge table. This `SELECT` query would retrieve all the neighbours of vertices that belong to the current BFS level, such that the edge that connects them is defined in the queried edge table.

This optimization would improve performance by substantially reducing the number of `SELECT` statements that are issued during neighbour expansion. In addition to this, applying this optimization would amplify the positive effects of batching for neighbor fetching (§4.4), since each `SELECT` statement would return more rows, leading to larger batches during fetching. This optimization was not implemented due to time constraints.

Chapter 7

Summary

Graphs are useful in a wide variety of applications. As graphs gain more and more attention from industry practitioners and scientists, there is an increased demand for technologies that facilitate extraction of insights from large-scale graph datasets. Specialized graph processing systems have been developed to serve data scientists that want to handle massive amounts of graph-structured data.

Even though specialized graph processing systems that perform well exist, there is a rising need for techniques that enable graph processing in the context of a relational database. Relational databases are the standard solution that organizations such as large enterprises and governments prefer in order to store, organize and manipulate their data. As a result, users often want to perform graph processing tasks over data that are already stored inside a relational database. Moving this data from the relational database to an external graph processing framework is a time-consuming and complex task. Moreover, since the data are will be duplicated, the users are forced to setup ETL processes to ensure that the results of the graph analytics task are fresh. Furthermore, performing graph analytics inside the database lets the users utilize the well-researched capabilities that relational databases offer, such as fault tolerance, security and support for transaction. At the same time, it helps them avoid the increased complexity that comes with establishing ETL process and managing one extra distributed system.

One of the most common use cases that are related to graph processing is the retrieval of subgraphs that match a given patterns. This type of requests are called graph queries. The patterns that the user aims to retrieve can be either fixed-length patterns, such as `(p1:Person)-[:Knows]->(p2:Person)`, or variable-length patterns. Variable-length patterns can be expressed through regular expressions. Those queries are also called regular path queries (RPQs). RPQs are classified as either bounded RPQs or unbounded RPQs. Bounded RPQs have an upper-bound for the length path, while unbounded RPQs do not.

Unbounded RPQs have many applications in many industries such as retail, social media and finance. For example, they assist investigators to detect suspicious transactions that could be related to money laundering. Even though unbounded RPQs are an important feature that greatly benefits users, the current methods to evaluate them fall short in terms of performance. The primary issue that prohibits these approaches from achieving sufficient performance is the fact that they do not have access to the database internals. To address this problem, a specialized that is embedded in the database has been developed with the goal to offer better performance.

The execution of the specialised kernel is SQL-based. This means that it issues SQL statements during runtime to execute the algorithm's steps. For example, it issues a `SELECT` statement to retrieve the neighbours of a vertex. The kernel essentially traverses the graph by executing a breadth-first

search with a given vertex s as the starting vertex and then returns a path for every vertex that is reachable from s. The output of the algorithm is stored in tables.

Even though the kernel is tightly integrated with the database, it does not perform well enough for large datasets. Performance analysis indicates that this happens because SQL statements are not handled efficiently. In this thesis, we implemented optimisations that target the performance bottlenecks of the kernel. Specifically, we use batching to accelerate row insertion to the tables at which the output is written. Additionally, we implement prepared statement to reduce the overhead that comes with repeatedly compiling the SQL statements. Finally, we fetch the neighbours of each vertex in batches to further reduce execution time.

Our experiments show that our optimisations lead to up to x67 improvement in performance compared to the original version of the specialized kernel without increasing its memory requirements.

Bibliography

[1] https://www.w3.org/TR/sparql11-query/. Accessed: 2024-01-26.

[2] https://aws.amazon.com/nosql/graph/. Accessed: 2024-01-26.

[3] https://neo4j.com/developer/graph-database/. Accessed: 2024-01-26.

[4] https://www.oracle.com/autonomous-database/what-is-graph-database/. Accessed: 2024-01-26.

[5] https://github.com/gkosm314/giraph-vs-gelly/blob/main/report.pdf. Accessed: 2024-01-26.

[6] https://www.gqlstandards.org/. Accessed: 2024-01-26.

[7] https://docs.oracle.com/cd/B10500_01/server.920/a96533/hintsref.htm. Accessed: 2024-01-26.

[8] http://www.w3.org/RDF/. Accessed: 2024-01-26.

[9] R. Angles, J. B. Antal, A. Averbuch, A. Birler, P. Boncz, M. Búr, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. L. L. Pey, N. Martínez, J. Marton, M. Paradies, M.-D. Pham, A. Prat-Pérez, D. Püroja, M. Spasić, B. A. Steer, D. Szakállas, G. Szárnyas, J. Waudby, M. Wu, and Y. Zhang. The ldbc social network benchmark, 2024.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117, 1998.

[11] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing-a paradigm for transportable software. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 268–275 vol.2, 1995.

[12] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, aug 2015.

[13] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[15] T. Faltín, V. Trigonakis, A. Berdai, L. Fusco, C. Iorgulescu, J. Lee, J. Yaghob, S. Hong, and H. Chafi. Distributed asynchronous regular path queries (rpqs) on graphs. In *Proceedings of the 24th International Middleware Conference: Industrial Track*, Middleware '23, page 35–41, New York, NY, USA, 2023. Association for Computing Machinery.

[16] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[17] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoc. A researcher's digest of gql. In *The 26th International Conference on Database Theory, 2023*, pages 1–1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[18] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, Oct. 2014. USENIX Association.

[20] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1789–1792, New York, NY, USA, 2018. Association for Computing Machinery.

[21] P. Hayes. Rdf semantics: W3c recommendation 10 february 2004. *W3C Recommendation*, 02 2004.

[22] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. Pgx.d: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] G. Jin, N. Anzum, and S. Salihoglu. Graindb: A relational-core graph-relational dbms. In *Conference on Innovative Data Systems Research*, 2022.

[24] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! *Proc. VLDB Endow.*, 7(13):1669–1672, aug 2014.

[25] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: a new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on*

Uncertainty in Artificial Intelligence, UAI'10, page 340–349, Arlington, Virginia, USA, 2010. AUAI Press.

[27] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[28] M. Morales, V. I. Haprian, S. Karthik, D. Porobic, L. Daynès, and A. Ailamaki. Efficient property projections of graph queries over relational data. In *CDMS@VLDB*, 2022.

[29] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

[30] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 627–640. ACM, 2009.

[31] K. Panetta. Gartner top 10 data and analytics trends for 2021. https://www.gartner.com/smarterwithgartner/gartner-top-10-data-and-analytics-trends-for-2021. Accessed: 2024-01-26.

[32] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[33] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[34] I. Research. Global graph database market forcase 2019-2027. https://inkwoodresearch.com/reports/global-graph-database-market. Accessed: 2024-01-26.

[35] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[36] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146. ACM, 2013.

[37] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[38] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1887–1901, New York, NY, USA, 2015. Association for Computing Machinery.

[39] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: high performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, jul 2015.

[40] F. Tetzel, R. Kasperovics, and W. Lehner. Graph traversals for regular path queries. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[41] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, dec 2014.

[42] Y. Tian. The world of graph databases from an industry perspective, 2022.

[43] C.-H. Tseng. Efficient in-memory processing of recursive path graph queries in a relational database. *EPFL Master Thesis*, 2023.

[44] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[45] D. t. Wolde, G. Szárnyas, and P. Boncz. Duckpgq: Bringing sql/pgq to duckdb. *Proc. VLDB Endow.*, 16(12):4034–4037, aug 2023.

[46] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 897–912, New York, NY, USA, 2017. Association for Computing Machinery.