National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science and Technology
Database and Knowledge Systems Laboratory

Integrating a Graph Database as a Machine Learning Feature Store Registry

DIPLOMA THESIS

CHRYSOULA E. PANIGYRAKI

Supervisor : Dimitrios Tsoumakos
Associate Professor, NTUA

Athens, February 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science and Technology
Database and Knowledge Systems Laboratory

Integrating a Graph Database as a Machine Learning Feature Store Registry

DIPLOMA THESIS

CHRYSOULA E. PANIGYRAKI

Supervisor : Dimitrios Tsoumakos
Associate Professor, NTUA

Approved by the examining committee on the February 27, 2025.

Dimitrios Tsoumakos Associate Professor, NTUA	Georgios Goumas Associate Professor, NTUA	Athanasios Voulodimos Assistant Professor, NTUA

Athens, February 2025

Chrysoula E. Panigyraki
Graduate of Electrical and Computer Engineering, NTUA

Περίληψη

Η ταχεία εξέλιξη των συστημάτων μηχανικής μάθησης και η εκτεταμένη εφαρμογή τους σε περιβάλλοντα παραγωγής έχουν αναδείξει την ανάγκη για αξιόπιστες και κλιμακούμενες λύσεις για τη διαχείριση χαρακτηριστικών (features). Η αποθήκη χαρακτηριστικών (feature store), ένα αποθετήριο που ελέγχει κεντρικά τα χαρακτηριστικά που χρησιμοποιούνται στην εκπαίδευση και την εξαγωγή συμπερασμάτων των μοντέλων, είναι αναπόσπαστο κομμάτι αυτών των συστημάτων. Συμβατικά, τα feature stores αποθηκεύουν μεταδεδομένα χαρακτηριστικών και στοιχεία γενεαλογίας σε καταλόγους βασισμένους σε SQL. Αυτή η μέθοδος δεν μπορεί να μοντελοποιήσει και να υποβάλει ερωτήματα αποτελεσματικά σε χαρακτηριστικά και οντότητες με πολύπλοκες εξαρτήσεις μεταξύ τους, ενώ υπάρχουν εναλλακτικά εργαλεία που προορίζονται για τέτοιες εργασίες. Η παρούσα μελέτη διερευνά κατά πόσον μια βάση δεδομένων γράφων μπορεί να λειτουργήσει ως κατάλογος χαρακτηριστικών, αξιοποιώντας την εγγενή ικανότητά της να αναπαριστά περίπλοκες σχέσεις και γενεαλογία με πιο φυσικό τρόπο.

Αυτή η εργασία επεκτείνει το Feast, ένα feature store ανοικτού κώδικα που χρησιμοποιείται ευρέως στη βιομηχανία, με έναν κατάλογο βασισμένο στο Neo4j, χτίζοντας πάνω σε προηγούμενες έρευνες που δίνουν έμφαση στα πλεονεκτήματα των βάσεων δεδομένων γράφων στις διασχίσεις πολλαπλών βημάτων και στις αναζητήσεις που βασίζονται σε σχέσεις. Η προσέγγισή μας μοντελοποιεί τα αντικείμενα του feature store ως κόμβους, με τις εξαρτήσεις και τη διαδοχή τους να αποτυπώνονται ως σχέσεις. Προστέθηκε νέα λειτουργικότητα για την ανίχνευση σχέσεων μεταξύ χαρακτηριστικών, μαζί με προσαρμοσμένες εντολές CLI που σχεδιάστηκαν για να επωφεληθούν από την περιγραφική δύναμη των ερωτημάτων γράφων. Για την αξιολόγηση της προτεινόμενης λύσης, πραγματοποιήθηκαν δοκιμές επιδόσεων μετρώντας τον χρόνο εκτέλεσης, χρησιμοποιώντας καταλόγους που βασίζονται σε γράφους, σε SQL και σε αρχεία.

Τα αποτελέσματα υποδεικνύουν ότι ο κατάλογος με βάση τον γράφο υπερέχει στη διαχείριση περίπλοκων σχέσεων πολλαπλών βημάτων και είναι ιδιαίτερα αποτελεσματικός για εφαρμογές που απαιτούν βαθιά ανάλυση εξαρτήσεων. Ωστόσο, τα πλεονεκτήματά του όσον αφορά τη χρηστικότητα και τις επιδόσεις είναι πιο αισθητά όταν τα ερωτήματα περιλαμβάνουν πολύπλοκες διασχίσεις σχέσεων, ενώ για απλούστερα, πιο άμεσα ερωτήματα τα οφέλη του μπορεί να είναι λιγότερο σημαντικά και να επισκιάζονται από την επιβάρυνση του προγραμματισμού των ερωτημάτων.

Εν κατακλείδι, η παρούσα μελέτη καταδεικνύει πώς οι βάσεις δεδομένων γράφων μπορούν να αναπαραστήσουν αποτελεσματικά τις περίπλοκες σχέσεις μεταξύ των πληροφοριών, βελτιώνοντας τη διαχείριση και την ερμηνευσιμότητα των συστημάτων μηχανικής μάθησης. Προσφέροντας εμπειρική απόδειξη των πλεονεκτημάτων και των μειονεκτημάτων των μεθόδων που βασίζονται σε γράφους, όχι μόνο καλύπτει ένα κενό στη βιβλιογραφία για τα feature stores, αλλά ανοίγει, επίσης, τον δρόμο για περαιτέρω έρευνα σε υβριδικές αρχιτεκτονικές καταλόγων, που μπορούν να εξισορροπήσουν δυναμικά τα προτερήματα διαφόρων συστημάτων backend και να βελτιστοποιήσουν τις ερωτήσεις γράφων.

Λέξεις κλειδιά

Μηχανική Μάθηση, Αποθήκη Χαρακτηριστικών, Feast, Γενεαλογία, Κατάλογος Χαρακτηριστικών Γράφου, Βάση Δεδομένων Γράφων, Neo4j.

Abstract

The rapid evolution of machine learning systems and their extended application in production environments have unveiled the need for robust and scalable feature management solutions. The feature store, a repository that centrally controls features used in model training and inference, is essential to these systems. Traditionally, feature stores have stored feature metadata and lineage data in SQL-based registries. This method cannot effectively model and query complex dependencies between features and entities, whereas there are alternative tools intended for such tasks. This study investigates whether a graph database can function as a feature store registry, leveraging its native capability to represent intricate relationships and lineage in a more natural manner.

This work extends Feast, an open-source feature store that is widely used in industry, with a Neo4j-backed registry, building on previous research that emphasizes the strengths of graph databases in multi-hop traversals and relationship-based searches. Our approach models feature store objects as nodes, with their dependencies and lineage captured as relationships. New functionality was added to detect relationships between features, along with custom CLI commands designed to benefit from the descriptive power of graph queries. To evaluate the proposed solution, performance tests were conducted measuring execution time across graph-based, SQL-based and file-based registries.

The results indicate that the graph-based registry excels in managing intricate, multi-hop relationships and is particularly effective for applications requiring deep dependency analysis. However, its usability and performance advantages are most pronounced when the query patterns involve complex relationship traversals, whereas for simpler, more direct queries its benefits may be less significant and may get overshadowed by the query planning overhead.

In conclusion, this study demonstrates how graph databases may effectively represent the intricate relationships between information, improving the management and interpretability of machine learning systems. By offering empirical proof of the advantages and disadvantages of graph-based methods, it not only closes a gap in the feature store literature but also paves the way for further research into hybrid registry architectures that can dynamically balance the advantages of several backend systems and optimize graph operations.

Keywords

Machine Learning, Feature Store, Feast, Lineage, Graph-based Feature Registry, Graph Database, Neo4j.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της εργασίας, κ. Δημήτριο Τσουμάκο, για την καθοδήγηση και την εμπιστοσύνη του.

Ακόμα, θέλω να ευχαριστήσω τους συμφοιτητές και φίλους μου, που πάντα πιστεύουν σε εμένα, με στηρίζουν και με ενθαρρύνουν, ιδιαίτερα κατα την διάρκεια πραγμάτωσης της συγκεκριμένης εργασίας.

Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, οι οποίοι με υποστήριξαν και έκαναν δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Χρυσούλα Πανηγυράκη, Φεβρουάριος 2025

Contents

List of Tables

List of Figures

Κεφάλαιο 1

Εκτεταμένη Περίληψη στα Ελληνικά

1.1 Εισαγωγή

Η ραγδαία εξέλιξη της μηχανικής μάθησης έχει αναδείξει τη σημασία της διαχείρισης δεδομένων και της μηχανικής χαρακτηριστικών. Τα χαρακτηριστικά (features), που προέρχονται από ακατέργαστα δεδομένα, είναι θεμελιώδη για την εκπαίδευση και πρόβλεψη μοντέλων. Ωστόσο, ο μεγάλος όγκος και η πολυπλοκότητα των σύγχρονων δεδομένων δυσχεραίνουν τη διαχείρισή τους με συνεπή και κλιμακούμενο τρόπο.

Για την αντιμετώπιση αυτής της πρόκλησης δημιουργήθηκαν οι αποθήκες χαρακτηριστικών (feature stores), που διασφαλίζουν τη συνέπεια μεταξύ εκπαίδευσης και πρόβλεψης. Ενώ πολλές υπάρχουσες λύσεις ανοικτού κώδικα βασίζονται σε SQL βάσεις δεδομένων, αυτές δυσκολεύονται να ανακαλύψουν πολύπλοκες αλληλεξαρτήσεις μεταξύ των οντοτήτων.

Η παρούσα μελέτη εξετάζει τη χρήση βάσεων δεδομένων γράφων ως εναλλακτική λύση, καθώς είναι σχεδιασμένες για τη μοντελοποίηση και διάσχιση πολύπλοκων σχέσεων. Διερευνάται κατά πόσο ένας κατάλογος βασισμένος σε γράφους μπορεί να προσφέρει βελτιωμένες δυνατότητες για την παρακολούθηση εξαρτήσεων και τη διάσχιση πολλαπλών συνδέσμων, ξεπερνώντας τους περιορισμούς των συμβατικών καταλόγων.

1.2 Αποθήκες χαρακτηριστικών

Μια αποθήκη χαρακτηριστικών είναι ένα εξειδικευμένο σύστημα διαχείρισης δεδομένων για εφαρμογές μηχανικής μάθησης, το οποίο εξυπηρετεί πολλούς σημαντικούς σκοπούς [1]. Πρώτον, απλοποιεί τη διαχείριση, αποθήκευση και παροχή των δεδομένων που χρειάζονται τα μοντέλα μηχανικής μάθησης. Παράλληλα, λειτουργεί ως κεντρικός κόμβος για τη συγκέντρωση χαρακτηριστικών και μεταδεδομένων, διευκολύνοντας την οργάνωση και πρόσβαση σε αυτά. Επιπλέον, εκτελεί μετατροπές από ακατέργαστα δεδομένα σε χρήσιμα χαρακτηριστικά, καθιστώντας τα έτοιμα για χρήση από τα μοντέλα. Η αποθήκευση των δεδομένων καλύπτει τόσο ιστορικά δεδομένα όσο και δεδομένα σε πραγματικό χρόνο, διασφαλίζοντας παράλληλα τη χρονική συνέπεια των δεδομένων εκπαίδευσης χωρίς διαρροές πληροφορίας. Τέλος, η αποθήκη χαρακτηριστικών επιτρέπει την εύκολη εξερεύνηση και επαναχρησιμοποίηση των δεδομένων, ενώ παρέχει αποτελεσματικές διεπαφές για πρόσβαση στα δεδομένα, είτε πρόκειται για εκπαίδευση είτε για προβλέψεις.

Τα βασικά χαρακτηριστικά του συστήματος περιλαμβάνουν τη διαχείριση μετασχηματισμών δεδομένων, την πολυεπίπεδη αποθήκευση, τη συνεπή παροχή δεδομένων στα μοντέλα, τη δυνατότητα παρακολούθησης μετρικών ποιότητας και τη διατήρηση ενός κεντρικού καταλόγου χαρακτηριστικών.

Πέρα από το Feast, την αποθήκη χαρακτηριστικών ανοιχτού κώδικα την οποία επεκτείνει η παρούσα εργασία, υπάρχουν κι άλλες λύσεις ανοιχτού κώδικα που αξιοποιούν μεταξύ άλλων και σχεσιακές βάσεις για να διατηρούν μεταδεδομένων και γενεαλογία των χαρακτηριστικών. Όσον

αφορά το Feast, αποτελείται από τα ακόλουθα υποσυστήματα [2]:

- Αποθήκη Εκτός Σύνδεσης (Offline Store): Αποθηκεύει ιστορικές τιμές χαρακτηριστικών για την εκπαίδευση μοντέλων και επεξεργασία σε παρτίδες, ενώ παρέχει χρονικά ακριβείς συνδέσεις, υποστηρίζοντας πλατφόρμες όπως Snowflake, BigQuery και PostgreSQL.

- Αποθήκη Σε Σύνδεση (Online Store): Περιέχει μόνο τις πιο πρόσφατες τιμές χαρακτηριστικών για προβλέψεις πραγματικού χρόνου και είναι βελτιστοποιημένη για χαμηλή καθυστέρηση και υψηλή διαθεσιμότητα, χρησιμοποιώντας βάσεις δεδομένων όπως Redis, DynamoDB και Sqlite.

- Μηχανή Υλοποίησης Παρτίδας (Batch Materialization Engine): Μεταφέρει δεδομένα από την offline στην online αποθήκη, χρησιμοποιώντας είτε τοπική διαδικασία είτε κλιμακώσιμες λύσεις όπως το AWS Lambda.

- Κατάλογος Χαρακτηριστικών (Feature Registry): Αποθηκεύει όλα τα αντικείμενα του Feast και τους ορισμούς τους, χρησιμοποιώντας είτε σύστημα βασισμένο σε αρχεία (σε μορφή protobuf) είτε κατάλογο βασισμένο σε SQL που δέχεται ερωτήματα μέσω SQLAlchemy.

1.2.1 Αντικείμενα Feast

Ένα έργο Feast αποτελεί τη βασική μονάδα οργάνωσης στην αποθήκη χαρακτηριστικών, παρέχοντας απομονωμένα περιβάλλοντα για διαφορετικές εφαρμογές. Περιλαμβάνει τα εξής αντικείμενα:

- Πηγές Δεδομένων (Data Sources): Παρέχουν τα ακατέργαστα δεδομένα που εισάγει και επεξεργάζεται το Feast, οργανωμένα ως χρονοσειρές. Κατηγοριοποιούνται σε πηγές παρτίδων (π.χ. αποθήκες δεδομένων), ροής (π.χ. Kafka) και πηγές αιτήματος για δεδομένα που διατίθενται μόνο κατά την εκτέλεση.

- Οντότητες (Entities): Έννοιες-Σύνολα ιδιοτήτων που αντιπροσωπεύουν αντικείμενα του τομέα εφαρμογής, όπως πελάτες ή οδηγοί, με μοναδικό όνομα και κλειδί σύνδεσης.

- Προβολές Χαρακτηριστικών (Feature Views): Λογικές ομαδοποιήσεις δεδομένων σειρών χρόνου, που αντλούνται από πηγές δεδομένων. Περιλαμβάνουν οντότητες, σχήματα και προαιρετικά μεταδεδομένα, υποστηρίζοντας τη δημιουργία ιστορικών συνόλων εκπαίδευσης, τη φόρτωση δεδομένων στην online αποθήκη και την ανάκτηση χαρακτηριστικών.

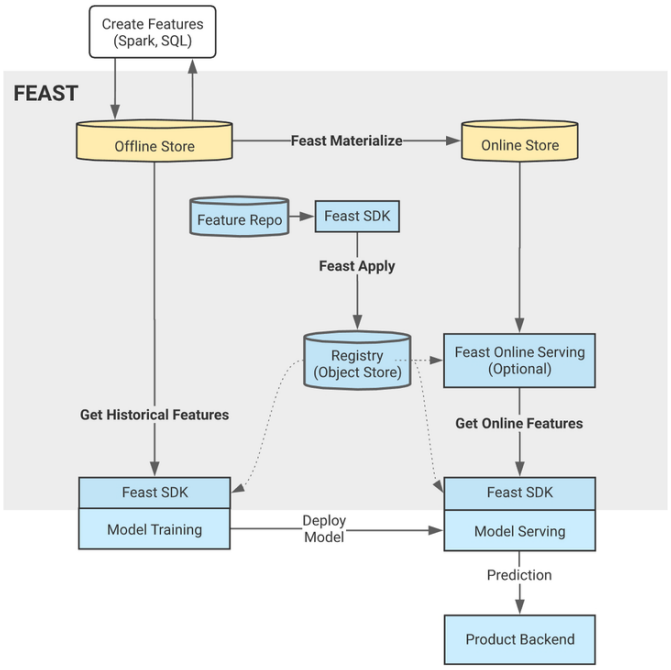
- Δυναμικές Προβολές Χαρακτηριστικών (On-demand Feature Views): Προβολές με ελαφρείς μετασχηματισμούς που εκτελούνται δυναμικά μέσω Python, αξιοποιώντας υπάρχοντα χαρακτηριστικά και πηγές αιτήματος.

