Bachelorarbeit

Eine deskriptive Anfrageschnittstelle für Hydra.PowerGraph

eingereicht von: Matthias Losch
Matrikelnummer: 215205549

eingereicht am: 08.11.2022

Gutachter/-innen: Dr.-Ing. Holger Meyer
Dipl.-Inf. Alf-Christian Schering

Betreuer: Dr.-Ing. Holger Meyer
Dipl.-Inf. Alf-Christian Schering

Zusammenfassung

Im Feld der Datenbankforschung stellen Graphdatenbanken eine große Innovation der letzten Jahre dar. Durch ihre Möglichkeit, hochgradig vernetzte Daten effizient darstellen und auswerten zu können, eignen sich solche Systeme für viele Anwendungsfälle, in denen das bisher vorherrschende Konzept der relationalen Datenbanken an seine praktischen Grenzen stößt. Allerdings hat dieses System den Graphdatenbanken eine wichtige Entwicklung voraus: eine einheitliche Anfragesprache fundiert auf einer soliden theoretischen Grundlage. Bei der Internationalen Organisation für Standardisierung (ISO) befindet sich ein solcher Sprachstandard, genannt GQL, gerade in der Entwicklung. Dieser ist allerdings nur für eine spezielle Art von Graphdatenbankmodell, dem Property-Graph-Modell, entworfen. Ein anderes Modell, das Hypergraphenmodell, kann unangepasst nicht mit GQL verwendet werden.

Diese Bachelorarbeit erarbeitet die Grundlagen, auf denen sich eine solche Anpassung an eine spezielle Art von Hypergraphdatenbankmodell, dem WossiDiA-PowerGraph-Modell, vornehmen lässt. Es wird eine theoretisch solide Grundlage, namentlich eine Graphenalgebra für Hypergraphen, erarbeitet, die sich mit der relationalen Algebra für relationale Datenbanken vergleichen lässt. Anschließend wird ein Vorschlag gegeben, wie diese Hypergraphenalgebra auf eine angepasste Version von GQL, HyperGQL, abgebildet werden kann. Dieser Vorschlag wird schließlich durch eine Implementation getestet und seine Tauglichkeit für Anfragen an die Datenbank untersucht. Dabei wird eine REST-Schnittstelle des WossiDiA-Systems verwendet, welche öffentlich bereitgestellt ist.

Abstract

Graph database systems have been a major innovation in database research over the past few years. Because of their ability to represent and evaluate highly interconnected data, these systems are well-suited for a wide range of applications where the currently dominating model of relational databases is reaching its practical limits. However, relational databases currently do have an important advantage over graph databases in that they have a standardized querying language grounded on a solid theoretical basis. The International Organization for Standardization (ISO) is developing a standard language called GQL for graph databases. This language however is only suited to one particular type of graph database model, the property-graph model. Other models such as the hypergraph model cannot be used with an unmodified version of GQL.

This Bachelor's thesis examines the foundations on which such a modification for hypergraph database models, namely the WossiDiA-PowerGraph model, can be made. A theoretical basis for querying languages, more precisely a graph algebra for hypergraphs, comparable to the relational algebra for relational database systems, is presented. The thesis explores how this algebra can be mapped onto a modified version of GQL, HyperGQL. These proposals are then implemented and tested to examine their suitability for querying the WossiDiA database. For this, the publicly available REST-API of the WossiDiA project is used.

Inhaltsverzeichnis

1. Einleitung

Der Kern jeder Software stellt die Verarbeitung und Auswertung von Informationen dar. Mit der Integration von elektronischen Geräten in immer mehr Situationen unseres Alltags werden immer mehr Informationen und Daten generiert. Dabei sind die Art und Herkunft dieser Daten sehr vielfältig: Smartphones speichern Bewegungsmuster und Anrufprotokolle, Suchmaschinen erfassen Suchverläufe und Gewohnheiten und soziale Netzwerke stellen die unterschiedlichsten Arten von Informationen über Menschen in Verbindung. Es erfordert effiziente und gut erforschte Techniken, um diese Informationen und deren Auswertungen auch langfristig speichern zu können.

Das gewählte System zur Festhaltung von Daten stellte dabei in der Vergangenheit meist das relationale Datenbanksystem dar, welches Informationen in Tabellen abspeichert und Verknüpfungen unter den einzelnen Einträgen herstellt. In den letzten Jahren wurden jedoch weitere Datenbankansätze entwickelt und vorgestellt, welche auf anderen Konzepten beruhen. Ein besonders vielversprechender Ansatz ist dabei das Graphdatenbankmodell, welches Daten nicht in tabellarischer Form erfasst, sondern in einer mathematischen Graphenstruktur speichert. Somit können auch stark verknüpfte und voneinander abhängige Daten effizient gespeichert und abgerufen werden, ohne dass ihre Verbindungen zueinander verloren gehen. Als Grundlage solcher Systeme dient meist das sogenannte Property-Graphen-Modell, erklärt in Abschnitt 2.2, welches eine besondere Art von Graphen darstellt.

Allerdings existiert für solche Systeme aktuell noch keine konsistente, allgemein anerkannte und theoretische Basis, auf welcher Anfragen an die Datenbank untersucht und optimiert werden können. Haben relationale Datenbanken etwa als Basis die relationale Algebra, welche jede Art von Anfrage abbilden kann, fehlt eine solche Algebra für Graphdatenbanken. Dies erschwert Optimierungen von Anfragen an die Datenbank und macht eine mathematische Untersuchung dieser über mehrere unterschiedliche Datenbanksysteme nahezu unmöglich. Es existieren Bestrebungen, diese Lücke zu füllen [1–4].

Ein weiteres Problem, auf welches Anwender von aktuellen Graphdatenbanken stoßen, ist das Fehlen eines einheitlichen Standards einer Anfragesprache. Während für relationale Datenbanken mit SQL [5] über alle Implementationen hinweg eine gemeinsame Schnittstelle existiert, muss bei jedem aktuellen Graphdatenbanksystem auf eine unterschiedliche Sprache zurückgegriffen werden, für Oracle-Systeme beispielsweise auf PGQL und Neo4j auf Cypher. Dies stellt einen erheblichen Mehraufwand bei der Entwicklung von Programmen dar, welche an solche Systeme angebunden werden sollen, da für jedes genutzte System eine weitere,

unterschiedliche Schnittstelle bedient werden muss. Um diesem Problem Einhalt zu gebieten, arbeitet die Internationale Standardorganisation ISO momentan an der Entwicklung eines einheitlichen Anfragestandards. Der sich in Entwicklung befindliche Sprachvorschlag GQL [6] soll diese Lücke schließen und die besten Eigenschaften der verschiedenen aktuellen Anfragesprachen in sich vereinen. Mit einer Veröffentlichung des Standards wird frühestens 2023 gerechnet.

1.1. Das Wossidlo-Archiv und WossiDiA

Abbildung 1.1.: Richard Wossidlo (rechts). Quelle: Archiv der Universitätsbibliothek Rostock

Richard Wossidlo war ein deutscher Volkskundler, welcher Ende des 19. und Anfang des 20. Jahrhunderts durch zahlreiche Feldstudien und Befragungen der Stadt- und Landbevölkerung alle Aspekte des Lebens der Menschen in Mecklenburg dokumentierte. Das Ergebnis seiner Arbeit stellen etwa 2 Millionen Notizen dar, auf welchen genaue Details über Ort, Zeitpunkt, Personen, Erzählern, Referenzen zu anderen Wissenschaftlern und Literatur und viele weitere Informationen dokumentiert sind. Wossidlo war es dabei wichtig, nicht nur die Information selbst sondern auch den Ort, Zeitpunkt und die Person zu dokumentieren, von welcher er lernte, festzuhalten, ebenso wie viele weitere Metadaten. Die dadurch entstandene Ansammlung an Karteikarten organisierte er in einem Boxenregal, in welchem Wossidlo Notizen mit ähnlichen Themen zusammenfasste.

Nach Wossidlos Tod lichteten Forscher seine Notizen für längerfristige Speicherung zunächst auf 35mm-Film ab. Zur besseren Erforschung und Aufarbeitung der Daten Wossidlos entstand schließlich ein digitales Archiv, dass Wossidlo Digital Archive (WossiDiA) [7]. Dabei gestaltete sich die korrekte Speicherung der stark untereinander verknüpften Daten als große Herausforderung. Um diese Verknüpfungen beizubehalten und untersuchen zu können,

entschied man sich für die Nutzung eines Graphdatenbankmodells. Am Geeignetsten erwies sich dabei die Speicherung in einer Hypergraphenstruktur, welche in Abschnitt 2.3 näher betrachtet wird.

Da ein solches Hypergraphendatenbankmodell noch nicht existierte, entstand im Rahmen des WossiDiA-Projektes die Sogenannte Hydra.PowerGraph-Datenbank. Diese besteht zum einen aus der Hypergraphendatenbank PowerGraph, welche eine Erweiterung der objekt-relationalen Datenbank PostgreSQL darstellt und deren Eigenschaften wie räumlich-zeitliche Erweiterungen und objekt-relationale Bestandteile wie Vererbung und nutzerdefinierte Typen verwendet. Aufsetzend auf PowerGraph ist das Hydra-Framework (HYpergraph of Documents in a Relational Archive), welches spezielle Funktionen für digitale Archive bereitstellt. Diese beinhalten unter anderem Standard-Objekttypen für Personen, Orte, Ereignisse und andere Objekte, welche häufig von Entwicklern von digitalen Archiven benötigt werden, sowie Funktionalitäten zur GIS-basierten Darstellung von Anfrageergebnissen und räumlich-zeitliche Analysewerkzeuge.

1.2. Ziel der Arbeit

Das Ziel dieser Bachelorarbeit ist die Erarbeitung einer theoretisch soliden Graphenalgebra für Hypergraphen als Grundlage für Hypergraphendatenbankensysteme. Dabei werden zunächst bestehende Ansätze für bereits existierende Graphdatenbanken evaluiert und deren Eignung und Erweiterung für Hypergraphen untersucht.

Auf Basis dieser neuen Hypergraphenalgebra wird darüber hinaus eine Anpassung des sich zum Zeitpunkt dieser Bachelorarbeit in Entwicklung befindenden Standards für Graphenanfragesprachen, GQL, auf Hypergraphendatenbanken untersucht. Hierbei betrachtet diese Arbeit ausschließlich lesende Anfragen an Datenbanken. Erweiternde oder ändernde Anfragen werden nicht in Betracht gezogen.

Schließlich soll eine prototypische Implementation der neu definierten Hypergraphenalgebra und der erweiterten GQL erstellt werden, welche Anfragen an das WossiDiA-System stellen und Antworten anzeigen kann.

Abbildung 1.2 veranschaulicht sowohl die Interaktion der einzelnen Bestandteile als auch den Platz, welche die Ergebnisse dieser Arbeit im WossiDiA-System einnehmen.

1.3. Struktur der Arbeit

Das folgende Kapitel 2 beschreibt die theoretischen Grundlagen, welche für das Verständnis dieser Arbeit benötigt werden. Dazu werden in Unterkapitel 2.1 zunächst eine grundlegende Erklärung des Konzepts mathematischer Graphen gegeben und formale Definitionen für ungerichtete und gerichtete Graphen aufgestellt. Unterkapitel 2.2 erweitert diese

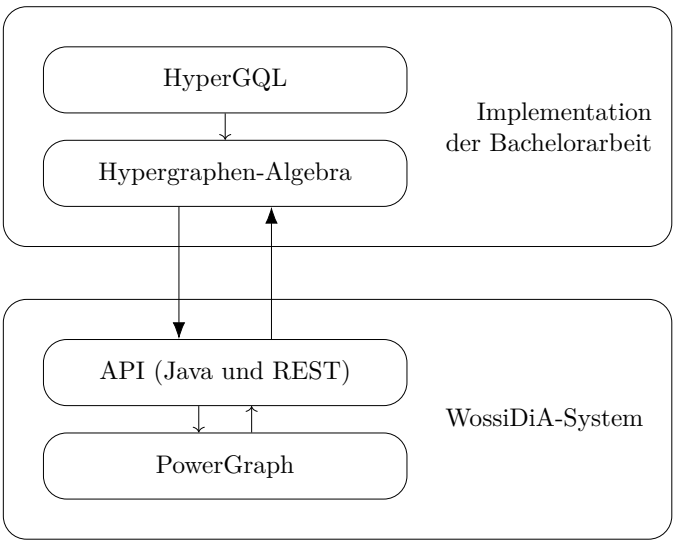


Abbildung 1.2.: Einordnung der Ergebnisse dieser Arbeit in das WossiDiA-System

Definition um das Konzept der Property-Graphen, auf welchen alle aktuellen Graphdatenbankmodelle beruhen und welches zum Verständnis der später aufgestellten Graphenalgebra notwendig ist. In Unterkapitel 2.3 werden schließlich Hypergraphen eingeführt, welche eine Erweiterung des Property-Graphen-Modells darstellen und auf welchen das WossiDiA-PowerGraph-Modell beruht, welches den zentralen Forschungsgegenstand dieser Arbeit darstellt.