- Υπηρεσίες Χαρακτηριστικών (Feature Services): Ομαδοποιήσεις χαρακτηριστικών από πολλές προβολές, που διευκολύνουν την πρόσβαση των μοντέλων στα απαιτούμενα δεδομένα. Συνιστάται η δημιουργία μίας υπηρεσίας ανά έκδοση μοντέλου για καλύτερη παρακολούθηση.

1.2.2 Λειτουργίες Feast

Οι βασικές λειτουργίες του Feast απεικονίζονται και στην εικόνα 1.1 και περιλαμβάνουν:

- Feast Apply: Συγχρονίζει το feature store με τα αρχεία ρυθμίσεων, σαρώνει τους ορισμούς σε Python για οντότητες, προβολές χαρακτηριστικών και πηγές δεδομένων, επικυρώνει τις αλλαγές και ενημερώνει την υποδομή.



Σχήμα 1.1: Αρχιτεκτονική του Feast.

- Feast Materialize: Φορτώνει δεδομένα χαρακτηριστικών για συγκεκριμένα χρονικά διαστήματα στην online αποθήκη, εκτελεί ερωτήματα στην offline αποθήκη, χαρτογραφεί τα αποτελέσματα και ενημερώνει τον κατάλογο χαρακτηριστικών.
- Ανάκτηση Χαρακτηριστικών:
  - get_historical_features: Δημιουργεί σύνολα εκπαίδευσης με χρονική συνέπεια, αναπαράγοντας την ιστορική κατάσταση των χαρακτηριστικών.
  - get_online_features: Παρέχει τις πιο πρόσφατες τιμές χαρακτηριστικών για online προβλέψεις, ανακτώντας τα δεδομένα από την online αποθήκη για τις ζητούμενες οντότητες.

1.3 Βάσεις Δεδομένων Γράφων

Οι βάσεις δεδομένων γράφων χρησιμοποιούν ένα μοντέλο δεδομένων βασισμένο σε κόμβους και ακμές για την αναπαράσταση και την αποδοτική ανάκτηση σύνθετων σχέσεων. Διαφέρουν από τις σχεσιακές βάσεις δεδομένων, καθώς είναι βελτιστοποιημένες για τη διαχείριση σχέσεων και παρέχουν ευέλικτα σχήματα που προσαρμόζονται εύκολα σε αλλαγές στη δομή των δεδομένων.

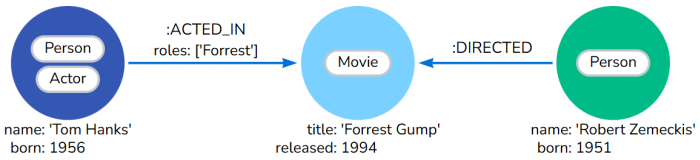
1.3.1 Κύριες Διαφορές με Σχεσιακές Βάσεις Δεδομένων

- Απόδοση: Οι βάσεις γραφημάτων διατρέχουν σχέσεις απευθείας, ενώ οι σχεσιακές εξαρτώνται από συνενώσεις πινάκων (joins) που επιβραδύνουν σύνθετες ερωτήσεις.
- Ευελιξία Σχήματος: Οι σχεσιακές βάσεις απαιτούν προκαθορισμένο σχήμα, ενώ οι βάσεις γραφημάτων επιτρέπουν την αλλαγή του χωρίς εκτεταμένες τροποποιήσεις.
- Περιπτώσεις Χρήσης: Οι βάσεις γραφημάτων είναι ιδανικές για εφαρμογές όπου οι σχέσεις είναι κεντρικής σημασίας, όπως συστήματα προτάσεων και ανίχνευσης απάτης, ενώ οι σχεσιακές είναι πιο κατάλληλες για σταθερά, δομημένα δεδομένα.

1.3.2 Επισκόπηση του Neo4j

Το Neo4j (Network Exploration and Optimization for Java) είναι μία κορυφαία πλατφόρμα βάσης γραφημάτων, που παρέχει μια ισχυρή και επεκτάσιμη λύση για την αποθήκευση, ανάκτηση και ανάλυση διασυνδεδεμένων δεδομένων. Περιλαμβάνει μια εύχρηστη γλώσσα ερωτημάτων, τη Cypher, αλγορίθμους γραφημάτων και εργαλεία οπτικοποίησης, διευκολύνοντας τόσο τους προγραμματιστές όσο και τους αναλυτές. Παράλληλα υποστηρίζει οδηγούς και connectors σε διάφορες προγραμματιστικές γλώσσες, όπως Python, JavaScript και Java [3].

Το Neo4j βασίζεται στο μοντέλο ιδιοτήτων, όπου οι κόμβοι περιγράφουν οντότητες και υπάρχουν κατευθυνόμενες σχέσεις που συνδέουν κόμβους. Τόσο οι κόμβοι όσο και οι ακμές μπορούν να έχουν χαρακτηριστικά σε μορφή ζεύγους κλειδιού-τιμής, ενώ οι κόμβοι έχουν, επιπλέον, ετικέτες και αντίστοιχα οι σχέσεις έχουν κάποιον τύπο. Στην εικόνα 1.2 παρουσιάζεται ενδεικτικά ένας τμήμα απλού γραφήματος.



Σχήμα 1.2: Παράδειγμα απλού γράφου στο Neo4j.

Η γλώσσα ερωτημάτων Cypher, που χρησιμοποιείται στο Neo4j, προσφέρει απλή, ευανάγνωστη σύνταξη για την πλοήγηση στο γράφημα. Οι κόμβοι αναπαριστώνται ως (κόμβος), οι σχέσεις ως -[:ΣΧΕΣΗ]-> και οι διαδρομές μπορούν να έχουν δυναμικό μήκος, π.χ. *1..5. Υποστηρίζει λειτουργίες φιλτραρίσματος, συγκεντρωτικών υπολογισμών και τροποποίησης δεδομένων μέσω εντολών όπως MATCH, WHERE, RETURN, CREATE και MERGE.

Το Neo4j χρησιμοποιεί εγγενή επεξεργασία γραφημάτων με index-free adjacency, όπου οι κόμβοι αποθηκεύουν άμεσες αναφορές στους γείτονές τους. Αυτή η αρχιτεκτονική επιτρέπει γρήγορη πλοήγηση χωρίς επιπλέον κόστος, διατηρώντας την απόδοση ανάλογη με το μέγεθος των δεδομένων.

1.4 Σχεδιασμός Συστήματος και Υλοποίηση

1.4.1 Υλοποίηση Καταλόγου Γράφου

Μία νέα κλάση καταλόγου αναπτύχθηκε για ενσωμάτωση του Neo4j, επιτρέποντας CRUD λειτουργίες μέσω Cypher queries. Το μοντέλο δεδομένων πετιλαμβάνει κόμβους που αναπαριστούν αντικείμενα του Feast, όπως Feature Views, Feature Services και Data Sources, περιέχοντας μεταδεδομένα όπως ονόματα και χρονικές σφραγίδες. Παράλληλα ορίζονται σχέσεις οι οποίες καταγράφουν ρητά τις εξαρτήσεις και αλληλεπιδράσεις μεταξύ αντικειμένων. Κύριες σχέσεις περιλαμβάνουν:

- HAS: Συνδέει Feature Views και Data Sources με τα Fields.
- POPULATED_FROM: Δηλώνει την προέλευση των Feature Views.
- PRODUCES: Προσδιορίζει τα χαρακτηριστικά εξόδου των On-Demand Feature Views.
- SERVES: Καθορίζει ποια Fields εκτίθενται μέσω Feature Services.

Κάθε αντικείμενο του Feast (Data Source, Feature View, On-Demand Feature View, Feature Service) ακολουθεί συγκεκριμένα βήματα καταχώρησης που δημιουργούν σχέσεις στο Neo4j και εκτελούνται μέσω της εντολής `feast apply`. Όταν πραγματοποιούνται αλλαγές, ο κατάλογος ενημερώνει τις συνδέσεις για τη διατήρηση της συνοχής και την ιχνηλασιμότητα των χαρακτηριστικών.

1.4.2 Ανίχνευση Εξαρτήσεων

Μια νέα λειτουργικότητα που προστέθηκε στο σύστημα είναι η ανίχνευση εξαρτήσεων μεταξύ εισόδων και εξόδων στις συναρτήσεις μετασχηματισμού των on-demand προβολών χαρακτηριστικών. Αυτές οι προβολές περιλαμβάνουν συναρτήσεις μετασχηματισμού που δημιουργούν νέα χαρακτηριστικά μέσω Pandas DataFrames ή Python dictionaries. Η χαρτογράφηση των εξαρτήσεων βασίζεται στην ανάλυση της αφηρημένης συντακτικής δομής (AST) του κώδικα, καταγράφοντας τις σχέσεις μεταξύ εισόδων και εξόδων. Η υλοποίηση αυτή υποστηρίζει μόνο βασικές λειτουργίες ανάθεσης και απλούς υπολογισμούς, ενώ πιο σύνθετες δομές, όπως βρόχοι και συνθήκες, δεν καλύπτονται ακόμα.

1.4.3 Εμπλουτισμός CLI

Νέες εντολές προστέθηκαν στη CLI του Feast, επιτρέποντας την εύκολη αλληλεπίδραση με το Neo4j και την ανάλυση σχέσεων μεταξύ χαρακτηριστικών. Όλες οι σχετικές εντολές ανήκουν στην ομάδα feast graph. Συνοπτικά, οι εντολές που υλοποιήθηκαν περιγράφονται παρακάτω:

- `execute-query`: Εκτελεί προσαρμοσμένα ερωτήματα Cypher στον κατάλογο χαρακτηριστικών.
- `most-used`: Εμφανίζει τα πιο συχνά χρησιμοποιούμενα αντικείμενα στη βάση χαρακτηριστικών.
- `most-dependencies`: Εντοπίζει τις πιο κοινές εξαρτήσεις στις on-demand προβολές χαρακτηριστικών.
- `served-by`: Προσδιορίζει ποια feature services περιλαμβάνουν συγκεκριμένα χαρακτηριστικά.
- `upstream-impact`: Δείχνει όλα τα αντικείμενα που εξαρτώνται από μια συγκεκριμένη πηγή δεδομένων, βοηθώντας στη διαχείριση αλλαγών. Αυτή η εντολή χρησιμοποιεί έξυπνη αναζήτηση διαδρομών στο Neo4j για να εντοπίσει εξαρτήσεις έως και τέσσερα επίπεδα βάθους.
- `common-tags`: Ομαδοποιεί αντικείμενα με βάση κοινές ετικέτες (tags) για εύκολη κατηγοριοποίηση.
- `common-owner`: Προβάλλει όλα τα αντικείμενα που διαχειρίζεται ένας συγκεκριμένος χρήστης, διευκολύνοντας τη διακυβέρνηση και την ιχνηλασιμότητα.

Αυτές οι βελτιώσεις καθιστούν την ανάλυση των εξαρτήσεων και των σχέσεων των χαρακτηριστικών πιο εύκολη, προσφέροντας μεγαλύτερη διαφάνεια και έλεγχο στην αποθήκη χαρακτηριστικών.

1.5 Αξιολόγηση και Σύγκριση Καταλόγων

Εκτελέστηκαν πειράματα με File, SQL και Neo4j καταλόγους, αυξάνοντας σταδιακά το μέγεθος της αποθήκης χαρακτηριστικών για να μετρηθεί η απόδοση κάθε συστήματος. Τα συμπεράσματα που προέκυψαν είναι τα εξής:

- Απόδοση του Feast Apply: Η εντολή `feast apply` ήταν πιο αργή στη βάση γραφημάτων λόγω του κόστους δημιουργίας κόμβων και σχέσεων. Ωστόσο, σε πραγματικά σενάρια με σταδιακές ενημερώσεις, αυτός ο αρχικός χρόνος δεν αποτελεί σημαντικό πρόβλημα.

- Εκτέλεση Εντολών: Ο κατάλογος γραφου ξεπέρασε τα άλλα συστήματα σε πολύπλοκες ερωτήσεις πολλαπλών επιπέδων, όπως `upstream-impact`, λόγω της αποδοτικής διαχείρισης σχέσεων. Το ίδιο συνέβη και για το ερώτημα `served-by` που αξιοποιεί τη σχέση SERVES, η οποία υφίσταται μόνο σε αυτό τον τύπο καταλόγου. Αντίθετα, απλά ερωτήματα όπως `most-used fields` ήταν ταχύτερα σε SQL λόγω χαμηλότερης υπολογιστικής επιβάρυνσης.

- Πλεονεκτήματα του Καταλόγου Γράφου: Το Neo4j επιτρέπει την άμεση μοντελοποίηση των σχέσεων, διευκολύνοντας την ανάλυση εξαρτήσεων. Η γλώσσα Cypher επιτρέπει αποδοτικές αναζητήσεις σε μεγάλο βάθος, καθιστώντας τη λύση ιδανική για ανάλυση αλληλεξαρτήσεων χαρακτηριστικών σε σύνθετες διαδικασίες μηχανικής μάθησης.

Αν και ο κατάλογος γράφου είναι εξαιρετικός για σύνθετες αναζητήσεις, έχει υψηλότερο αρχικό κόστος και δεν είναι ιδανικός για απλές ανακτήσεις δεδομένων. Για μικρής κλίμακας εφαρμογές, ο κατάλογος βασισμένος σε αρχείο είναι η καλύτερη επιλογή λόγω απλότητας. Αντιθέτως, για μεγάλα σύνολα δεδομένων με πολλές σχέσεις, ο κατάλογος γράφου είναι η ιδανική επιλογή για την αποτύπωση και ανάλυση εξαρτήσεων. Μια υβριδική προσέγγιση με SQL και γραφήματα θα μπορούσε να βελτιώσει την απόδοση σε μεικτά φορτία εργασίας μεσαίας-μεγάλης κλίμακας.

Chapter 2

Introduction

2.1 Background and Motivation

The quick development of machine learning (ML) applications across diverse production environments has emphasized the significance of data management and feature engineering. In modern ML workflows, features —quantitative or qualitative attributes derived from raw data— form the fundamental inputs for model training and prediction. However, the sheer size and complexity of modern datasets, potentially compiled from heterogenous sources, pose significant difficulties to feature engineering, the critical step of designing and refining attributes to effectively represent observations for ML algorithms [4]. As models and pipelines grow more intricate and data inputs diversify, the task of coordinating these features in a consistent, scalable and sustainable way has emerged as a substantial challenge.

Centralized repositories known as feature stores have emerged to facilitate the storage, administration and delivery of features throughout the ML lifecycle [5]. They not only enable reproducible feature engineering practices but also ensure consistency between the training and inference phases. There are several proprietary feature store solutions in the market, however we were able to study and analyze only the open-source options. Many of them have traditionally relied on SQL databases to store feature metadata and lineage information. While these systems have proven effective in simpler environments, they may encounter difficulties when attempting to capture and query the complex interdependencies that naturally occur in large-scale, dynamic settings.

Graph databases offer a potential remedy to this issue, given that they are inherently designed to model and traverse complex relationships, providing a powerful means to represent and search among interrelated data. Considering the potential of this approach, the present study explores whether a graph-based registry can offer enhanced, efficient and accurate modeling capabilities for feature stores, which is necessary to answer dependency-related questions. Specifically, the study investigates whether a graph database can provide a system that facilitates multi-hop traversal of links and fine-grained dependency tracking, thereby overcoming the drawbacks of conventional registries.

2.2 Objectives and Contributions of the Thesis

The primary objectives of this thesis are to:

- Develop a Neo4j-backed registry for Feast that leverages graph structures to capture intricate relationships and lineage among features.
- Extend Feast's CLI with relationship-focused commands to facilitate the querying and analysis of entity and feature dependencies.
- Conduct a quantitative performance evaluation comparing the graph-based registry with traditional file-based and SQL-based registries, focusing on execution times and scalability.