Kapitel 3 liefert einen Überblick über den aktuellen Stand der Forschung und Technik im Bereich der Graphdatenbanken und Anfragesprachen. Dabei wird ein kurzer Überblick über die Herkunft und Verbreitung von Property-Graph-Datenbanken in Abschnitt 3.1 gegeben, bevor in Abschnitt 3.2 eine grundlegende Darstellung des PowerGraph-Modells folgt. Abschließend behandelt Unterkapitel 3.3 den sich abzeichnenden neuen Graphanfragestandard GQL, auf welchem die erweiterte Anfragesprache an PowerGraph, HyperGQL, basiert.

In Kapitel 4 wird die Aufstellung einer neuen Graphenalgebra für Hypergraphendatenbanken beschrieben, welche den Kern dieser Arbeit darstellt. Zunächst werden in Unterkapitel 4.1 logische Grundlagen von regulären Anfragen an Graphen erläutert und eine Hierarchie aufgebaut. Unterkapitel 4.2 bildet aus diesen Grundlagen eine logisch konsistente Graphenalgebra für Property-Graphen-Modelle, welche schließlich im folgenden Unterkapitel 4.3 erweitert und umgeformt wird, um auch für Hypergraphen-Datenbanken anwendbar zu sein. Abschließend wird eine Verbindung zwischen der neu aufgestellten Algebra und der GQL-Erweiterung „HyperGQL“ in Unterkapitel 4.4 gegeben. Es folgen einige Beispiele von Anfragen in HyperGQL und deren Ausdrücke in regulärer Hypergraphenalgebra.

Auf Basis dieser neuen Definitionen erläutert Kapitel 5 den Implementierungsanteil dieser Arbeit. Dabei wird zunächst die WossiDiA-REST-API erläutert, welche als Schnittstelle zwischen der Graphdatenbank und dieser Implementation liegt. Als nächstes folgt eine Erklärung des Parser-Generators ANTLR, welcher von der Implementation genutzt wird, um HyperGQL-Ausdrücke in Hypergraphenalgebra-Ausdrücke zu übersetzen. Anschließend folgt

eine Erklärung der Struktur der Implementation in Unterkapitel 5.3 und eine Reihe von Tests, welche die Funktion des Programms illustrieren. Schließlich wird in Unterkapitel 5.5 eine Reihe von Unzulänglichkeiten der REST-API aufgezeigt und mögliche Verbesserungsvorschläge gegeben.

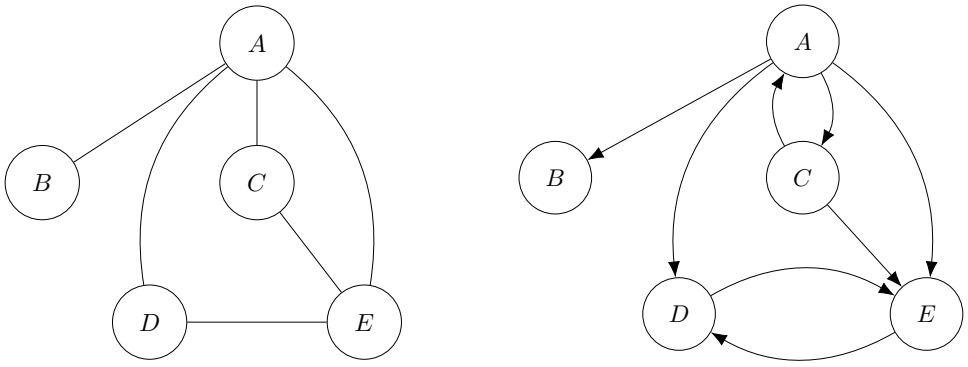
Den Abschluss dieser Arbeit stellt Kapitel 6 dar, welches in zwei Teile gegliedert ist. Der erste Teil ist dabei eine Zusammenfassung der Ergebnisse dieser Arbeit. Im zweiten Teil wird ein Ausblick über weitere Verbesserungen und mögliche Schritte gegeben, welche auf die Ergebnisse dieser Arbeit folgen können.

2. Grundlagen

In diesem Kapitel sollen grundlegenden Konzepte zusammengefasst und erklärt werden, auf welche in späteren Abschnitten dieser Bachelorarbeit Bezug genommen wird. Dabei handelt es sich hier nur um eine sehr kurze Besprechung der aufgezählten Themengebiete. Die Graphentheorie ist wie jedes Feld der Mathematik sehr groß und weitreichend und diese Arbeit behandelt nur einen sehr kleinen Teilaspekt.

Der erste Abschnitt dieses Kapitels behandelt zunächst grundlegende Definitionen, in welchem das Konzept von Graphen sowie gerichteten Graphen eingeführt wird. Darauf folgend werden sogenannte Property-Graphen definiert, welche in nahezu allen modernen Graphdatenbankmodellen Anwendung finden. Abschließend befasst sich der letzte Abschnitt mit Hypergraphen, welche eine Verallgemeinerung der Graphen des ersten Abschnitts darstellen und die Basis des PowerGraph-Systems bilden, welches den Hauptgegenstand der Untersuchungen dieser Arbeit sind.

2.1. Graphentheorie



(a) Ein ungerichteter Graph (b) Ein gerichteter Multigraph

Abbildung 2.1.: Beispiele für verschiedene Typen von Graphen

Ein Graph ist eine mathematische Struktur, welche aus Knoten und Kanten besteht. Dabei verbinden die Kanten die Knoten des Graphen. Knoten und Kanten sind dabei disjunkte Mengen, ein Knoten kann also nicht gleichzeitig eine Kante sein. Ein einfaches Beispiel solch eines Graphen ist in Abbildung 2.1a veranschaulicht. Dieses Konzept ergibt die folgende formale Definition aus [8]:

Definition 2.1 (Graph) *Ein Graph G ist ein paar (V, E) bestehend aus einer Menge von Knoten V und einer Menge von Kanten E, wobei $V \cap E = \emptyset$ und $E \subseteq [V]^2$.*

Diese sehr simple Definition stößt bei praktischen Anwendungsfällen schnell an ihre Grenzen. Eine naheliegende Erweiterung ist, den Kanten eines Graphen eine Richtung zuzuweisen. Dadurch werden beispielsweise Modellierungen der Fahrtrichtungen von Straßen auf Karten oder auch der Wirkungen von physikalischen Kräften möglich. Auf Abbildungen werden gerichtete Kanten in Graphen mit einem Pfeil gekennzeichnet, wie in Abbildung 2.1b zu sehen ist. Es ergibt sich die folgende Definition:

Definition 2.2 (Gerichteter Graph) *Ein Graph $G = (V, E)$ heißt gerichtet, wenn seine Kanten gerichtet sind. Eine Kante (v_1, v_2) heißt gerichtet, wenn $(v_1, v_2) \neq (v_2, v_1)$.*

Darüber hinaus sind weitere Eigenschaften für Graphen möglich. So werden Graphen, wessen Knoten nicht nur eine, sondern mehrere Kanten aufweisen, Multigraphen genannt. Graphdatenbanksysteme verwenden Property-Graphen in ihren Implementationen, auf welche im nächsten Abschnitt genauer eingegangen wird.

2.2. Property-Graphen

Property-Graphen erweitern das Konzept von gerichteten Multigraphen um Properties und Labels. Dabei stellen Labels zusätzliche Informationen dar, welche die Knoten und Kanten genauer beschreiben. Diese Informationen werden in Graphdatenbanken häufig für Typsysteme verwendet. Bei Properties handelt es sich hingegen um Schlüssel-Werte-Paare, welche die eigentlichen Daten der Knoten und Kanten darstellen.

Bonifati et al. geben in [1] die folgende formale Definition eines Property-Graphen an:

Definition 2.3 (Property-Graph) *Sei $\mathcal{O}$ eine Menge von Elementen, $\mathcal{L}$ eine endliche Menge von Labels, $\mathcal{K}$ eine Menge von Property-Schlüsseln und $\mathcal{N}$ eine Menge von Property-Werten. Es wird davon ausgegangen, dass diese Mengen paarweise disjunkt sind. Dann ist ein Property-Graph eine Struktur $G = (V, E, \eta, \lambda, v)$, wobei gilt:*

- *$V \subseteq \mathcal{O}$ ist eine endliche Menge Knoten;*
- *$E \subseteq \mathcal{O}$ ist eine endliche Menge Kanten;*
- *$\eta : E \to V \times V$ ist eine Funktion, welche jeder Kante ein geordnetes Knotenpaar aus V zuweist;*
- *$\lambda : V \cup E \to \mathcal{P}(\mathcal{L})$ ist eine Funktion, welche jedem Element eine endliche Menge an Labels aus $\mathcal{L}$ zuweist ($\mathcal{P}(\mathcal{L})$ stellt die Potenzmenge von $\mathcal{L}$ dar);*

- $v : (V \cup E) \times \mathcal{K} \to \mathcal{N}$ *ist eine partielle Funktion, welche Knoten und Kanten eine Menge von Property-Schlüsseln aus* $\mathcal{K}$ *und jedem Property-Schlüssel eine Menge von Property-Werten aus* $\mathcal{N}$ *zuordnet.*

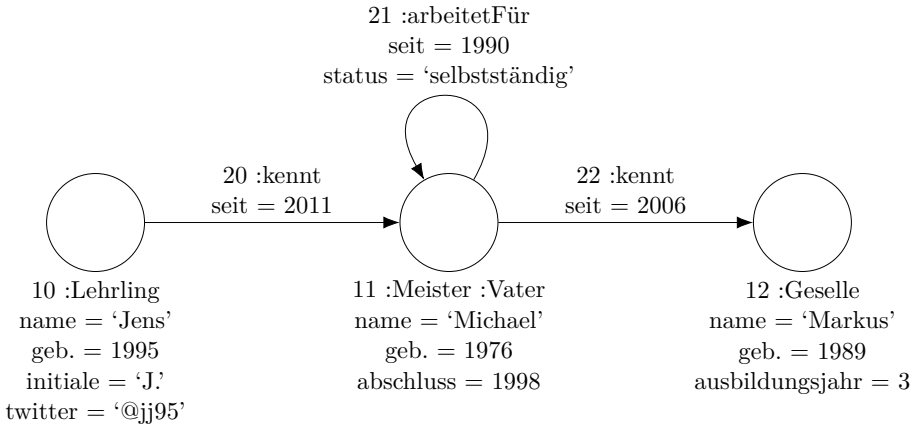


Abbildung 2.2.: Beispiel eines einfachen Property-Graphen (übernommen aus [1])

Abbildung 2.2 veranschaulicht diese Definition. Die Kreise stellen hier Knoten, die Pfeile Kanten dar. Der Graph besitzt drei Knoten (10, 11 und 12) und drei gerichtete Kanten (20, 21 und 22). Jeder Knoten und jede Kante besitzen sowohl Labels, hier gekennzeichnet mit einem vorangestellten Doppelpunkt, als auch Properties. Diese sind dadurch zu erkennen, dass sie sowohl einen Schlüssel als auch einen Wert besitzen, getrennt durch ein Gleichheitszeichen. Jedem Knoten und jeder Kante kann eine beliebige Menge an Properties zugewiesen werden, welche von den Labels unabhängig ist.

2.3. Hypergraphen

Hypergraphen stellen eine Verallgemeinerung der in den Definitionen 2.1 und 2.2 formalisierten Graphen dar. Dabei wird unter anderem die Einschränkung aufgehoben, dass Kanten maximal zwei Knoten miteinander verbinden können. Kanten mit mehr als zwei Knoten werden dementsprechend Hyperkanten genannt. Eine gängige Definition, welche auch vom PowerGraph-System verwendet wird, wird von Gallo et al. [9] angegeben und lautet wie folgt:

Definition 2.4 (Hypergraph) *Ein Hypergraph ist ein Tupel* $H = (V, E)$*, bestehend aus einer Menge von Knoten* V *und einer Menge von Hyperkanten* $E = \{E_1, E_2, \ldots, E_m\}$ *mit* $E_i \subseteq V$ *für* $i - 1, \ldots, m$*.* H *ist ein Standard-Graph, falls* $|E_i| = 2, i = 1, \ldots, m$*.*

Abbildung 2.3 macht den Unterschied zwischen „normalen“ Graphen und Hypergraphen deutlich. So ist es möglich, viele, thematisch verwandte Knoten über eine einzelne Hyperkante miteinander zu verbinden. Dies macht sich das PowerGraph-System bei der Vernetzung der Daten Wossidlos zunutze, welche viele einzelne Datenpunkten darstellen, die jeweils thematisch eng miteinander in Verbindung stehen.