- Analyze trade-offs between registry implementations, particularly how each performs under various query patterns and scales.

This research is significant for a number of reasons. Managing feature interdependencies is essential for maintaining data integrity and model performance as machine learning systems grow and incorporate more complicated data sources. This work advances the field of feature store architectures, contributing to both academic understanding and practical applications in industry. By delivering empirical evidence on the benefits and drawbacks of a graph-based registry, it addresses a gap in the literature and provides insights into how a graph database might be used to more naturally model complicated feature interactions and lineage. Until now, graph databases, and Neo4j in particular, have only been integrated with feature stores as a data source [6, 7, 8]. Furthermore, by further developing Feast, an open-source feature store, our research not only capitalizes on the advantages of community-driven development but also offers a flexible framework that can be expanded upon or modified for use with proprietary systems.

2.3 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 3**: Provides an overview of feature store capabilities and architecture, focusing on the Feast open-source feature store.
- **Chapter 4**: Describes graph databases, highlighting Neo4j and its framework.
- **Chapter 5**: Outlines the implementation approach.
- **Chapter 6**: Details the experimental setup and findings from comparing different Feast registries.
- **Chapter 7**: Concludes the thesis, summarizing the work and suggesting directions for future research.

Chapter 3

Feature Stores in Machine Learning

3.1 Definition of Feature Stores

A feature store is a specialized data management system created exclusively for machine learning systems. Its primary purpose is to simplify and improve the management, storage and serving of feature data, which are needed by machine learning models to make predictions. Feature stores serve as a hub for features and metadata that are important for model training and inference tasks, assuring consistency, reusability and scalability across ML workflows [1].

Some feature stores support the execution of data pipelines that transform raw data into valuable feature values, while others only accept precomputed features as input. After this step, the feature store acts as a repository for storing both historical and real-time feature data. It guarantees that training datasets are built so that training samples do not contain feature values from the future, avoiding future data leakage. Furthermore, the feature store organizes feature data in a way that allows easy exploration, maintenance, monitoring and reuse by different teams and models within an organization. A feature store offers efficient interfaces for accessing feature data through various methods, either to generate large training datasets or recent values for ML applications to make predictions based on up-to-date information.

3.1.1 Components of Feature Stores

According to the Feast and Tecton documentation [9, 10], there are five core components of a modern feature store, also shown in Fig. 3.1:

1. Transformation: Feature stores organize data transformations that produce feature values from unprocessed data and, additionally, consume values produced by external systems. The transformations managed by a feature store are described in the feature registry and may be utilized across development, testing and production environments. This common approach eliminates the need of rewriting definitions in code and precludes training-serving skew. The principal transformation types encountered in feature stores are presented below:

    - Batch Transformation: Applied to data at rest, usually stored in a data warehouse, data lake or database.
    - Streaming Transformation: Applied to data from streaming sources, such as Kafka.
    - On-demand Transformation: Applied to data that are available only at the time of inference and cannot be computed in advance, usually in user-facing applications.

2. Storage: Feature stores facilitate both online and offline feature retrieval, therefore various storage layers are incorporated to fulfill different requirements. Older data and feature values, that may be used for training, are commonly stored in external data warehouses or data lakes. The latest feature values, which need to be accessed with low

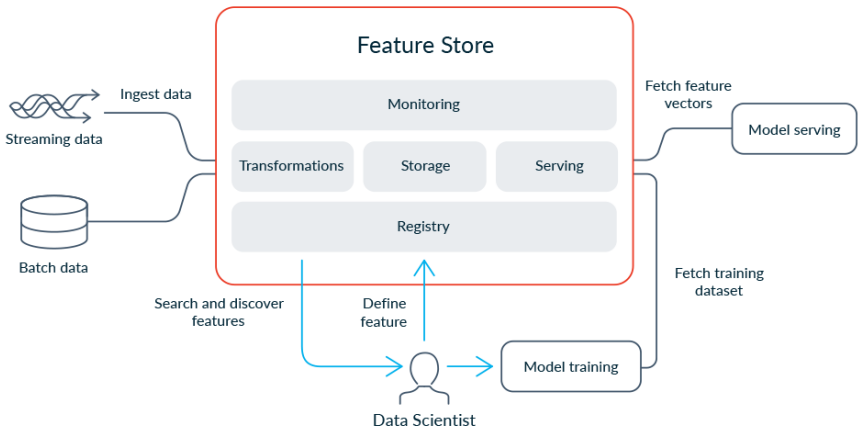


Figure 3.1: Feature Store components.

latency during inference, also persisted in the online storage layer, usually implemented with key-value stores.

3. Serving: Feature stores are responsible for fetching and feeding data to predictive models in production, while guaranteeing a consistent view across training and serving. In other words, the definitions of features and the transformations applied to them must be identical between the versions used to train a model and for real-time predictions in production. In case of discrepancies, the model's credibility and performance is compromised due to training-serving skew.

4. Monitoring: Feature stores have access to underlying data sources and constitute an intermediary stage for serving features to models, therefore they can calculate metrics on the precision and quality of these features. Monitoring these measured properties may provide meaningful insight into the health of an entire ML application. Another way to manage data quality is by comparing the data served to models with the data on which the model was trained to detect inconsistencies that could degrade model performance. Not all feature stores implement such monitoring internally, however they may provide the necessary values to existing monitoring infrastructure through appropriate interfaces.

5. Feature Registry: Feature stores organize feature definitions and metadata in a common catalog referred to as the feature registry. In brief, it serves as the single source of truth for information about feature groups, sources and services, which forms the basis for feature store system behavior. The feature registry interface allows users to explore, publish or collaborate on features, while simultaneously supplying potential automated jobs and serving APIs with information regarding which feature values should be available, who should be able to access them and how they should be served. Any additional metadata stored with the definitions in the feature registry may be useful in versioning, as well as tracking ownership and lineage.

3.2 Open-source feature stores

In this section, we explore different open-source feature stores that highlight various approaches to managing the machine learning lifecycle, with a focus on data lineage. Platforms like Hopsworks, Featureform and Feathr offer unique capabilities for tracking feature provenance, simplifying collaboration and scaling feature management across teams and projects. Before diving into Feast, which forms the core of this thesis, we will briefly compare these platforms to understand their contributions to the evolving landscape of feature stores.

3.2.1 Hopsworks

Hopsworks is a versatile and modular MLOps platform that provides a full suite of tools for managing the entire machine learning lifecycle [11]. The feature store, which may be used as a standalone component, provides its primary functionality by allowing teams to efficiently manage and serve features, promoting improved reuse and consistency across ML models.

Hopsworks also serves as a robust data science and engineering platform, enabling the creation of feature engineering and training pipelines. It offers collaboration, version control and workflow orchestration capabilities, which make it easier for teams to produce and share machine learning content.

Beyond the feature store and feature engineering, Hopsworks supports model management and deployment. This includes a model registry for organizing models and tools for monitoring their performance once deployed. Additionally, the platform integrates a vector database, enabling advanced tasks like similarity searches, which are often critical for recommendation systems.

Hopsworks feature store allows users to track provenance (lineage) through its UI and special endpoints, which provide intel on what features are used in which feature view or training dataset as well as what training dataset was used to train a given model. An example from the Hopsworks UI is pictured in Fig. 3.2. According to Hopsworks Research Papers [12, 13], the necessary metadata to track lineage are stored in a MySQL Cluster (NDB).

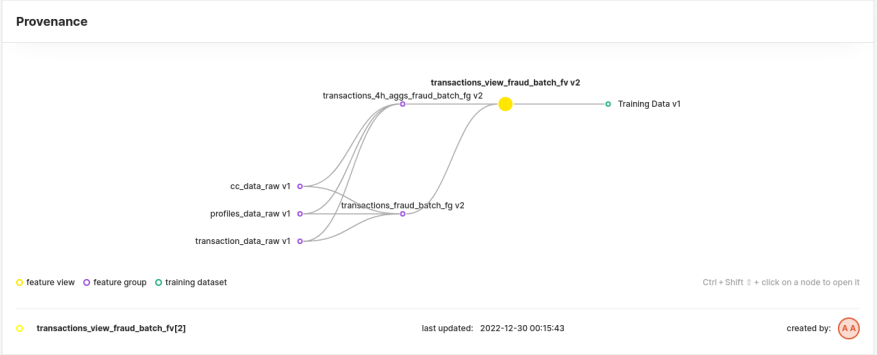


Figure 3.2: Lineage in Hopsworks UI.

3.2.2 Feathr

Feathr is a data and AI engineering platform that was developed and widely used at LinkedIn, before becoming open-source in 2022. Its architecture revolves around providing a cohesive platform where data scientists and engineers can define, share and reuse features across ML projects at scale. Two of the core components of Feathr are its Feature Registry and UI, which allow users to track feature metadata, explore data lineage and manage access control in a collaborative environment [14].

Users can browse and explore available features with ease thanks to the Feathr UI. They may quickly discover data sources, examine feature details, and trace feature lineage across projects and individual features with this intuitive and user-friendly platform, as seen in Fig. 3.3. By graphically representing feature pipelines, Feathr helps users understand the entire lifecycle of a feature—from its raw data source to its final transformation—enhancing transparency and traceability.

The Feathr Registry, which can be backed by Azure Purview or a SQL-based system, is essential for tracking feature information and managing metadata. This registry is a central repository where feature metadata, lineage and access control information are stored. By leveraging the registry, Feathr enables teams to efficiently share and reuse features across different projects, thereby eliminating redundant feature engineering tasks.

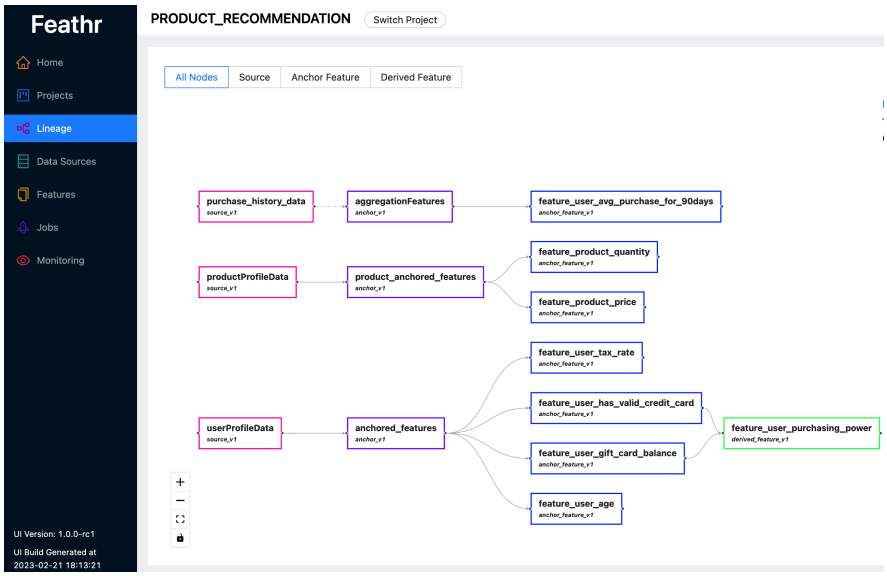


Figure 3.3: Lineage in Feathr UI.

3.2.3 Featureform

Featureform, as mentioned, is a virtual feature store, that orchestrates and manages feature engineering and serving pipelines without the need for additional infrastructure. It remains agnostic to the working details of existing data infrastructure, yet it provides a generalized framework to transform it into a feature store and ensure the seamless management of feature resources and transformations.

Featureform establishes a Directed Acyclic Graph (DAG) to map connections between resources such as features, transformations, labels and training sets [15]. Users can get a thorough understanding of the entire feature pipeline by exploring this DAG via the CLI and Featureform dashboard. Teams can use this to not only track the origins of features but also to comprehend how features change with time.

Additionally, Featureform offers search and discovery capabilities to support feature reuse across models and teams. Through the feature registry UI and CLI commands, users can examine feature lineage, track versions and ensure immutability. With custom tags and an intuitive interface, Featureform enhances visibility, enabling data scientists to confidently manage and monitor feature dependencies. From a more technical standpoint, all details regarding Featureform's state are stored in key-value stores or Postgres databases, which act as the single source of truth and feed the feature registry interfaces with accurate data.

3.3 Feast

Feast (**Fea**ture **St**ore) is a highly configurable operational data system that manages and delivers machine learning features to real-time models by reusing current infrastructure. The architecture of Feast aims to be both adaptable and scalable. It is constructed of a number of components that cooperate to create a feature store that may be utilized to consistently offer features for both inference and training. Feast aids data scientists in preventing data loss, allowing them to concentrate on feature engineering, by producing point-in-time accurate feature sets. It also establishes a single data access layer that separates feature storage from feature retrieval, allowing ML teams to decouple machine learning from data infrastructure and ensuring that models are portable when switching between batch and real-time models, training and serving models and data infrastructure systems. A high level diagram of Feast's funtionality is depicted in Fig. 3.4, which will be analyzed further in the following sections.

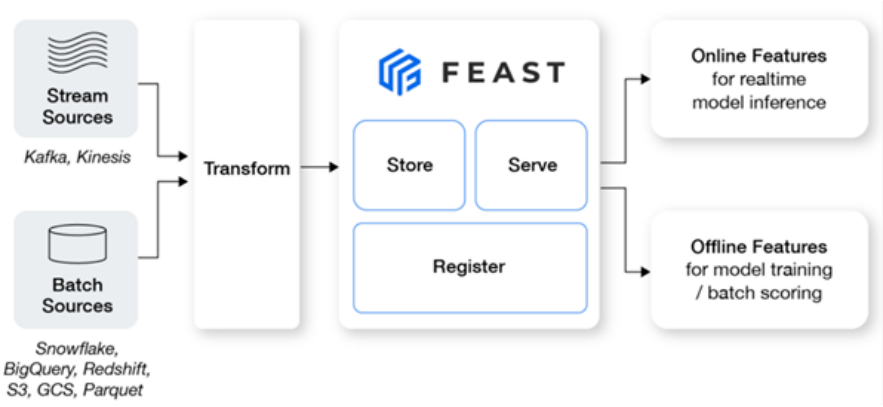


Figure 3.4: Feast components.

3.3.1 Offline store

The offline store stores previously computed feature values, which are called historical values and are used for model training and batch inference. A complete history of feature records is maintained, allowing data scientists to analyze data at a specific point in time or time period, identify trends and discover insights. Typically, the offline store is based on a data warehouse, such as Snowflake, Google BigQuery, a database, like PostgreSQL, Cassandra and MySQL, or a distributed file system [16, 17].

In Feast, an offline store is the interface for interacting with time-series feature values that are stored in data sources. There are currently four main offline store implementations with corresponding storage and compute engines: File, BigQuery, Snowflake and Redshift. Only one offline store may be configured at a time, meaning that features can only be ingested from data sources compatible with the selected store.

The core functionalities provided by the offline store are:

- Point-in-time correct joins to retrieve historical features and create training datasets. This is achieved through the get_historical_features method.
- Retrieval of the latest feature values for materialization into the online store. Feature values need to be loaded from the offline to the online store, in order to become available for low latency serving. This is achieved through the pull_latest_from_table_or_query method.

3.3.2 Online Store

The online store in a feature store only contains the latest feature values, which are used for real-time or near-real-time predictions. It is usually based on a row-oriented database or key-value store, optimized for low latency and high availability. It is important that the online store can fulfill many concurrent requests in a short time frame, while remaining accessible to ensure that precomputed features can be retrieved without interruptions or downtime [16, 17].

In Feast, the online store may hold materialized feature values from the offline store or real-time data from streaming sources directly. Data can be written to the online store through the online_write_batch method and retrieved from it through the online_read method. There are currently five main online store implementations: Sqlite, Redis, DynamoDB, Snowflake and Datastore.

3.3.3 Batch Materialization Engine

Data from the offline store is loaded and made available to the online store by the Batch Materialization Engine in Feast. A materialization engine typically abstracts over certain frameworks or technologies that are utilized to materialize data. By default, Feast pulls data from the offline store and writes it to the online store using a local in-process engine that is based on a pure local serialized method. However, this method is not scalable for big data sets since it is executed on a single process. To make the materialization process more scalable and allow for the delegation of materialization to distinct components, additional infrastructure can be employed. For instance, users may utilize AWS Lambda, a serverless computing service, to perform distributed materialization or the Bytewax stream processing framework for more scalable data movement.