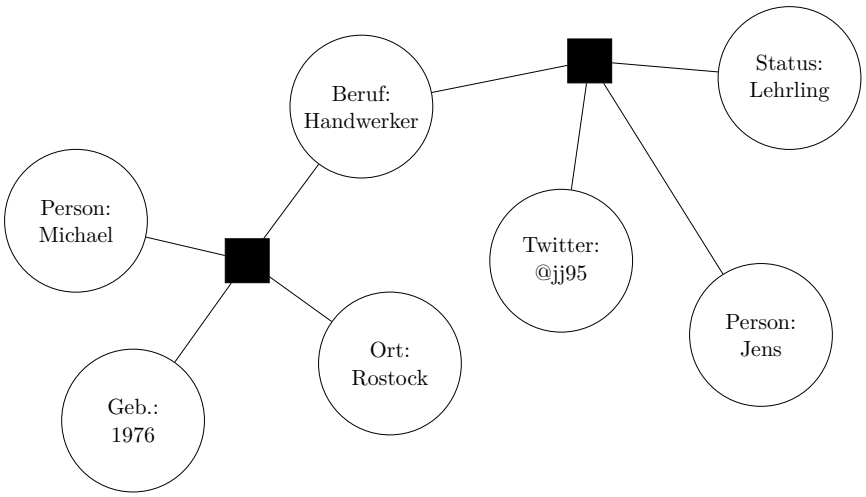


Abbildung 2.3.: Beispiel eines ungerichteten Hypergraphen

Wie auch bei den in Definition 2.2 definierten gerichteten Graphen ist es bei Hypergraphen möglich, Hyperkanten eine Richtung zuzuweisen. Diese Erweiterung wird von Gallo in [9] wie folgt angegeben:

Definition 2.5 (Gerichteter Hypergraph) *Ein gerichteter Hypergraph H ist ein Hypergraph mit gerichteten Hyperkanten. Eine gerichtete Hyperkante ist ein Tupel $e = (S, T)$, wobei $S, T \subseteq V \cup \emptyset$ und $S \cap T = \emptyset$.*

Hierbei handelt es sich bei den Mengen S und T um Mengen von Knoten aus der Gesamtmenge V. S stellt dabei die Menge der Quellknoten, T die Menge der Zielknoten der Hyperkante dar. Diese müssen nach dieser Definition disjunkt sein, dürfen aber auch der leeren Menge entsprechen.

Neben dieser Definition von Hypergraphen sind auch noch andere Definitionen geläufig. So definieren unter anderem Feng et al. in [10] nicht nur Hyperkanten sondern auch Hyperknoten. Diese finden in Hypergraphendatenbanken allerdings zumindest zum jetzigen Zeitpunkt noch keine Anwendung und werden von dieser Arbeit daher außer Acht gelassen. Des Weiteren definiert Ausiello in [11–13] Hyperkanten als Tupel (T, h) mit $h \in V$ und $T \neq \emptyset$. Diese Definition schränkt die Menge der Zielknoten auf ein Element ein und erlaubt deshalb nur m:1-Beziehungen. Im Folgenden wird daher die Definition von Gallo verwendet, welche auch m:n-Beziehungen zwischen Knoten erlaubt.

3. Stand der Technik

Das folgende Kapitel gibt einen Überblick über den aktuellen Stand der Forschung und Technik zu Graphdatenbanken und Anfragesprachen, genauer Property-Graph-Datenbanken und PowerGraph sowie den sich abzeichnenden Standard für Graphdatenbankanfragesprachen, GQL.

3.1. Property-Graph-Modelle

Zum Zeitpunkt dieser Arbeit wird von allen industriell genutzten Graph-Datenbanken das Property-Graph-Modell aus Abschnitt 2.2 verwendet. Zu den großen Vertretern dieser Modelle gehören Neo4j, IBM Graph, Oracle Graph Database und SAP HANA. Dabei existieren unterschiedliche Definitionen dieses Datenbankmodells [14–17], da jeder Hersteller Property-Graphen auf unterschiedliche Weise in seinem System verwendet und erweitert.

Dieser Typ von Datenbanken entstand Mitte der 2000er und nutzt anstelle von stark strukturierten Datenstrukturen wie Relationen in relationalen Datenbanken mathematische Graphen, um Daten zu speichern und auszuwerten. Damit reihen sich Property-Graph-Datenbanken in eine Entwicklung von semi- und schwach strukturierten Datenbankmodellen ein, welche Mitte der 1990er Jahre mit XML-Datenbanken ihren Anfang nahm und mit NoSQL-Systemen bis heute fortgeführt wird.

Als Grund für die Popularität solcher Systeme ist zum einen die schlechte Verwendbarkeit von relationalen Datenbanksystemen bei stark vernetzten und nur schwach strukturierten Daten. Beispiele hierfür sind Datenbestände von sozialen Netzwerken oder die Analyse von chemischen und biologischen Prozessen, welche deutlich besser und natürlicher durch Graphen beschrieben werden können. Ein ebenfalls wichtiger Faktor ist das Aufkommen und Wachstum des Internets, speziell des semantischen Webs, in welchem viele heterogene Daten stark verteilt vorliegen. Als letzter Grund kann die Anhäufung von großen Datenmengen zu Analysezwecken genannt werden. Diese sogenannte Big Data hat in den letzten Jahren stark an Wichtigkeit und Aufmerksamkeit gewonnen. Durch Graphalgorithmen kann Big Data teilweise deutlich effizienter und kosteneffektiver analysiert werden, was die Entwicklung von Graph-Datenbanken begünstigt.

3.2. PowerGraph-Modell

Das PowerGraph-Datenbank-Modell, beschrieben von Meyer et al. in [18], stellt ein Graphdatenbankmodell dar, welches im Gegensatz zu den meisten anderen modernen kommerziellen Graphendatenbanken anstelle von Property-Graphen auf Hypergraphen fußt. Hierbei wird die Hypergraphen-Definition von Gallo et al. in 2.5 genutzt und um verschiedene Eigenschaften erweitert.

Eine erste Modifizierung von Definition 2.5 ist, dass die Menge der Quell- und Zielknoten des Hypergraphen nicht mehr disjunkt sein muss, d.h. $(v, r) \in S \cap T$. Dies ermöglicht, Knoten sowohl Quell- und Zielknoten derselben Kante sein können und ermöglicht in dem ansonsten gerichteten Hypergraphen auch bidirektionale Kanten.

Eine andere Erweiterung stellt die Einführung von Rollen dar. Diese stellen ein Äquivalent zu den Labels von Property-Graphen dar und können sowohl Knoten als auch Kanten zugeordnet werden. Wenn einem Knoten mehrere Rollen zugeordnet werden, kann dieser, identifiziert durch die jeweilige Rolle, auch mehrmals in einer Hyperkante vertreten sein. Eine Konsequenz aus dieser Erweiterung ist, dass die Mengen der Quell- und Zielknoten S und T keine reinen Teilmengen mehr von V sind, sondern um die Menge alle Rollen R erweitert werden: $S \subseteq V \times R \wedge T \subseteq V \times R$.

Die dritte Änderung des PowerGraph-Systems an der Hypergraphen-Definition von Gallo ist die Einführung von Typen für Knoten und Kanten. Während in Property-Graphen Properties zur Beschreibung von Daten für Elemente des Graphen verwendet werden, lassen sich in PowerGraph Objekte mit ähnlichen Eigenschaften durch ein Typsystem in Gruppen einteilen. Meyer et al. geben als Beispiel den Datenbestand des Wossidlo-Archivs an, welcher einzelne Datenobjekte anhand von Eigenschaften wie Personen, Orte, Daten, etc. typisiert. Objekte mit gemeinsamen Typen werden in spezialisieren Knoten-Containern gespeichert, was Anfrageoptimierungen wie beispielsweise Top-k Queries zulässt [18].

Im Folgenden Bezeichnet A die Menge aller Hyperkanten. Das Typsystem von PowerGraph lässt sich durch die oben genannten Erweiterungen in folgende Mengen aufteilen:

- Γ^V als Menge von Knotentypen
- Γ^A als Menge von Kantentypen
- R als Menge aller Rollen

Darüber hinaus existiert eine Menge von Abbildungen $\{\alpha^V, \alpha^A, \rho^V, \rho^A, \delta\}$, welche die Zuordnung von Typen für Knoten, Kanten und Rollen festlegt:

$$\alpha^V : V \mapsto \Gamma^V$$
$$\alpha^A : A \mapsto \Gamma^A$$
$$\rho^V : R \mapsto \Gamma^V$$
$$\rho^A : R \mapsto \Gamma^A$$

Als α werden hier Abbildungen bezeichnet, welche einem Knoten beziehungsweise einer Kante einen jeweiligen Typ zuordnen. So weist α^V einem Element aus der Menge aller Knoten V einen Knotentyp aus der Menge Γ^V zu. Analog verhält es sich mit der Abbildung α^A, welche eine Kante aus A auf einen Kantentyp aus Γ^A abbildet. Die Abbildungen ρ verfahren gleich. Anstelle von Knoten und Kanten wird hier nun einer Rolle ein Knotentyp (ρ^V) beziehungsweise ein Kantentyp (ρ^A) zugewiesen.

Weiterhin wird eine weitere Abbildung δ definiert, welche die Richtung einer Hyperkante angibt. Da das Graphenmodell in PowerGraph durch die Erweiterung von Rollen sowohl gerichtete Kanten (als f-arc für vorwärtsgerichtet und b-arc für rückwärtsgerichtet bezeichnet) als auch bidirektionale Kanten zulässt, ergeben sich die folgenden drei Fälle:

$$\delta \mapsto \begin{cases} 1, & \text{falls f-arc} \\ -1, & \text{falls b-arc} \\ 0, & \text{falls bidirektional} \end{cases}$$

Meyer et al. definieren in [18] für dieses Typsystem schließlich die folgenden Integritätsbedingungen, welche für jeden Typen einer Hyperkante erfüllt sein müssen:

$$\forall(v,r) \in S \cup T : \rho^V(r) = \alpha^V(v)$$
$$\forall(v,r) \in S \cup T : \rho^A(r) = \alpha^A(A)$$
$$\forall(v,r) \in S : \delta(r) \in \{-1, 0\}$$
$$\forall(v,r) \in T : \delta(r) \in \{0, 1\}$$

3.3. GQL

Die „Graph Query Language“, kurz GQL, ist ein neuer Standard für Anfragesprachen an Property-Graph-Datenbanken. 2017 angekündigt stellt dieser Standard eine Vereinigung und Zusammenfassung bestehender populärer Graphanfragesprachen wie Cypher [19, 20], PGQL [21] und G-Core [22] dar. Ein Überblick über diese einzelnen Anfragesprachen wird in [23] gegeben. Ziel ist die Schaffung eines einheitlichen Sprachstandards für alle Property-Graph-Datenbanken äquivalent zum Sprachstandard SQL für relationale Datenbanksysteme. Aktuell befindet sich GQL in der Entwicklung der ISO. Mit einer Veröffentlichung des Standards wird frühestens 2023 gerechnet.

Im folgenden wird ein grober Überblick über die Grammatik von GQL gegeben, wie sie zum Zeitpunkt dieser Arbeit bekannt ist. Die Basis dieser Informationen bildet dabei der technische Bericht „GQL Scope and Features“ [24], welcher von einer Forschungsgruppe verschiedener Forscher der Hersteller populärer Graphdatenbankhersteller unter Führung Neo4js verfasst wurde. Er kann als Ausgangspunkt und Rahmen der Eigenschaften und Funktionen von GQL angesehen werden.

GQL-Anfragen bestehen aus genau einer `procedure`. Diese beinhaltet Deklarationen lokaler Variablen sowie den `body`, welcher den Hauptteil der Anfrage darstellt. Ein `body` besteht aus mindestens einem `composite statement`, welches wiederum aus einer Menge einzelner `statements` zusammengestellt ist. Ein Statement stellt eine einzelne Anfrage nach dem Prinzip des Pattern Matchings dar. Sie besteht aus einem Patten, welches die Art der Anfrage beschreibt und einem Rückgabewert, definiert durch **`RETURN`**. In diesem können einzelne Variablen und deren Attribute zur Rückgabe ausgewählt werden. Außerdem ist es möglich, mit Hilfe des **`WHERE`**-Schlüsselwortes Filter-Prädikate zum Filtern einzelner Knoten und Kanten zu definieren.

Die folgende Abbildung 3.1 veranschaulicht eine beispielhafte Anfrage in GQL auf dem in Abbildung 2.2 dargestellten Graphen sowie ihre einzelnen Bestandteile.

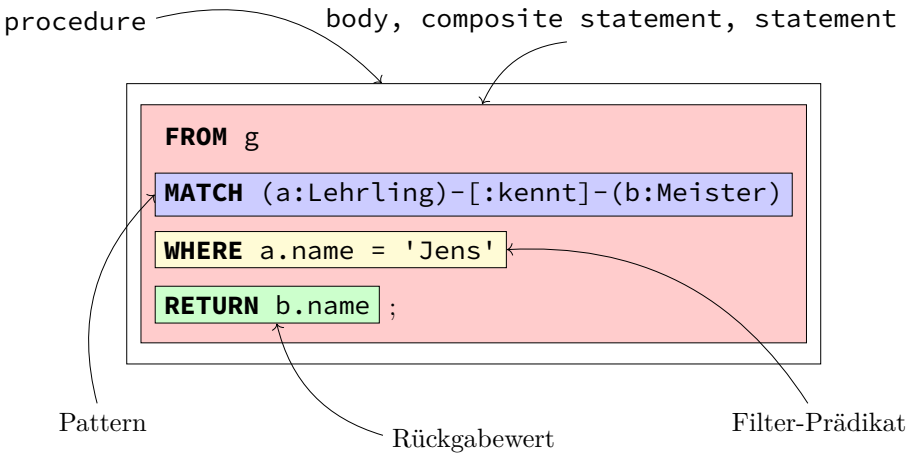


Abbildung 3.1.: Aufbau einer GQL-Anfrage anhand eines Beispiels

4. Graphenalgebra

Zur Untersuchung von Programmiersprachen ist es sinnvoll, eine logische Grundlage für solche zu schaffen. Daher beschäftigt sich die theoretische Informatik mit der Untersuchung von formalen Sprachen und teilt diese in Klassen und Hierarchien ein, um sie miteinander vergleichen zu können und ihre Mächtigkeit sowie ihre möglichen Ausdrücke zu untersuchen.