The materialization process is typically triggered by running the `feast materialize` command, which is explained in detail in 3.3.6, either manually or through a scheduled job.

3.3.4 Feature Registry

Feast maintains a single feature registry to store all defined Feast objects, such as feature views and entities. Methods to apply, list, retrieve and delete these objects are implemented and exposed by the registry. The registry is updated during various Feast operations, including when applying changes via the Feast CLI or during metadata updates triggered by actions like materialization.

Feast by default employs a file-based registry, where the contents of the registry are serialized into a file in protobuf format. This file can be stored locally or on cloud storage platforms like S3, GCS or Azure. However, a file-based registry has certain drawbacks. For instance, in case a single field in the registry gets changed, a complete rewrite of the entire registry file is required, which can cause bottlenecks. This is particularly problematic during operations like materialization across multiple feature views or time slices, where concurrent changes must be serialized.

As an alternative option, Feast offers an SQL-based registry. This implementation allows for atomic updates to individual objects by storing the registry in a relational database. The SQL Registry utilizes SQLAlchemy, which allows it to support any database that is compatible with SQLAlchemy. Testing and out-of-the-box support are offered for databases like PostgreSQL MySQL and SQLite.

3.3.5 Feast Objects

A project is the top-level organizational unit within the Feast feature store. Projects provide complete isolation at the infrastructure level, meaning that each project operates independently from others.

The basic objects in a Feast project are described in more detail in the following sections, according to the Feast documentation [2].

Data Source

A data source in Feast refers to raw underlying data, stored in external systems and managed by the users. Feast is responsible for ingesting this data and applying the defined operations to retrieve or serve features. Data are represented in a time-series data model, a set of data points accompanied by the respective timestamps that are ordered in time.

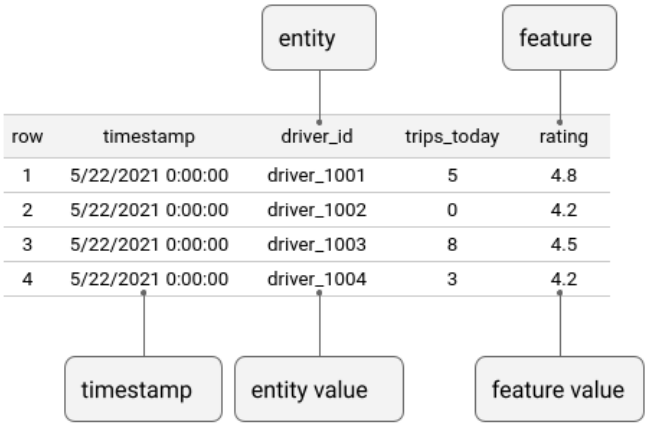


Figure 3.5: Data source example.

For completeness purposes, data source categories are listed below:

- Batch data sources: Batch data sources are associated with corresponding offline stores. These live in data warehouses (BigQuery, Snowflake, Redshift), data lakes (S3, GCS) or local storage.
- Stream data sources: Feast offers some classes and methods that make streaming features available in different environments. Sources with this functionality must also have a batch source specified, to retrieve historical features. There are two kinds of sources:
  1. Push sources, which provide functionality to push feature values both to the online store and offline store in real time, making them immediately available to applications. When data is pushed to a push source, Feast forwards the fresh feature values to all consuming feature views.
  2. Stream sources, which are still an experimental feature, allow Kafka or Kinesis streams to be registered as data sources. In this case, users are responsible for establishing and monitoring their own ingestion jobs, which are then used to write feature values to the online store, while writing to the offline store is also supported.
- Request data sources: Request data sources are currently utilized in conjunction with on demand feature views, to provide data that are only available at request time.

Entity

An entity is a set of semantically connected features. Users create entities that relate to the domain of their use case. For example, a ride-hailing service's entities could be customers and drivers, with similar features grouped together.

The information needed to define an entity is: the entity name, which serves as a unique identifier and the join key, which is used to specify the physical primary key on which feature values should be combined in order to be fetched during feature retrieval.

Feature View

A feature view is Feast's foremost object for defining features. It is defined as a logical group of time-series feature data, that is retrieved directly from a data source. The main components of a feature view are:

- A data source.
- One or more entities, if the feature view models features that are properties of specific objects.
- A schema, which lists one or more features. If the schema is not specified, Feast will infer the features from the columns of the underlying data source. The names and data types of the columns will be used as the names and types of the derived features.
- A description and a dictionary with metadata are optional parts of the feature view, however it is considered best-practice to provide as much information as possible for better documentation and future discovery of features.

Feature views allow Feast to represent data in a consistent way in both an offline and online environment, that are used for training and serving respectively. The applications of feature view are summarized below:

1. Producing training datasets by querying the data source of feature views to discover historical feature values. A single training dataset may include features from several feature views.
2. Loading feature values from either a batch or stream source into an online store. Feature views determine the online store's storage schema.
3. Retrieving features from the online store. Feature views supply Feast with the schema definition so that it can look up features from the online store.

On-demand feature view

This is a newer kind of feature view, that is still tested and adapted, and its singularity lies in the fact that on demand feature views contain some lightweight feature transformations. They allow data scientists to use existing features from previously defined feature views in combination with data from request sources to transform and produce new features. These transformations may be defined using Pandas dataframes or native Python dictionaries and are executed in both the historical retrieval and online retrieval paths.

Feature Service

A feature service is an object that represents a logical set of features from one or more feature views. Feature services enable machine learning models to use features as necessary, giving users the ability to reference all or a subset of the features from each feature view. The scenarios where feature services are used correspond to the aforementioned applications of feature views, since they provide an alternative interface for grouping features. However, it is recommended to construct one feature service per model version, which will allow for tracking of the features that models employ.

3.3.6 Core Functionality

The diagram in Fig. 3.6 provides a high-level view of how Feast manages the lifecycle of machine learning features. By connecting offline and online stores, Feast ensures that both historical and real-time feature data can be efficiently retrieved for model training and serving. The platform utilizes the components and objects described above to automate complex processes like feature materialization, point-in-time correctness and real-time feature retrieval. In the following subsections, we will explore how Feast's elements work together to streamline feature management, ensuring consistency and scalability in machine learning pipelines.

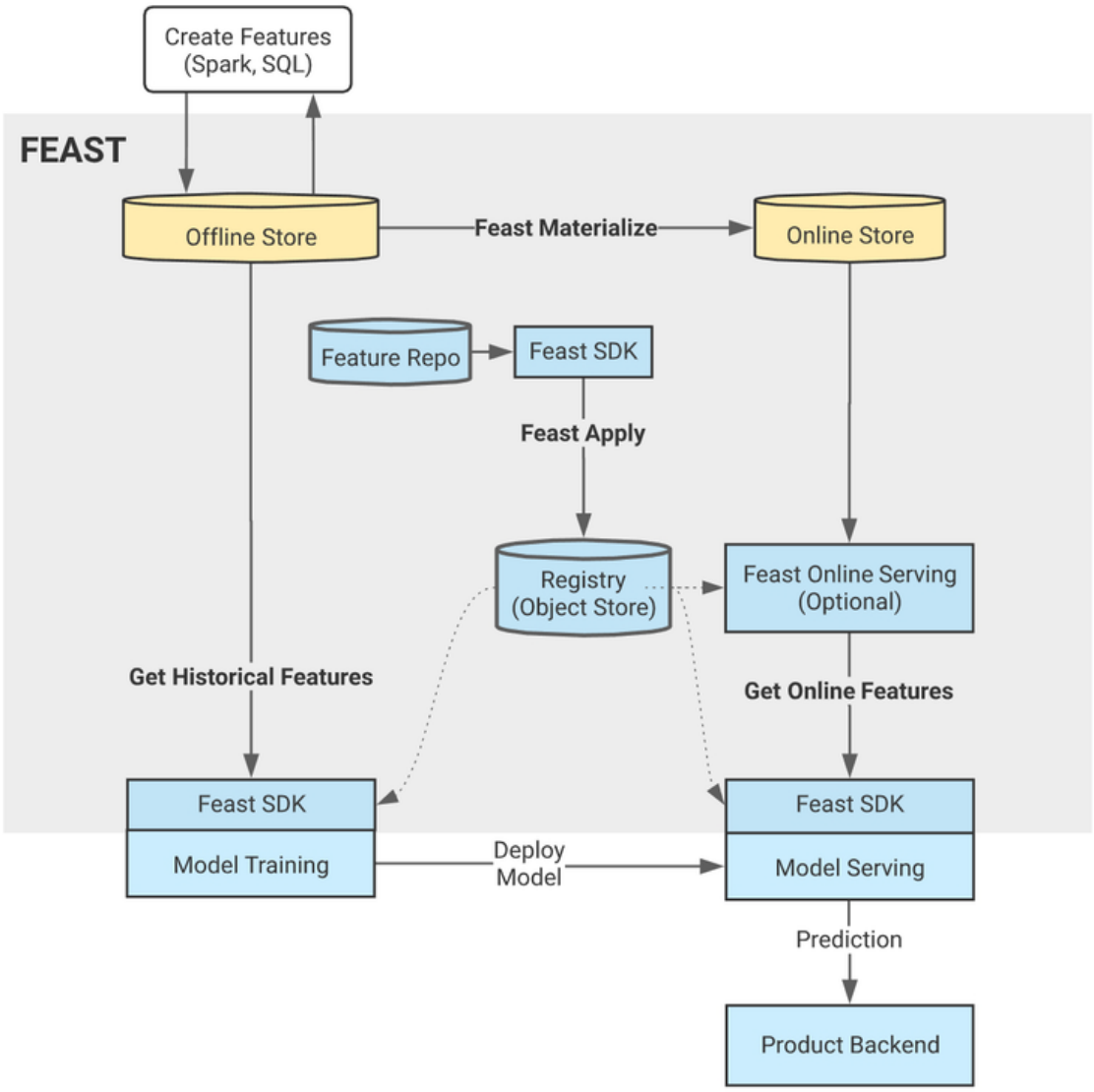


Figure 3.6: Feast architecture.

Feast Apply

The `feast apply` command is responsible for synchronizing the state of the Feast feature store with the definitions provided in the configuration files. It automates the process of registering and updating Feast objects and any associated infrastructure, such as online and offline stores. The way it works is by following the steps below:

1. First, Feast initializes a FeatureStore object, which is the main interface for interacting with the feature store. This object is initialized based on the configuration specified in the feature_store.yaml file in the current working directory. The feature_store.yaml file contains the infrastructure details, such as where the online and offline stores are located and what registry type to use.

2. Next, all Python files in the current directory are scanned for any definitions related to Entity, Data Source, Feature View or other relevant Feast objects. These definitions are written in Python and describe how the objects relate to one another, what features they concern and may also contain additional metadata. An example of such definitions is presented in Fig. 3.7.

```
driver = Entity(name=''driver'', join_keys=[''driver_id''])

driver_stats_fv = FeatureView(
    name=''driver_activity'',
    entities=[driver],
    schema=[
        Field(name=''trips_today'', dtype=Int64),
        Field(name=''rating'', dtype=Float32),
    ],
    source=BigQuerySource(
        table=''feast-oss.demo_data.driver_activity''
    )
)
```

Figure 3.7: Example of Feast object definitions.

   The definitions are read, validated and compared to the current state of the feature store, which is tracked in the feature store's registry. The comparison helps determine if there are any new objects, modifications to existing ones or objects that should be deleted. The objects are split based on their class. For each object type, the corresponding method in the registry is called to either register the object -in the case of updates or additions- or delete it -if it no longer exists.

3. Finally, Feast must also update the cloud infrastructure supporting the feature store to match the changes. This involves updating the online store, which serves real-time features to models, and potentially triggering the batch materialization engine, which processes batch data and writes it to the appropriate storage locations.

Feast Materialize

The `feast materialize` command loads feature data in the specified interval from either the specified feature views or all feature views into the online store, where it is available for online serving. The materialization process is orchestrated by the underlying batch materialization engine, which delegates individual steps to other components. The offline store is responsible for querying and pulling the desired feature values, which are mapped accordingly and then written to the online store in batches. Ultimately, the feature registry is updated with the latest materialized intervals for the affected feature views.

Get Historical Features – Get Online Features

Feast provides two key functionalities for retrieving feature data: get historical features and get online features. Both serve distinct use cases in the lifecycle of machine learning models

and address the need for feature consistency in different stages, from training to real-time inference.

- The get_historical_features API simplifies the process of conducting point-in-time joins when preparing features for training machine learning models. One of the main challenges in feature engineering is ensuring that the training dataset reflects the state of the world at the time when an event (such as a prediction or a transaction) occurred. This process, called point-in-time correctness, ensures that features used for model training do not leak future information that would not have been available at the prediction time. Feast automates this process by joining features from one or more feature views onto an entity dataframe and reproducing the historical state of these features through the necessary joins, allowing data scientists to obtain a training dataset that mirrors the conditions at the specified times.
- The online feature retrieval functionality of Feast addresses the need for real-time access to the most up-to-date feature values to make accurate predictions. The get_online_features API focuses on providing the latest feature values for a given entity or set of entities. Users provide a list of entities for which they want to retrieve features and Feast ensures that these latest feature values are available and can be retrieved from the online store.

Chapter 4

Graph Databases

4.1 Definition of Graph Databases

A graph database is an online database management system that supports Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph databases use nodes and edges to represent data rather than tables, as relational databases do, or documents and key-value pairs, as other NoSQL databases do.

Graph databases are designed to manage associative and contextual data, making it easy to represent and query complex relationships. They store nodes and relationships directly, which allows for quick traversal and scalability, while significant insights may be derived from the connections between data. Another asset of graph databases is their flexibility, since they can easily handle changes in data structures, making them ideal for handling data that are susceptible to change [18].

4.1.1 Comparison with Relational Databases

Graph databases are fundamentally different from relational in terms of data storage and management, with each providing significant advantages depending on the complexity and nature of the relationships in the data.

Data Management and Performance

Relational databases organize data into rows and columns, with relationships between records expressed through primary and foreign keys, that link different tables. While this system is effective for structured data and has been the basis of software applications for years, it can become inconvenient when dealing with composite relationships and multi-hop queries. These queries often require joins between multiple tables, which can become complex and slow, causing performance bottlenecks [19].

On the contrary, graph databases store data as nodes and edges and they are optimized for such tasks. Traversing relationships directly through edges is much faster than performing table joins, especially for queries with multiple levels of connections. By leveraging graph theory, these databases perform exceptionally well in scenarios where relationships are as important as the entities themselves, allowing for faster queries in highly connected data settings [20].

Schema Flexibility

One of the key differences between relational and graph databases is their approach to schema. Relational databases enforce a strict, predefined schema, which makes them ideal for well-structured data that requires absolute integrity and consistency. The guarantee of Atomicity, Consistency, Isolation and Durability (ACID principles) assures that data transactions are reliable and fault-tolerant, which is critical for applications like financial systems or enterprise resource planning (ERP) software.

Graph databases, on the other hand, provide a far more flexible schema, where relationships between nodes can be updated or added over time without needing to reform the database structure completely. This adaptability is especially useful in circumstances where the data model is evolving, such as knowledge graphs, social networks or semantic searches. In these cases, the ability to adjust the schema without downtime or complex migrations makes graph databases a more suitable solution [21].

Use Cases

Graph databases excel in situations where relationships between data points are central to the analysis. They are also superior in applications where relationships are dynamic or ever-changing, due to their capacity to model and query intricate, interrelated data in an intuitive and efficient manner. For instance, recommendation engines, supply chain networks and fraud detection systems all benefit from the graph model's ability to quickly navigate relationships.

Relational databases are best suited for scenarios where the data model is stable and relationships between data points are minimal or well-defined. Their strong support for data consistency, through ACID compliance, makes them the go-to choice for traditional business applications, such as inventory management, financial systems or human resources, where structured data and transactional integrity are critical.

4.2 Graph Data Models

A set of conceptual tools used to define real-world objects to be modeled in the database and how they relate to one another is generally considered a data model. According to Codd [22], a data model is more precisely defined as a collection of:

- data structure types, which form the basis for any database that conforms to the model,
- query language and transformation operators, which may be applied to instances of these data types to retrieve data,
- general integrity rules, which standardize the compliant states of the database.

Graph data models aim to represent information using a graph-like structure and they differ from other data models in that they contain two types of information. One, the attributes of entities and relationships, and two, the connectivity structure that makes up the network itself. The two leading graph models supported by most commercial graph databases are the RDF model and the property graph model.