Anfragesprachen stellen eine besondere Form von formalen Sprachen dar. Durch ihren speziellen und sehr eng definierten Einsatzbereich sind sie besonders zur Untersuchung und Definition durch Logik geeignet. Die Grundlage solcher Untersuchungen stellt eine sogenannte Algebra dar, welche ein logisches System aus Operatoren und Semantiken darstellt. Dabei wird darauf geachtet, dass Operatoren definiert werden, welche gleichmächtig mit den zu untersuchenden Sprachen sind und gleiche Definitions- und Wertebereiche aufweisen. Somit lässt sich die zu untersuchende Sprache auf die definierte Algebra abbilden und dadurch formal besser auf ihre Eigenschaften untersuchen. Für das relationale Datenbankmodell existiert eine solche Algebra mit der relationalen Algebra bereits [25].

Die Hauptmotivation hinter dieser Arbeit ist die Definition einer solchen Algebra für Hypergraphendatenbankmodelle. Zum aktuellen Zeitpunkt dieser Arbeit ist solch eine Definition für diese spezielle Art von Graphdatenbankmodell in der Literatur nicht bekannt. Des Weiteren stellt das PowerGraph-Modell das einzige bekannte Hypergraphendatenbankmodell dar, sodass auch in praktischen Implementation eine solche logische Grundlage für Anfragesprachen nicht existiert. Dies macht die Ergebnisse dieser Arbeit zum ersten Versuch einer Aufstellung einer solchen Algebra für Hypergraphen. Das folgende Kapitel beschreibt die Aufstellung dieser Algebra.

Die folgenden Abschnitte nehmen starken Bezug auf [1]. Zunächst werden simple Bausteine von Graphenalgebren eingeführt, welche nach und nach generalisiert werden um immer mächtigere Anfragen an das Datenbanksystem stellen zu können. Anschließend wird eine Erweiterung dieser Graphenalgebra auf Property-Graphen vorgenommen, welche als Basis für moderne Graphdatenbanksysteme verwendet werden kann. Schließlich wird untersucht, inwieweit diese Algebra ergänzt und abgeändert werden muss, um auch für Hypergraphendatenbankmodelle, speziell das PowerGraph-System, nutzbar zu sein.

4.1. Logische Grundlagen

Eine Graphenalgebra für Property-Graphen und Hypergraphen lässt sich in unterschiedliche Teile gliedern, welche jeweils immer umfangreichere Anfragen an das Datenbanksystem zulassen. Mit diesen Bausteinen wird im Folgenden eine Hierarchie aufgestellt und die einzelnen Ebenen erklärt. Die Definitionen der einzelnen Abschnitte sind dabei aus [1] übernommen.