- The Resource Description Framework (RDF) was initially developed by the World Wide Web Consortium (W3C) as part of the Semantic Web initiative. It is an assertional language intended for representation of metadata on the Web, which can also be applied to portray any information in a graph structure, enabling data to be linked and shared across different domains. RDF is based on a structure called a "triple," which consists of three components: subject, predicate and object [23]. Each triple represents a single fact or statement. DBPedia and YAGO are two examples of projects that utilize RDF and they support the SPARQL query language, designed to retrieve and manipulate data stored in the RDF format. SPARQL queries match patterns in the RDF triples, making it highly effective for traversing linked datasets.
- The Property Graph Model is a more common and flexible approach to graph modeling, which represents entities as nodes and the relationships between them as edges. Vertices and edges may have multiple properties in the form of key-value pairs, that

describe their attributes [24]. They can also have labels that categorize them. Property graphs typically use query languages like Cypher (used by Neo4j) or Gremlin (used by Apache TinkerPop). These query languages are designed to navigate the graph structure efficiently by allowing traversal of nodes and edges while taking advantage of the properties stored in them.

In general, the property graph model can capture the same information as the RDF model, but often with fewer nodes and edges. This is because information that in RDF requires the addition of extra nodes or edges can be represented in the property graph model as a property of an existing node or edge [25]. This allows the complexity of the graph to be reduced, which can lead to more efficient storage and faster data retrieval.

4.3 Neo4j Overview

Neo4j, which stands for Network Exploration and Optimization 4 Java, is an extensive ecosystem containing numerous tools, applications and libraries based on Java and used for storing and managing data in a natural, connected state. The core product is a robust and scalable graph database, which has evolved into the industry leader in this sector. The engineers of Neo4j also created the Cypher query language, which enables intuitive traversal and pattern-matching of the complex, interrelated data that it can represent. Beyond its query language, Neo4j also offers a multitude of tools that enhance the developer's experience [3]:

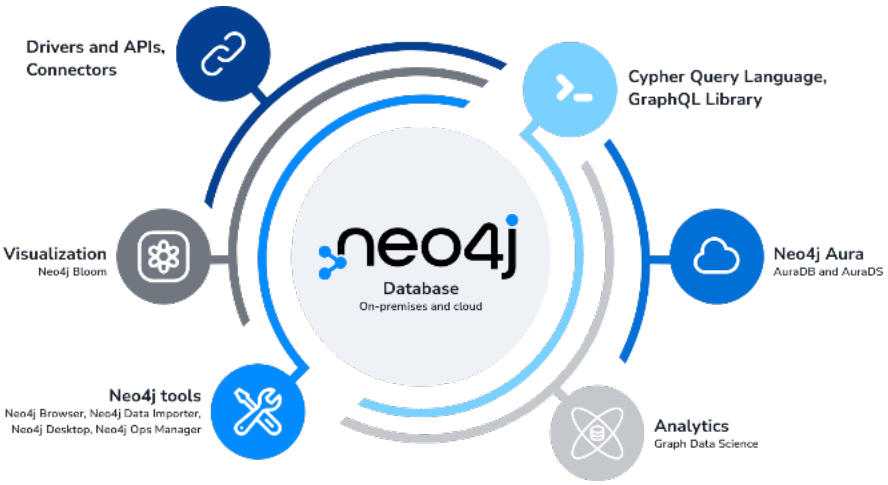


Figure 4.1: Overview of the Neo4j ecosystem.

- Neo4j Browser is a built-in tool that provides an interactive interface to manage and explore the graph database from any browser. Users can execute Cypher queries, immediately see their results as graph visualizations and interact directly with the database. It is the default interface for both Enterprise and Community Editions of the Neo4j database.
- Neo4j Bloom is a powerful graph visualization tool intended for business users, analysts and data scientists, who may not be familiar with Cypher. It allows users to query and interact with the graph visually to discover patterns, without requiring any programming skills. Thus, a business view of the data in the database can become accessible to a wider audience to analyze and extract insights.
- Neo4j Graph Data Science (GDS) offers more than 65 efficiently implemented, parallel versions of popular graph algorithms, such as community detection, centrality, node

similarity and pathfinding. They are exposed as Cypher procedures, ready to be executed with Neo4j and optimized for enterprise pipelines and workloads. GDS aids in analyzing large and complex networks to get insights from big data and answer critical questions. To handle graph problems like missing relationship prediction, GDS also provides machine learning pipelines for training predictive supervised models.

- Neo4j Aura is a fully managed, cloud-based graph platform designed to provide fast, scalable and highly available graph database services. Neo4j Aura is divided into two main offerings: AuraDB, which is geared towards developers creating intelligent, context-driven applications, and AuraDS, a tool aimed at data scientists working on machine learning and advanced analytics. AuraDB, the graph database as a service product, allows for importing, visualizing data and executing queries extremely quickly, using built-in developer tools and integrations, but without worrying about infrastructure. AuraDS, on the other hand, is the data science as a service solution that combines machine learning and graph database layers into a single workspace, simplifying the process of finding connections in big data and delivering answers to crucial business problems.

- Neo4j has developed an extensive library of connectors and drivers that allow it to be seamlessly incorporated into a variety of environments. Major programming languages including Python, JavaScript, Java,.NET, and Go have drivers available. The Neo4j Python driver lets developers interact with Neo4j directly from Python applications. This driver connects to a Neo4j database instance through a binary protocol called Bolt. It supports executing Cypher queries, managing transactions and handling connections to Neo4j, making it an essential component for Python-based projects that require graph-based data storage.

4.3.1 Data Model in Neo4j

Neo4j graph database is a NoSQL database, which is based on the property graph model [26], so it consists of:

- Nodes, which describe discrete objects in a domain. They are allowed to have zero or more labels to classify what kind of nodes they are.

- Relationships, which always have a direction and describe a connection between a source node and a target node. They are required to have a type, to define what kind of relationship they are.

Nodes and relationships can have properties, which further describe them and take the form of key-value pairs.

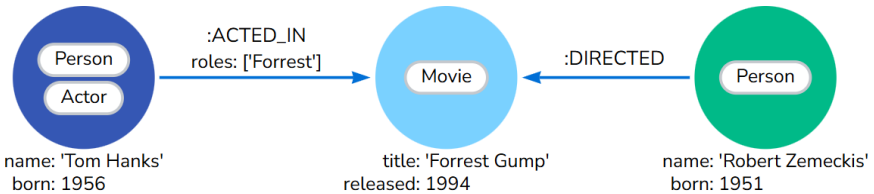


Figure 4.2: Simple example of Neo4j graph.

Fig. 4.2 shows a simple example of a graph as it would be represented in a Neo4j database. The first node has the labels "Person" and "Actor" and the properties "name" with value "Tom Hanks" and "born" with value 1956. There is an outgoing relationship from the "Tom Hanks" node to the "Forrest Gump" node of type ACTED_IN.

4.3.2 Cypher Query Language

The programming languages that are used to query and manipulate graph databases are known as graph query languages. They enable developers to quickly access and modify data stored in a graph structure and allow them to express intricate queries spanning several levels of connections in a straightforward manner. Graph query languages allow users to search for patterns or relationships by navigating through the nodes and edges in the database, therefore providing understanding of the data and the ability to make well-informed decisions [27].

Cypher was created in 2011 by Neo4j engineers as an SQL-equivalent language for graph databases. It allows for efficient and expressive queries, enabling users to realize the full potential of their property graph databases. Writing a query is effectively like drawing a pattern through the data in the graph. Graph patterns and operations can be expressed in a format that is straightforward and understandable, streamlining the process of working with complex interconnected data structures.

A key feature of Cypher is its expressive pattern-matching syntax, which uses visual representations to depict nodes and relationships. Nodes are enclosed in parentheses (), and relationships are represented by arrows –> or <– with optional labels and properties inside square brackets []. For example, a simple pattern to find actors in a movie might look like `(actor)-[:ACTED_IN]->(movie)`. This approach makes it easier to visualize and construct queries that mirror the structure of the graph itself.

Cypher lets users define the desired outcomes without getting entangled in the technical details of how to obtain them. This allows the database engine to optimize query execution while adhering to the principles of declarative languages. The language includes a variety of clauses such as MATCH for specifying patterns to search for, WHERE for filtering results based on conditions, RETURN for selecting the data to output and CREATE or MERGE for adding or updating nodes and relationships.

One of Cypher's strengths is its support for variable-length path queries, which lets users search for patterns that span an arbitrary number of relationships. This is particularly useful for traversing hierarchical data or any graph where the depth of connections is not fixed. For instance, the pattern `(start)-[:CONNECTED_TO*1..5]->(end)` finds paths between start and end nodes that are between one and five relationships long.

Cypher also provides aggregation functions like COUNT, SUM, AVG, MIN, and MAX, enabling users to perform statistical computations over the graph data directly within queries. Combined with grouping capabilities, these functions facilitate complex analytics without the need to extract data into external tools.

4.3.3 Architecture and Performance

Neo4j is a native graph database, which means that it is designed to store and process data as a graph. The underlying structure of the database is built specifically for storing graph-like data and ensures that data is stored effectively by utilizing storage patterns that place nodes and relationships in close proximity to one another. Every layer of Neo4j's architecture – from the Cypher query language runtime to the management of store files on disk – is optimized for storing and accessing graph data, and not a single component is built on top of other non-graph technologies.

Additionally, Neo4j uses index-free adjacency to provide native processing capabilities. This implies that nodes have direct references to their neighbors, so that retrieving relationships and related information is simply a memory pointer lookup [28]. Because of this, native graph processing time is not increased exponentially with the number of relationships traversed and hops navigated, instead native graph queries perform proportionally to the volume of data processed. With processing specifically built for graph datasets, relationships are exploited to maximize traversal performance as opposed to relying heavily on indexes for

joins.

Chapter 5

System Design and Implementation

5.1 Tools and Technologies Used

Various tools and libraries were utilized in this implementation to incorporate Neo4j as a graph-based feature registry and expand the capabilities of the Feast feature store. The tools used are broken down in depth below:

Python Programming Language

Python is a versatile, high-level programming language, that is easy to use, readable and has a large library ecosystem, making it the preferred choice for data science and machine learning systems [29].

Python was chosen for its compatibility with Feast, which is itself written primarily in Python. Its ability to use Object-Oriented Programming facilitated the development of a new class that extends the Feast registry. It also offers a broad library ecosystem that made it possible to integrate other tools like Click and the AST module, which sped up the project's development.

Feast Feature Store

Feast is an open-source feature store used to manage and serve machine learning features, which served as the foundation for the implementation. By using Feast, this application benefits from its existing ecosystem of feature views, data sources and entity management, while adding advanced capabilities such as querying feature dependencies using a graph database. More specifically, the implementation is based on a fork of Feast's v0.38-branch, which can be found here. Modifications were made to the Python SDK to allow for custom interactions with the new graph registry.

Neo4j Graph Database

Neo4j is the central database used for the graph-based feature registry, chosen for its efficient graph traversal and relationship management. The project offers flexible deployment options by supporting both a locally hosted Neo4j instance and Neo4j AuraDB, a cloud-based managed service.

Python Neo4j Driver

The Neo4j Python Driver offers a native interface to interact with Neo4j from Python programs. It enables developers to handle transactions, maintain sessions and run Cypher queries in a manner that seamlessly interacts with Python codebases.

- Cypher Query Language: The driver allowed Cypher queries to be executed smoothly from within the Python code. This made it possible to query the graph registry and insert or update data efficiently.

- Session and Transaction Handling: When adding or modifying big sets of nodes and relationships, the driver's integrated session management guaranteed safe and effective transactions. Transactions were crucial in preserving the graph's integrity during the updates and materialization of features.

Abstract Syntax Tree (AST) Module

Python's Abstract Syntax Tree (AST) module is a powerful tool for programmatically parsing and analyzing Python source code. It converts source code into a tree-like structure, representing the syntactic elements and their relationships [30]. The AST module was leveraged to automatically detect dependencies between features by parsing Python code used in on-demand feature view transformations. By analyzing these transformations, the AST enabled the system to extract input features that contributed to the calculation of output features. These dependencies were then mapped to Neo4j relationships, providing a clear representation of feature relationships in the graph database.

Click Module

Click is a Python package for creating command-line interfaces (CLI) with minimal code. It supports nested commands, automatic help generation, and argument handling, making it an ideal choice for building command-line tools [31].

Click was used to extend the Feast CLI with custom commands related to the graph-based registry. This made it easier for users to interact with the Neo4j database, query feature relationships and explore feature lineage directly from the command line.

5.2 Extending Feast with a Graph-Based Registry

In order to address the limitations inherent in traditional registry implementations, this work extends Feast with a graph-based registry that leverages the capabilities of Neo4j. The extension was motivated by the need to capture and query the complex interdependencies among feature store objects. This section describes the design decisions and integration steps we followed during the development of the graph registry within Feast.

To integrate the graph-based registry within Feast, we defined a new registry class along with a corresponding configuration specifically tailored for Neo4j. This integration involved introducing new methods to perform CRUD operations using Neo4j's transactional model, ensuring data consistency and integrity throughout all operations. These registry methods are used across the entire feature store and serve as the central mechanism for all interactions with the registry.

The first step to create and run a Feast project is to define its basic configuration on the feature_store.yaml file, which includes infrastructure configuration for the associated project. When working with a graph database registry, the section about it should look like the following:

```
registry:
    registry_type: graph
    uri: neo4j+s://55649b29.databases.neo4j.io:7687
    database: neo4j
    user: neo4j
    password: password
```

Creating an instance of Neo4j AuraDB or the Neo4j Database is a prerequisite to gather the following information, that is necessary to create and use a graph feature registry:

- The URL to connect to the Neo4j instance, including the port.
- The username and password for a user authorized with read and write privileges.
- The name of the database used for the queries.

5.3 Data Model and Schema Design

5.3.1 Node labels and properties

In the graph-based feature registry, nodes represent the core Feast objects that make up the data structure. Each node is assigned a specific label corresponding to its object type, and is enriched with properties that capture essential metadata, such as names and timestamps.

When a new project is initialized a new Project node is created, which forms the base for all other objects-nodes related to the specific project. It bears a unique project_id property.

As mentioned in subsection 3.3.6, the first time that the feast apply command is executed Feast object definitions are parsed and nodes corresponding to each object type such as a Field, Entity, FeatureView, OnDemandFeatureView, DataSource or FeatureService are created. These nodes adhere to the following rules:

- They contain a name and proto property, which are labeled according to the object's class. For instance, Field nodes have the properties field_name and field_proto. The value of the proto property is a protobuf representation of the object, which is used across the feature store by multiple operations.
- All nodes have a last_updated_timestamp property.
- If a description is specified, it is added as a node property.
- Field nodes have an additional data_type property.
- Feature Views have an additional materialization_intervals property.

5.3.2 Introduced Relationships

The interactions between nodes in the registry are represented by relationships such as CONTAINS, HAS, POPULATED_FROM, PRODUCES and USED_FOR. These relationships were chosen because they directly reflect the operational and logical interactions observed in the feature store. By explicitly modeling these connections, the registry supports efficient querying, provides an intuitive and interpretable representation of feature interdependencies and ultimately guarantees consistency and traceability of the data flow in the system. The relationships evolve as new feature views, on-demand feature views and feature services are introduced, or when modifications are made to existing entities.

CONTAINS Relationship: There is a CONTAINS relationship from the Project to any other node, which is essential to identify the elements of different workspaces and any potential overlap between different projects.

USES Relationship: Every time a new feature view is created or modified, a USES relationship is established from the FeatureView to its associated Entity nodes, ensuring that the feature view properly reflects which entities it depends on.

HAS Relationship: For each feature defined in the schema of a feature view, a HAS relationship connects the FeatureView to the Field node. This ensures that the features within the feature view are directly linked to their definitions in the registry. When

a request source is defined, it is mandatory to specify the features it contains, and a HAS relationship is established from the DataSource to the Field node. For other data source types, the features they contain are inferred from the related feature views, and a HAS relationship is established from the DataSource to the Field node.

POPULATED_FROM Relationship: If a stream data source is used, the FeatureView has a POPULATED_FROM relationship with the DataSource. Otherwise, the target of this relationship is the batch data source, ensuring that the origin of data for the feature view is accurately represented.

RETRIEVES_FROM Relationship: If a push data source is used, the push data source has a RETRIEVES_FROM relationship with the batch DataSource

BASED_ON Relationship: For on-demand feature views, the model introduces a BASED_ON relationship to depict its dependency on a base feature view or data source.

PRODUCES Relationship: For every feature defined in an on-demand feature view, there is a PRODUCES relationship from the OnDemandFeatureView to the Field node.

USED_FOR Relationship: An integral part of every on-demand feature view is a function-transformation applied to input data fields in order to compute the final features. This function's syntax is analyzed to infer dependencies between features. This process is described in more detail in section 5.4 and it results in a USED_FOR relationship from one Field node to another.

CONSUMES Relationship: When defining a feature service, it is necessary to list all associated feature views and on-demand feature views. This is captured by a CONSUMES relationship from the FeatureService to the relevant FeatureView and OnDemandFeatureView nodes, ensuring that the services are aware of which features they consume.

SERVES Relationship: When defining a feature service, there is an option to select only part of the features from each feature view and on-demand feature view. Therefore, a SERVES relationship from the FeatureService node is necessary to depict which Field nodes are actually served by it.

OWNS Relationship: An OWNS relationship is directed from an Owner node, which only has a name property, to either a DataSource, Entity, FeatureService, FeatureView or OnDemandFeatureView node.

TAG Relationship: This is the most versatile type of relationship, since it is created based on a dictionary of tags. DataSource, Entity, FeatureService, FeatureView and OnDemandFeatureView nodes may have TAG relationships, where the destination is determined by the key to each dictionary entry.

5.3.3 Schema Diagrams

A collection of schema sub-diagrams to assist in better understanding of the relationship between different nodes is provided below. The relationships that are formed or updated when a particular type of node is changed are shown in each diagram. As stated in subsection 3.3.6, these actions are performed because of the `feast apply` command.

Data Source

When changes to a Data Source are applied-registered, the registry is updated according to the following steps:

1. Initially, the connection to the current Project node is verified.
2. Optionally, if an owner is specified in the Data Source definition, the node is connected to the respective Owner node.
3. Another optional argument in the definition is a dictionary of tags. If it is present, for each entry in the dictionary the Data Source node is connected to a node with a label that matches the entry's key and a value property that matches the entry's value.
4. If the source's schema is specified, detailing the features it contains, relationships from the Data Source to Field nodes are inserted.
5. In the case of Push Sources, another Data Source is defined as the batch source, hence a RETRIEVES_FROM relationship is included in the registry.

These relationships are depicted in Fig. 5.1.

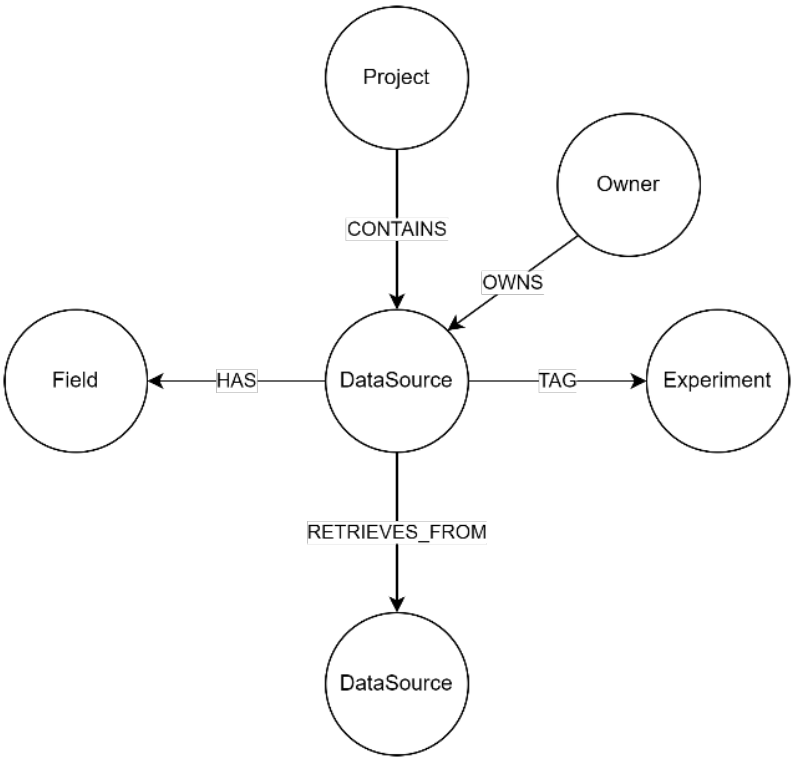


Figure 5.1: Relationships from Data Source registration.

Feature View

The feature registry is updated according to the following steps when modifications to a Feature View are applied-registered:

1. First, the connection to the current Project node is verified.
2. The node is linked to the appropriate Owner node, if an owner is mentioned in the Feature View specification.
3. Another optional argument in the definition is a dictionary of tags. If it is available, the Feature View node is connected to a node that has a label matching the entry's key and a value property matching the entry's value for every entry in the dictionary.

4. The Feature View node is linked to all the entities that it describes. If no entity is included in the definition, a relationship to the dummy Entity node is created instead.

5. Either a batch or a stream source must be filled in the Feature View, therefore a relationship is created from the Feature View node to the Data Source that it is populated from.

6. For every feature the Feature View contains, a HAS relationship from the Feature View to the Field node must be created in the graph. If a relationship from the Data Source is not already present in the registry, it is inserted at this stage.

These relationships are portrayed in Fig. 5.2.

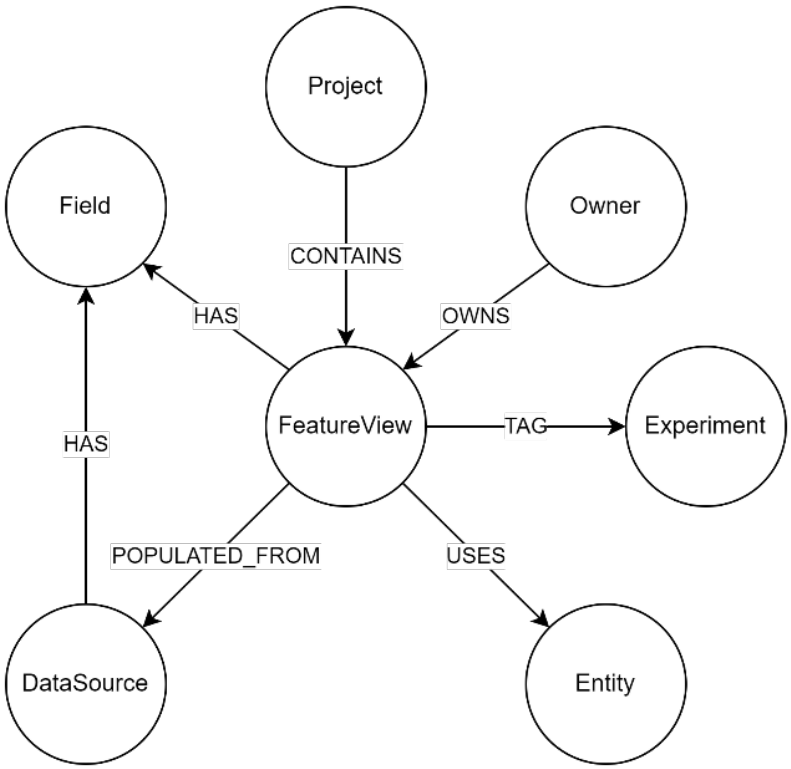


Figure 5.2: Relationships from Feature View registration.

On Demand Feature View

When changes to an On Demand Feature View are applied-registered, the registry is updated according to the following steps:

1. Initially, it is ensured that it is connected to the current Project node.

2. Optionally, if an owner is included in the On Demand Feature View definition, the node is linked to the respective Owner node.

3. A dictionary of tags is an additional optional argument in the definition. If it is present, for each entry in the dictionary the On Demand Feature View node is connected to a node with a label that matches the entry's key and a value property that matches the entry's value.

4. On Demand Feature View transformations may use features from Feature Views or Request Sources as input, therefore the On Demand Feature View node is linked with nodes of these two types.

5. The list of features in the output of the On Demand Feature View is used to create a set of PRODUCES relationships towards the respective Field nodes.

6. Additionally, dependencies between input and output features are calculated and represented as relationships between Fields in the graph.

These relationships are illustrated in Fig. 5.3.

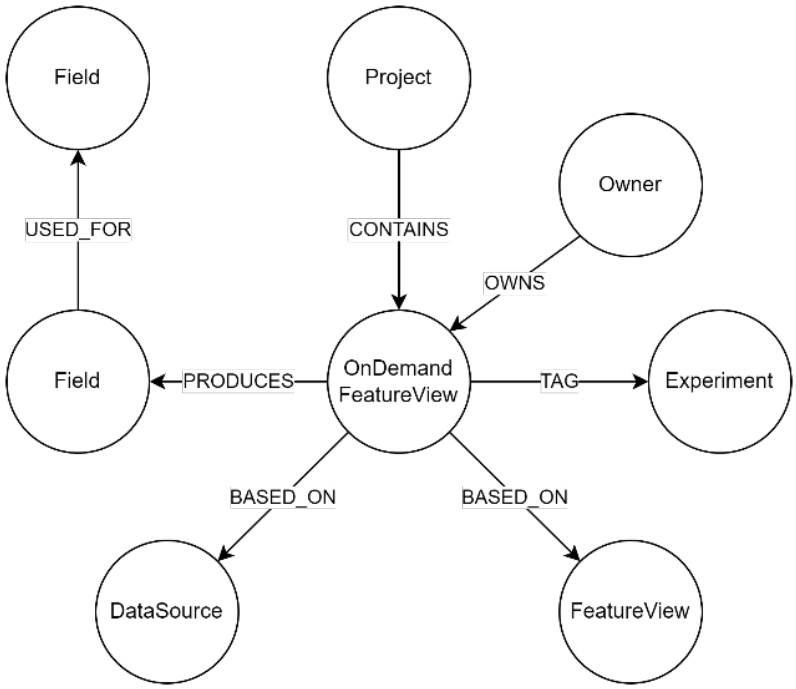


Figure 5.3: Relationships from On Demand Feature View registration.

Feature Service

The feature registry is updated according to the following steps when modifications to a Feature Service are applied-registered:

1. First, it is ensured that it is connected to the current Project node.

2. If an owner is specified in the Feature Service definition, the node is connected to the respective Owner node.

3. Another optional argument in the definition is a dictionary of tags. If it is present, for each entry in the dictionary the Feature Service node is linked to a node with a label matching the entry's key and a value property matching the entry's value.

4. Feature Services make a group of existing features available to be retrieved together during training or serving. There are SERVES relationships from the Feature Service node to all the Fields it involves.

5. These features may come from entire Feature Views, On Demand Feature Views or projections of these. To represent the of origin of the features being served, there is

a relationship from the Feature Service to any Feature View and On Demand Feature View it consumes features from.

These relationships are depicted in Fig. 5.4.

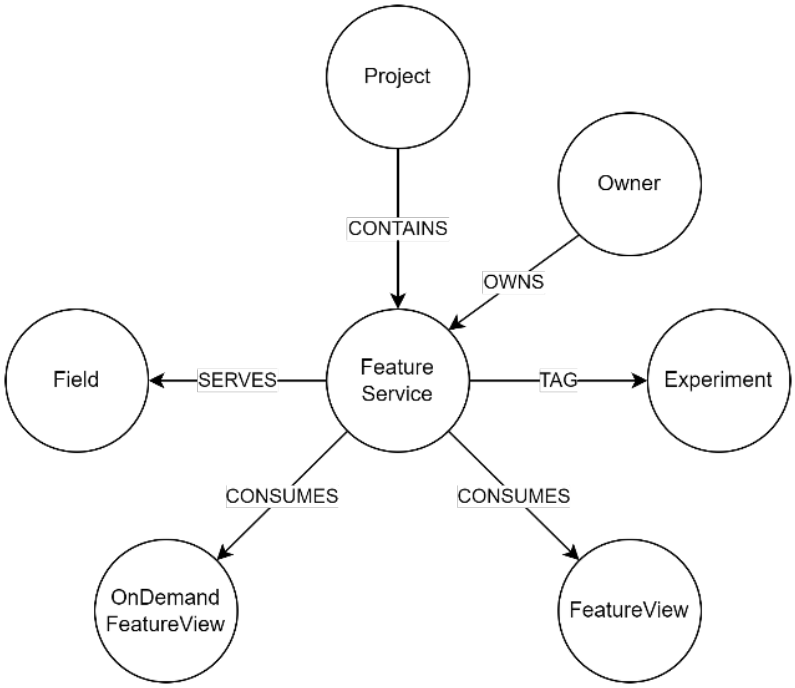


Figure 5.4: Relationships from Feature Service registration.

5.4 Tracking Dependencies and Lineage

On demand feature view definitions are accompanied by a transformation function that utilizes either Pandas DataFrames or Python dictionaries to create and calculate new features. New dependency tracking classes were defined focusing on tracking the dependencies between input and output dictionaries within these functions. These classes utilize Python's ast module to parse and traverse the abstract syntax tree (AST) of the code, identifying how the inputs and outputs are connected.

This implementation primarily focuses on straightforward assignment operations and basic manipulations, assuming a direct mapping between input and output elements. It does not yet handle more complex Python constructs, such as loops, conditionals, or deeply nested structures, which may involve more intricate relationships between inputs and outputs. Despite these limitations, the implementation serves as a foundational tool for tracking dependencies in relatively simple transformation functions, making it useful for understanding data flow in both Pandas-based and pure Python-based data processing tasks.

5.4.1 Pandas transformation

The added DependencyTracker class is responsible for analyzing a Python user-defined function (UDF) that transforms pandas DataFrames. The goal is to track how columns in the input DataFrame are used to produce columns in the output DataFrame. The process to achieve this can be separated in the following steps:

1. Determining the input and output DataFrames: In order to track which input DataFrame columns contribute to which output DataFrame columns, it is required to identify the

corresponding DataFrame names. The input DataFrame is extracted from the function's signature, while the output DataFrame name is inferred by inspecting the return statement of the UDF.

2. Tracking column dependencies: The tracking itself is done by parsing the UDF, extracting its representation as an AST and then visiting the nodes it contains one by one. After visiting all the nodes, the tracker builds a dictionary where the keys are the output DataFrame columns or other variables, and the values are sets of the input DataFrame columns or other variables that contribute to them. The types of nodes visited are explained in detail below:

   - Assignment nodes: The primary cases of assignment supported are DataFrame subscript assignment and temporary variable assignment. The first applies in cases when the output DataFrame is being assigned a value, the target column is stored, and the value being assigned is processed to capture any dependencies. Similarly, if a temporary variable is assigned a value, the variable's name is stored, and the assigned value is later processed.
   - Binary operation nodes: In cases where a binary operation is performed, both sides of the operation are visited, allowing the tracker to understand which input columns contribute to the result.
   - Subscript nodes: When the UDF accesses a DataFrame column via subscript notation (e.g. df["column_name"]), this method captures which column is being accessed.
   - Variable access nodes: If a temporary variable is used later in the UDF, it's expanded to capture its underlying dependencies. This ensures that even multi-step transformations are correctly tracked.
   - Function call nodes: Function calls on DataFrame columns are also tracked, by visiting the function and its arguments.

3. Resolving dependencies recursively: If the dependency map includes intermediate steps or temporary variables, a recursive resolution is performed so that the final result only maps output columns directly to input columns.

Since the described method is not exhaustive and it does not cover all AST node types, it is vital to ensure that the tracking process does not fail in cases of unexpected syntax or unsupported code structures. If an exception occurs during the tracking process, a warning is printed, and the method returns an empty dictionary. This way may result in missing relationships in the graph feature registry, but the main flow is not interrupted.

5.4.2 Python transformation

Similarly, for the Python transformation case the dependency tracking functionality focuses on analyzing a UDF that operates on dictionary-like inputs, with the goal of tracking how input dictionary keys contribute to the output dictionary keys. The overall steps correspond to the ones previously described, adjusted to dictionaries instead of Pandas Dataframes.

1. Determining the input and output dictionaries: The input dictionary is extracted from the UDF's function signature, while the output dictionary is identified by inspecting the return statement of the function.

2. Tracking dictionary key dependencies: The UDF is once again parsed into an AST and visited node by node. After visiting all the nodes, the tracker builds a dictionary where the keys are the output dictionary keys or other variables, and the values are sets of the

input dictionary keys or other variables that contribute to them. The types of nodes are the same as in the Pandas dependency tracking, with the addition of Dictionary nodes.

3. Resolving dependencies recursively: Once the AST is traversed, if intermediate variables or multi-step transformations are present in the dependency map, a recursive resolution process is applied to ensure that only direct relationships between input and output dictionary keys remain.

In order to prevent disruption in the transformation logic's main flow, a warning is logged and an empty dependency map is returned if the tracking process comes across unexpected syntax or unsupported structures.

5.5 CLI Enhancements

The graph-based feature registry is accessible through Neo4j-specific tools, such as the Neo4j Browser and Neo4j Bloom, which can be used to directly explore, query the database and visualize the results. Additionally, new commands have been introduced to the Feast CLI that allow users to interact with the registry and retrieve results in a tabular format.

All graph-related commands are grouped under the feast graph command and can be executed from the feature store repository.

```
feast graph [command] [options]
```

The available commands with an overview of their functionality are presented below:

- `execute-query`: Execute a raw Cypher query on the graph database.
- `most-used`: Display the most used objects of a specified type, based on relationships in the graph.
- `most-dependencies`: Display objects that are most often used as dependencies in on-demand feature view transformations.
- `served-by`: Display feature services that serve a particular feature from a specified data source.
- `upstream-impact`: Display objects that utilize a specified data source.
- `common-tags`: Group objects based on their tags and display related objects.
- `common-owner`: Group objects based on their owner and display related objects.

5.5.1 `feast graph execute-query`

This command provides the most flexibility for advanced users, allowing them to execute raw Cypher queries against the graph database. It grants users full control over querying the graph registry's content, thus enabling custom exploration of relationships, dependencies and metadata within the registry. This acts as an interface for executing complex and customized queries that might not be covered by standard commands.

Usage:

```
feast graph execute-query --query ''<cypher-query>''
```

Option:

- –query / -q: The Cypher query to be executed (required).

5.5.2 `feast graph most-used`

This command provides insights into the most used objects in the feature store, whether they are feature views, on-demand feature views, data sources, entities or fields. It reveals which objects play a critical role across the feature store by counting key relationships. A potential use for this command is for identifying and optimizing commonly used features or addressing bottlenecks in highly used entities or views. In order to extract the number of times each object is used, the relationships counted in each case are:

- Feature Views, On-Demand Feature Views: Counts the number of CONSUMES relationships from feature services.
- Data Sources: Counts both POPULATED_FROM and BASED_ON relationships from feature views and on-demand feature views respectively.
- Entities: Counts the number of USES relationships from feature views.
- Fields: Counts the number of HAS relationships from feature views.

Usage:

```
feast graph most-used --object <object-type> [--limit <number>]
```

Options:

- –object / -o: The type of object to query for (required). The only acceptable values are: feature-views, on-demand-feature-views, data-sources, entities and fields.
- –limit / -l: The maximum number of objects to display (optional).

5.5.3 `feast graph most-dependencies`

This command highlights objects that are most frequently used as dependencies in on-demand feature view transformations. It helps identify dependencies that are foundational for derived features or transformations. Additionally, users can get insight into which fields or feature views are most often reused in downstream computations, making it easier to understand potential impacts when these objects are updated. Depending on the specified object type, this command fulfills one of the following purposes:

- Feature Views: Counts the number of BASED_ON relationships from on-demand feature views to feature views.
- Fields: Counts the number of USED_FOR relationships from fields to other fields.

Usage:

```
feast graph most-dependencies --object <object-type> [--limit <number>]
```

Options:

- –object / -o: The type of object to query for (required). The only acceptable values are: feature-views and fields.
- –limit / -l: The maximum number of objects to display (optional).

5.5.4 `feast graph served-by`

This command allows you to query which feature services are serving a particular feature from a specified data source. It specifically identifies the feature services that serve a feature field from a given data source, either directly or through a chain of intermediate feature views or on-demand feature views. It is very easily implemented in Cypher since the SERVES relationship already exists in the registry, while if any other type of regitry were used, we would be forced to search through multiple layers of objects to extract the same information.

When planning changes to a data source or a specific field, it's crucial to understand which feature services would be affected. The `served-by` command enables you to trace the feature's dependencies and see where it is being used. By knowing which feature services are dependent on a particular field from a data source, you can better understand the downstream impacts and the scope of data usage in your feature store.

Usage:

```
feast graph served-by --data-source <data-source-name>
```

Options:

- –data-source / -d: The name of the data source (required).
- –field / -f: The name of the field (required).

5.5.5 `feast graph upstream-impact`

This command displays push data sources, feature views, on-demand feature views and feature services that depend on the specified data source. By showing all objects that are either directly or indirectly reliant on a given data source, users can easily understand the ripple effects of changes to the source, allowing for dependency management and safer modifications to critical data sources, without accidental disruptions.

Usage:

```
feast graph upstream-impact --data-source <data-source-name>
```

Option:

- –data-source / -d: The name of the data source (required).

It is particularly interesting to understand how this command is implemented in Cypher, since variable-length relationship patterns are used to traverse multiple layers of relationships efficiently and finally extract a deep dependency chain. By analyzing graph structures that may occur in the graph registry, we have determined the maximum possible path length for this scenario is 4, which limits the search in the query shown in Fig. 5.5 to paths of certain relationship types of length 1 to 4.

5.5.6 `feast graph common-tags`

This command organizes objects by their tags and displays which objects share the same metadata tags. Tagging is often used to group objects by certain characteristics or attributes, and this command helps users quickly identify how different entities, data sources and feature views are categorized. It is ideal for discovering common themes or patterns within the feature store.

Usage:

```
feast graph common-tags
```

```
MATCH (s:DataSource {{ data_source_name: <data-source-name> }})
<-[:POPULATED_FROM|RETRIEVES_FROM|BASED_ON|CONSUMES*1..4]-(d)
WITH DISTINCT d
RETURN
    COLLECT
     (DISTINCT CASE WHEN ''DataSource'' IN labels(d) THEN d.data_source_name END)
     AS data_sources,
    COLLECT
     (DISTINCT CASE WHEN ''FeatureView'' IN labels(d) THEN d.feature_view_name END)
     AS feature_views,
    COLLECT
     (DISTINCT CASE WHEN ''OnDemandFeatureView'' IN labels(d) THEN d.feature_view_name END)
     AS on_demand_feature_views,
    COLLECT
     (DISTINCT CASE WHEN ''FeatureService'' IN labels(d) THEN d.feature_service_name END)
     AS feature_services
```

Figure 5.5: Cypher query implementation of upstream-impact command.

5.5.7 `feast graph common-owner`

This command groups objects based on ownership, displaying all objects created or managed by a specific user. This can be especially useful for team management and governance, since it facilitates accountability by showing which user is responsible for different objects.

Usage:

```
feast graph common-owner
```

Chapter 6

Evaluation and Discussion

This section focuses on the results of experiments designed to evaluate and compare the performance of different Feast registries as they grow in size and complexity. These experiments are vital in showcasing the graph registry's capability to handle heavy workloads of interconnected data in production environments as well as identifying potential limitations as the feature store scales.

6.1 Performance Testing Framework

6.1.1 Environment

All experiments were conducted on a Virtual Machine with the following hardware and software specifications:

- Processor: 4 CPUs
- Memory: 16 GB RAM
- Storage: 100 GB disk capacity
- Operating System: Ubuntu 22.04.3 LTS

As far as Feast is concerned, the instance we executed the commands leverages a File offline store and a Redis online store. For the file registry, contents were stored on the disk, while local MySQL and Neo4j databases were utilized as SQL and graph registries respectively.

6.1.2 Data Generation

The goal of the data creation process is to generate a feature store with a scalable number of objects, such as data sources, feature views, on-demand feature views and feature services. This setup allows for performance testing across different registries while the feature store grows. The data generation process includes the following definitions:

1. Data Sources: Data sources consist of both file-based and push sources, with each scale unit containing two of each type. For every file data source, a corresponding Parquet file is dynamically generated, holding a fixed number of fields (20 by default) with integer values.
2. Feature Views: Feature views are created in groups of three for each data source, with each view referencing a pair of fields. A shared field is included in each feature view to simulate field reuse, and both batch and stream (fresh) versions are defined.
3. On-Demand Feature Views: On-demand feature views are derived from combinations of feature views and request sources. Each on-demand feature view is linked to a separate request source, with three on-demand feature views defined per scale unit. These

transformations involve arithmetic operations on selected features, producing derived fields while referencing multiple feature views to demonstrate complex dependencies.

4. Feature Services: Feature services aggregate features from multiple feature views and on-demand feature views. Each service includes ten features, selected from three feature views and two on-demand feature views, with a total of three feature services defined per scale unit.

The size of the feature store is controlled by a scale parameter, which determines the number of data sources, feature views and feature services, following the pattern described above.

The output of this process consists of:

- A Python file containing all generated feature definitions.
- Parquet files stored in a data directory, corresponding to each data source.

6.1.3 Testing Strategy

The objective of the testing strategy is to evaluate the performance of CLI commands across different registry configurations (Graph, SQL, File). These tests simulate real-world scenarios with complex feature dependencies and transformations.

1. Execute the data creation process for a given scale, generating feature definitions and Parquet files. The scale is gradually increased from 2 to 5000.
2. Run `feast apply` to register these definitions to the feature store utilizing the desired registry type.
3. Execute various commands and log the execution time of each one. The CLI commands that were tested are explained in brief below:
    - `served-by`: Displays feature services that serve a particular feature from a specified data source.
    - `upstream-impact`: Displays push data sources, feature views, on-demand feature views and feature services that depend on the same data source.
    - `common-owner`: Displays all objects managed by a specific user.
    - `most-dependencies feature-views`: Displays objects that are most frequently used as dependencies in on-demand feature view transformations
    - `most-used fields`: Displays the features that are most common in feature views.
4. After each test, the script cleans up the generated definitions, data files and registry contents.

Each experiment was repeated 3 times and average metrics were calculated.

This strategy allows for a comprehensive comparison of the performance and scalability of different registry configurations. The collected metrics provide insights into the ability of each registry to handle increasing numbers of objects and relationships, as well as the performance of graph-based dependency resolution compared to SQL/File-based implementations.

6.2 Query Execution Time and Efficiency Analysis

6.2.1 Feast Apply Performance

The `feast apply` command is responsible for creating and updating registry objects, making it a critical operation for scalability. Execution times were measured in seconds for the file-based, SQL and graph-based registries across varying numbers of objects -determined by the scale parameter- and the results are presented in Table 6.1.

Scale	File	SQL	Graph
2	8.7	12.3	38.3
20	10.3	44.3	209
100	20.3	177.3	993.3
1000	407.7	1497.3	22498.3
2000	984.3	3086.7	37706
5000	3707.3	9586.3	231351.3

Table 6.1: Execution Times for Feast Apply.

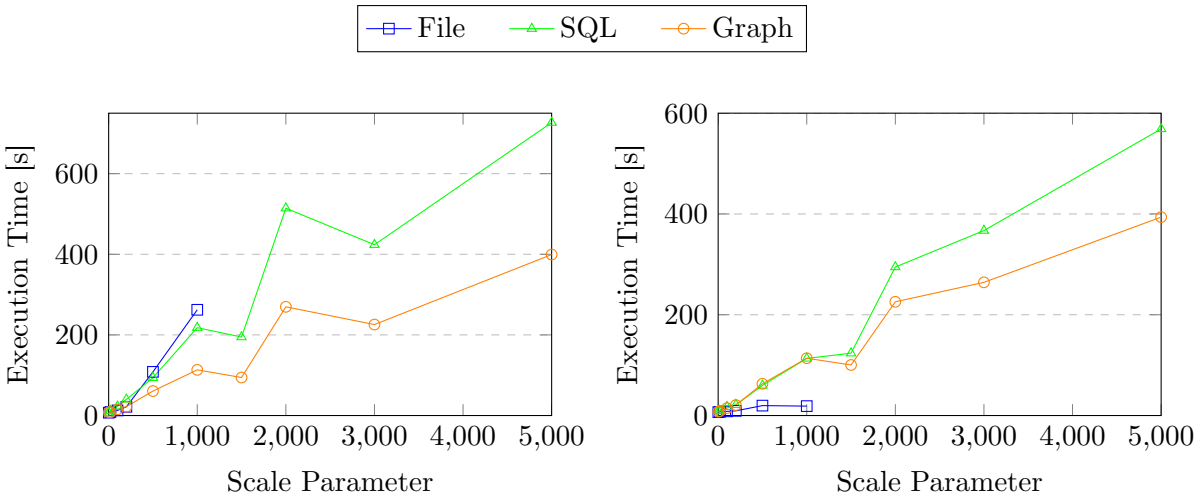
6.2.2 Command Execution Performance

After registering object definitions to the feature store, multiple CLI commands were executed to benchmark the performance of the graph registry against file and SQL versions. The specific execution times (in seconds) obtained for each experiment are given in the following tables and also plotted in figures. In general, when the feature store is small, all registry backends demonstrate comparable performance in executing CLI commands. Once the feature store increases in scale, however, clear differences emerge that reflect the underlying data access and query execution strategies of each implementation.

Scale	Served-by			Upstream-impact		
	File	SQL	Graph	File	SQL	Graph
2	7	7.7	8	6.7	8.7	7.7
20	8.7	12	8	7.3	9.7	9
100	12.7	22.7	15.3	8.3	17.3	14.7
200	21.7	40	22	9.3	21.3	20.7
500	108.3	93.7	60.7	19.7	59.7	63
1000	262.3	218	113.3	18.7	113.3	113.7
1500		194.7	94.3		123.7	100.3
2000		514.3	269.7		294.7	225.7
3000		423.7	225.7		367	264.3
5000		726.7	399.3		568.7	393.7

Table 6.2: Execution Times for Served-by and Upstream-impact Commands.

Because SERVES relationships are explicitly described inside the graph structure, the graph database registry simplifies the `served-by` command. In a graph database like Neo4j, relationships are first-class citizens, allowing for efficient traversal operations, since the query



(a) Served-by Performance

(b) Upstream-impact Performance

Figure 6.1: Performance of CLI Commands (1).

execution for this command directly leverages these predefined SERVES relationships. Performance is enhanced and query execution times are decreased as a result of this optimization, particularly as dataset sizes increase. The same connections must be found by searching through de-serialized objects in SQL and file-based registries, which need more computationally intensive operations to accomplish comparable outcomes because they do not naturally maintain relationships.

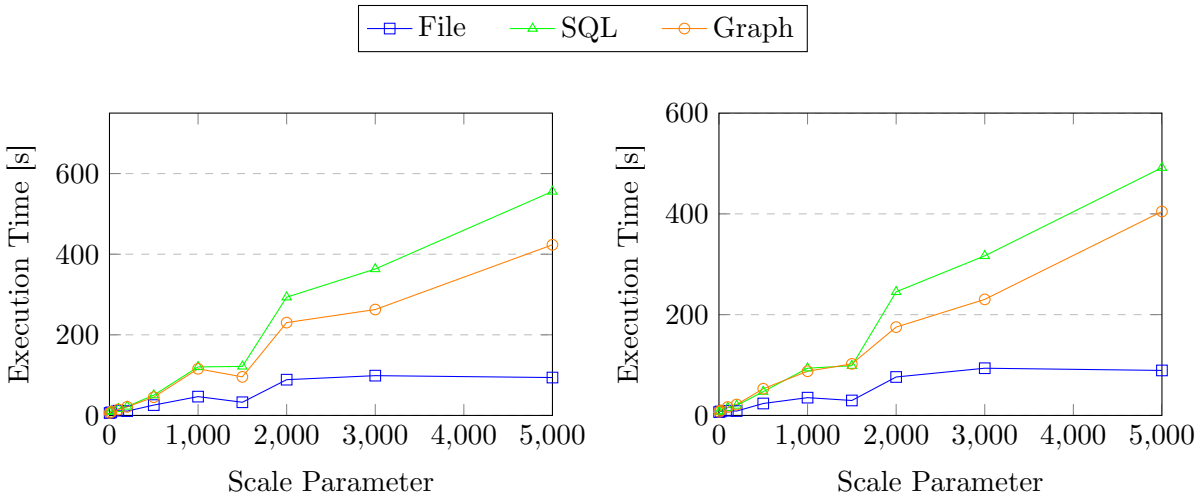
`upstream-impact` is a command that needs to traverse multiple relationships, offering an evident advantage for the graph-based registry. Neo4j natively supports variable-length and multi-hop traversals, which allows this query to scale efficiently as the complexity of the dependencies increases. In our tests, when the scale was increased, the graph registry consistently identified the impacted objects with only a moderate increase in query execution time.

However, when these same commands were executed against the file-based registry, they either returned empty results or failed entirely. This unexpected outcome may be attributed to memory constraints, inefficient searches or incomplete iteration over objects, since the file registry relies on in-memory de-serialization and search.

`common-owner` and `most-dependencies` commands both query properties that were added to the graph database as separate nodes and edges, while they were not stored separately in other types of registries. The file and SQL registries depend on fetching the feature store's protobuf representations and de-serializing them. Once in memory, filtering and aggregating objects (for example, grouping by owner or counting dependencies) can be performed extremely quickly using Python's native data structures. The file-based registry's performance advantage for these commands likely stems from its simplicity and in-memory processing. The SQL registry is slower in comparison due to the additional step of opening connections to the database to read objects. Between the two database options, Neo4j is queried directly to search for the desired relationships leading to faster results than the SQL registry.

Scale	Common-owner			Most-dependencies		
	File	SQL	Graph	File	SQL	Graph
2	6	9.3	6.3	6.7	7.3	7.3
20	7.7	11.3	8.7	8	8.7	10
100	11.7	16.3	14.7	9.3	15.3	16.7
200	10.7	22.3	21.3	8.7	19.7	21.7
500	25.7	50.3	45.3	23.7	47.3	53
1000	46.7	120.3	115.7	35.3	93.7	87.7
1500	32.7	121.7	95,7	29.7	98.7	102.3
2000	88.7	293.3	230,3	76.3	245.3	175.3
3000	98.7	363.3	262,7	93.7	316.7	230.3
5000	94	555.7	423,3	89.3	491.7	404.7

Table 6.3: Execution Times for Common-owner and Most-dependencies Commands.



(a) Common-owner Performance

(b) Most-dependencies Performance

Figure 6.2: Performance of CLI Commands (2).

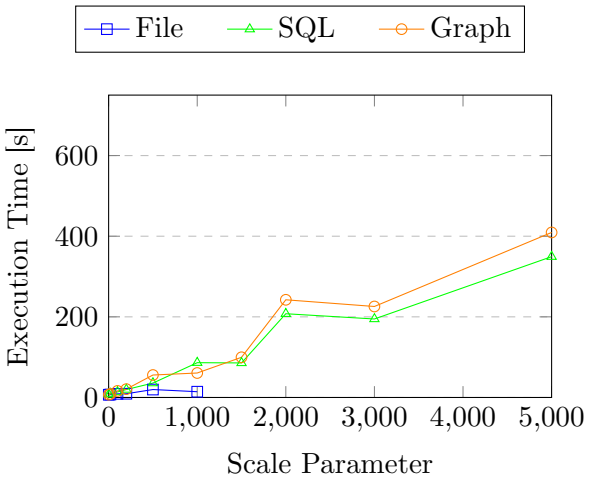


Figure 6.3: Performance of most-used fields CLI Command.

The most-used fields command is designed to aggregate relationship data and count the

Scale	File	SQL	Graph
2	6.7	7.7	7
20	7.3	9.7	8.7
100	9	13.7	16.3
200	9.3	19.7	20.7
500	19.7	35.7	55.7
1000	14.3	86.3	60.7
1500		85.7	99.7
2000		207.7	242.3
3000		194.7	225.7
5000		349.3	409.3

Table 6.4: Execution Times for Most-used fields Command.

dependencies across feature views. In scenarios with relatively shallow dependency chains, the cost of planning and executing a Cypher query in Neo4j sometimes leads to slower, rather than faster, response times than the file and SQL registries. The large number of Field nodes that need to be traversed should also be taken into consideration. As indicated by Table 6.4, the SQL registry outperforms the graph-based option for this simple query. Therefore, the graph registry's performance advantage was only noticed when the queries truly leveraged multi-hop relationships.

6.3 Strengths of Graph-Based Registries

The graph-based registry's intrinsic capacity to represent and navigate relationships is one of its main advantages. This capability is especially beneficial for operations that require complex dependency analysis, as the registry can efficiently navigate multi-hop connections across an intricate network of nodes and edges. In our experiments, queries that demanded detailed lineage and dependency tracking—such as those calculating upstream impact or identifying service dependencies—yielded precise and comprehensive results when executed against the graph-based registry.

Neo4j's use of the Cypher query language further augments these strengths. Cypher's expressive syntax enables the construction of sophisticated query constructs that can reveal deep interdependencies, a feat that is challenging to replicate with traditional file-based or SQL-based registries. Since nodes and relationships are stored directly in Neo4j, users can leverage these native structures to perform direct searches and optimizations that reduce the complexity of querying the registry. This flexibility offers immense potential for optimization and enables continuous performance improvements through query tuning and indexing.

Moreover, a more expressive and intuitive depiction of feature correlations is provided by the graph-based method. This qualitative benefit significantly enhances our comprehension of object dependencies, despite the fact that it might not be immediately measurable. Through Neo4j's graphical interfaces or dedicated CLI commands, data scientists and engineers can directly query the graph, enabling them to explore custom queries and extract insights that would be more difficult to obtain with other registry models. In intricate, large-scale machine learning settings, where debugging, auditing and system optimization all depend on a grasp of the intricate interactions between features, feature views and services, this degree of expressiveness and interactivity is critical.

6.4 Performance Trade-offs and Limitations

Although the graph-based registry offers significant benefits in handling complex, multi-hop queries and detailed dependency analysis, our experiments revealed several performance trade-offs and limitations that should be carefully considered.
The considerable overhead during initial setup and object creation is one of the main issues with the graph-based registry. In our experiments, the process of node creation and relationship verification in the Neo4j-backed registry was notably slower. This overhead arises from the need to commit multiple transactions, create a large number of nodes and validate numerous relationships to ensure data consistency within the graph structure. It is crucial to remember that the `feast apply` command's measured execution times reflect the first registration of every object in the registry. In a real-world scenario, however, subsequent updates typically involve only the incremental application of differences rather than a complete re-registration. Although the graph-based approach may exhibit slower initial apply times these overheads are expected to be insignificant in the context of the overall ML pipeline when only minor changes occur between `feast apply` executions.
Moreover, even though the graph-based registry is particularly favourable for executing complex queries that require deep traversals, this advantage is strongly contingent on query complexity. In cases where queries involve only shallow lookups or direct relationships, such as `common-owner` and `most-used fields`, the overhead associated with Cypher query planning and execution can diminish performance gains. In such scenarios, the relatively straightforward, in-memory processing of a file or SQL registry may yield faster response times.

6.5 Use Cases

According to our research, the registry backend selection should be based on the size of the deployment as well as the type of queries executed within the feature store.

6.5.1 Scale-Based Selection

- For small-scale deployments the file-based registry is optimal for development and experimental environments. Its simplicity and rapid, in-memory update mechanism deliver fast response times for basic lookups. In production scenarios at this scale, the SQL registry also offers a viable alternative by supporting atomic updates and efficient indexing without incurring the overhead associated with graph operations. However, because the benefits of multi-hop relationship traversal are not fully utilized, using a graph-based registry for such simple deployments may be needlessly complicated.

- At a larger scale and environments with highly connected objects, the performance characteristics begin to deviate more clearly. The file-based registry becomes less reliable, as its ability to process complex dependency queries diminishes. Although the SQL registry provides balanced performance, offering a middle ground, it is not as efficient in deep relationship traversal queries. In such scenarios, the graph-based registry is optimal, since its benefits start to emerge. Its native support for deep, multi-hop relationship traversal enables it to model and query complex feature dependencies effectively, making it indispensable for robust, production-grade ML systems despite the higher initial overhead.

6.5.2 Query Pattern Analysis

- Simple Lookups: For queries that involve direct, shallow data retrieval, the file-based and SQL registries excel, thanks to their straightforward access methods.

- Complex Relationship Traversal: The graph-based registry is unquestionably superior for procedures that call for navigating several relationships. More thorough and precise dependency analysis is made possible by its capacity to carry out effective multi-hop traversals utilizing the native graph query language Cypher.
- Mixed Workloads: A hybrid method might be used when both simple and complicated inquiries are common. With this method, deeper, relationship-intensive queries would be sent to the graph-based registry, while basic searches would benefit from the speed of the SQL or file-based registries.

These findings can be used to guide the design and optimization of feature store management systems by bringing the registry backend into compliance with the particular operational needs and anticipated query patterns. This approach ensures that the chosen solution maximizes performance while meeting the demands of modern, complex ML pipelines.

Chapter 7

Conclusions and Future Work

7.1 Summary of Findings

This thesis set out to investigate whether a graph database can be used as a feature store registry to better model relationships between objects and their lineage compared to traditional file-based and SQL-based registries. To this end, we implemented a Neo4j-backed registry for Feast, extending its core functionality by incorporating relationship-focused commands into the CLI. We also developed mechanisms for calculating feature dependencies in on-demand feature views.
Our experiments revealed that, as the feature store grows, the advantages of a graph-based registry become increasingly evident. In scenarios that involve complex, multi-hop dependency queries the Neo4j-backed registry excels, leveraging its native support for efficient relationship traversals. One drawback of using a graph database as the feature registry and introducing multiple relationships between objects is that the `feast apply` command becomes significantly slower. Conversely, for simpler, direct lookups, the file and SQL registries often outperform the graph registry due to their lower overhead and in-memory processing. These findings underscore that the choice of registry backend should be based on the nature of the queries and the size of the registry.
In conclusion, this study significantly advances and contributes to the field by demonstrating that a graph-based registry can provide an expressive and scalable solution for managing feature store objects and dependencies. Our graph-based registry provides a more natural modeling of relationships, with major benefits for dependency monitoring, impact analysis and data lineage. This is particularly valuable in modern, dynamic machine learning environments where features are regularly updated, and understanding their interrelationships is essential for debugging and auditing purposes.

7.2 Directions for Future Research

The findings of this research reveal several areas for future investigation and potential improvement:

- **Optimizing the graph registry**: The cost of multiple transactions and extensive query planning seem to hinder the graph registry's performance in certain operations, even though it exhibits great promise for intricate dependency queries. Future research could examine ways to optimize these processes, such as lowering the number of individual transactions through more effective query consolidation or introducing batch processing for node and relationship construction.

- **Applying the registry in production settings**: Additional research is required to assess whether the graph-based registry could be applied in the real-world. The system could be deployed and studied in a production environment in order to get insights into how the system performs under operational loads and how well it supports the evolving

needs of modern feature stores. Future work should also consider additional metrics, such as end-to-end latency in a production scenario, and analyze the impact of different workload patterns on resource utilization (memory, CPU and storage).

- **Combining registry backends**: It may also be worth investigating a hybrid approach, including multiple different registry implementations. Thus, the strengths of the SQL or file registry (for simple, direct queries) can be integrated with those of the graph registry (for deep, relationship-intensive queries). The best of both worlds would be possible with such a system, which could dynamically route queries to the most suitable backend depending on their complexity.

In summary, while this thesis illustrates the potential benefits of using a graph database as a feature store registry, particularly for complex dependency queries, it also identifies challenges that must be addressed to fully leverage these advantages at scale.

Bibliography

[1] J. Dowling, *Building Machine Learning Systems with a feature store.* O'Reilly Media, Inc, 2025.

[2] Feast, "Feast documentation," accessed: 2024-06-23. [Online]. Available: https://docs.feast.dev/

[3] Neo4j, "Neo4j documentation," accessed: 2024-07-06. [Online]. Available: https://neo4j.com/docs

[4] T. Verdonck, B. Baesens, M. Óskarsdóttir, and S. vanden Broucke, "Special issue on feature engineering editorial," *Machine Learning*, vol. 113, no. 7, p. 3917–3928, Aug 2021.

[5] J. Hermann, "Meet michelangelo: Uber's machine learning platform | uber blog," Sep 2017, accessed: 2024-08-26. [Online]. Available: https://www.uber.com/blog/michelangelo-machine-learning-platform/

[6] K. Sjöborg and R. Van Bruggen, accessed: 2024-08-25. [Online]. Available: https://www.hopsworks.ai/lp/neo4j-hopsworks-better-together

[7] F. Minutella and V. Piccioni, "Exploiting a feature store for graphs on neo4j," Sep 2022, accessed: 2024-07-17. [Online]. Available: https://www.youtube.com/watch?v=YSrkza8HY00

[8] B. Lackey and A. Prasad Thevaraj, "Graph feature engineering with neo4j and amazon sagemaker," Oct 2022, accessed: 2024-07-25. [Online]. Available: https://aws.amazon.com/blogs/apn/graph-feature-engineering-with-neo4j-and-amazon-sagemaker/

[9] W. Pienaar and M. Del Balso, "What is a feature store?" https://feastsite.wpenginepowered.com/blog/what-is-a-feature-store/, Jan 2021, accessed: 2024-06-21. [Online]. Available: https://feastsite.wpenginepowered.com/blog/what-is-a-feature-store/

[10] M. D. Balso, "What is a feature store?" May 2023, accessed: 2024-06-25. [Online]. Available: https://www.tecton.ai/blog/what-is-a-feature-store/

[11] Hopsworks, "Hopsworks platform," accessed: 2024-06-27. [Online]. Available: https://docs.hopsworks.ai/3.1/concepts/hopsworks/

[12] T. Kakantousis, A. Kouzoupis, F. Buso, G. Berthou, J. Dowling, and S. Haridi, "Horizontally scalable ml pipelines with a feature store," in *Proc. 2nd SysML Conf., Palo Alto, USA*, 2019.

[13] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, A. A. Ormenisan, S. Niazi, D. Bzhalava, K. Mak, V. Jouffrey, M. Ronström, R. Cunningham, and et al., "The hopsworks feature store for machine learning," *Companion of the 2024 International Conference on Management of Data*, p. 135–147, Jun 2024.

[14] Feathr, "Feathr documentation," accessed: 2024-06-27. [Online]. Available: https://feathr-ai.github.io/feathr/

[15] Featureform, "Featureform documentation," accessed: 2024-06-29. [Online]. Available: https://docs.featureform.com/concepts/immutability-lineage-and-dags

[16] Hopsworks, "The big dictionary of mlops llmops," accessed: 2024-06-29. [Online]. Available: https://www.hopsworks.ai/mlops-dictionary?utm_source=fs.org&utm_medium=web

[17] G. M. Arthur Olga, "Feature storing," accessed: 2024-06-29. [Online]. Available: https://mlops-guide.github.io/MLOps/FeatureStore/

[18] S. Hakro, "Graph databases (in-depth analysis)," Apr 2023, accessed: 2024-07-02. [Online]. Available: https://medium.com/@saifkhan666645/graph-databases-in-depth-analysis-4fbe280acbd2

[19] L. Stanescu, "A comparison between a relational and a graph database in the context of a recommendation system," in *Annals of Computer Science and Information Systems*, 09 2021, pp. 133–139.

[20] C. Rodrigues, M. R. Jain, and A. Khanchandani, "Performance comparison of graph database and relational database," 05 2023.

[21] G. Jaiswal, "Comparative analysis of relational and graph databases," *IOSR Journal of Engineering*, vol. 03, no. 08, p. 25–27, Aug 2013.

[22] E. F. Codd, "Data models in database management," *ACM SIGART Bulletin*, no. 74, p. 112–114, Jun 1980.

[23] R. Angles and C. Gutierrez, "Querying rdf data from a graph database perspective," in *The Semantic Web: Research and Applications*, A. Gómez-Pérez and J. Euzenat, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 346–360.

[24] R. Angles, "The property graph database model," in *Alberto Mendelzon Workshop on Foundations of Data Management*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:43977243

[25] R. Angles, H. Thakkar, and D. Tomaszuk, "Rdf and property graphs interoperability: Status and issues," in *Alberto Mendelzon Workshop on Foundations of Data Management*, 2019.

[26] R. Van Bruggen, *Learning Neo4j*. Packt Publishing Ltd, 2014.

[27] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys*, vol. 50, no. 5, p. 1–40, Sep 2017.

[28] J. Pokorný, "Graph databases: Their power and limitations," in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds. Cham: Springer International Publishing, 2015, pp. 58–69.

[29] P. Siva, D. Yamaganti, D. Rohita, and U. sikharam, "A review on python for data science, machine learning and iot," *International Journal of Scientific Engineering Research*, vol. 10, 11 2023.

[30] H. Qian, W. Liu, Z. Ding, W. Sun, and C. Fang, "Abstract syntax tree for method name prediction: How far are we?" *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, p. 464–475, Oct 2023.

[31] Click, "Welcome to click - click documentation (8.1.x)," accessed: 2024-09-02. [Online]. Available: https://click.palletsprojects.com/en/stable/