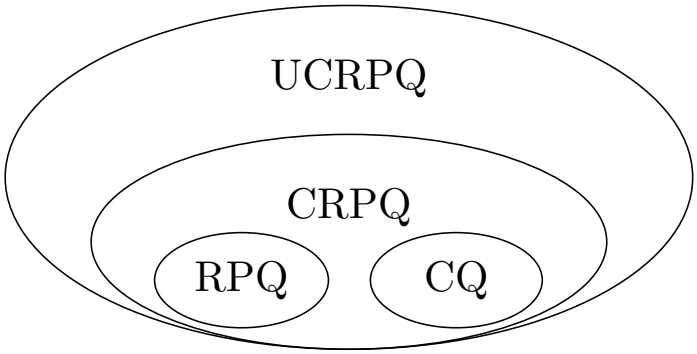


Abbildung 4.1.: Hierarchie von Algebra-Bausteinen für Graphenalgebras

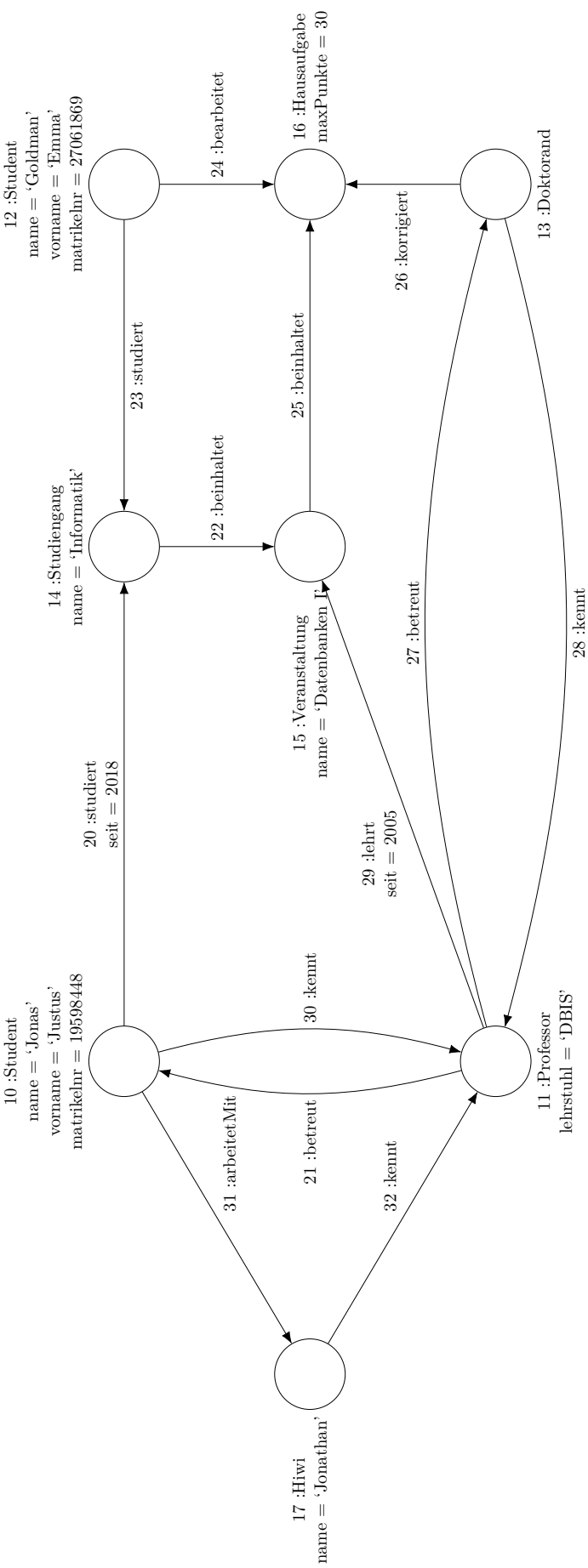


Abbildung 4.2.: Beispiel für einen Property-Graphen im Universitätsumfeld

4.1.1. Regular Path Queries

Eine sehr simple Anfrage an eine Graphdatenbank ist das Finden von direkten und indirekten Verbindungen in Graphen. Dies wird zum Beispiel in sozialen Netzwerken genutzt, wo durch solch eine Anfrage einer Person ein Wohnort zugewiesen werden kann. Solche Anfragen werden durch Regular Path Queries (RPQ) realisiert [26], welche reguläre Ausdrücke darstellen. Diese Ausdrücke beschreiben Pfade innerhalb der Graphen, welche aus einer beliebigen Anzahl von Hintereinanderreihungen von Kanten bestehen. Diese Kanten werden durch Labels aus der Label-Menge $\mathcal{L}$ identifiziert. Daher müssen auf dem Graphen, an welchen diese Anfrage gestellt wird, ebenfalls Labels definiert sein, wie es bei Property-Graphen in Definition 2.3 der Fall ist.

Definition 4.1 (Regular Path Queries) *Sei $\mathcal{L}$ die Menge aller Labels eines Graphen. Dann gilt:*

- *Wenn $a \in \mathcal{L}$, dann $a \in RPQ$*
  (Ein Label a aus der Menge aller Labels $\mathcal{L}$ ist eine RPQ),

- *Wenn $e \in RPQ$, dann $e^- \in RPQ$*
  (Wenn e eine RPQ ist, dann ist auch die Inverse $(e)^-$ eine RPQ),

- *Wenn $e, f \in RPQ$, dann $e/f \in RPQ$*
  (Eine Hintereinanderreihung (Konjunktion) zweier RPQs e, f bildet wieder eine RPQ),

- *Wenn $e, f \in RPQ$, dann $e + f \in RPQ$*
  (Eine Disjunktion zweier RPQs e, f bildet wieder eine RPQ),

- *Wenn $e \in RPQ$, dann $e^+ \in RPQ$*
  (Die transitive Hülle einer RPQ ist wieder eine RPQ).

Evaluation:

Sei $G = (V, E, \eta, \lambda, v)$ ein Property-Graph. Die Semantik zur Evaluation des Ausdrucks $g \in RPQ$ auf G ist die Menge an Knotenpaaren $[\![g]\!]_G \subseteq V \times V$, welche wie folgt definiert werden:

- *Wenn $g = a \in \mathcal{L}$, dann $[\![g]\!]_G = \{(s,t) \mid \exists\ Kante \in E : \eta(Kante) = (s,t) \wedge a \in \lambda(Kante)\}$.*
- *Wenn $g = e^- \in RPQ$, dann $[\![g]\!]_G = \{(t,s) \mid (s,t) \in [\![e]\!]_G\}$.*
- *Wenn $g = e/f \in RPQ$, dann $[\![g]\!]_G = \{(s,t) \mid \exists u \in V : (s,u) \in [\![e]\!]_G \wedge (u,t) \in [\![f]\!]_G$.*
- *Wenn $g = e + f \in RPQ$, dann $[\![g]\!]_G = [\![e]\!]_G \cup [\![f]\!]_G$.*
- *Wenn $g = e^+ \in RPQ$, dann $[\![g]\!]_G = \{(s,t) \mid (s,t) \in TC([\![e]\!]_G)\}$, wobei $TC(R)$ die transitive Hülle der binären Relation R darstellt.*

Beispiel Die Anfrage

$$q = \text{:studiert/:beinhaltet}^+$$

auf Graph G aus Abbildung 4.2 ergibt

$$[\![q]\!]_G = \{(10,15), (10,16), (12,15), (12,16)\}$$

Dabei werden zuerst alle Knotenpaare gesucht, welche eine Kante mit dem Label :studiert verbunden sind. Anschließend werden ausgehend von den gefundenen Zielknoten alle Knoten gesucht, welche über eine beliebige Anzahl an Kanten mit dem Label :beinhaltet verbunden sind. Diese gefundenen Knoten stellen anschließend mit den ersten Knoten die Ergebnisknotenpaare dar.

4.1.2. Conjunctive Graph Queries

Neben dem Finden von Verbindungen zwischen zwei Knoten ist es häufig auch von Interesse, Subgraphen innerhalb eines größeren Graphennetzwerkes zu finden. Ein Beispiel ist hierfür das finden zweier Personen in einem sozialen Netzwerk, welche mehrere gemeinsame Eigenschaften wie beispielsweise einen gemeinsamen Vornamen, Wohnort und Musikgeschmack aufweisen. Solche Subgraphen können mittels Conjunctive Graph Queries (CQ) bestimmt werden.

Definition 4.2 (Conjunctive Graph Queries) *Sei $\mathcal{V}$ eine Menge von Knotenvariablen. Dann ist eine CQ ein Ausdruck der Form*

$$(z_1, \ldots, z_m) \leftarrow a_1(x_1, y_1), \ldots, a_n(x_n, y_n)$$

wobei

- $m \geq 0, n > 0$,
  (Die Anzahl aller Elemente in der Menge aller im Subgraphen gefundenen Knoten ist ≥ 0 und es werden mindestens zwei Knoten aus dem Graphen zur Findung verwendet),
- $x_1, y_1, \ldots, x_n, y_n \in \mathcal{V}$,
  (Die Knotenvariablen zur Findung des Subgraphen entstammen der Menge der Knotenvariablen),
- $a_1, \ldots, a_n \in \mathcal{L}$,
  (Die Labels zur Findung des Subgraphen entstammen der Menge aller Labels),
- $z_i \in \{x_1, y_1, \ldots, x_n, y_n\} \; \forall \; 0 < i \leq m$.
  (Die Menge der gefundenen Knoten des Subgraphen ist eine Teilmenge der Quellknoten).

m wird die Stelligkeit oder Arität des Ausdrucks genannt.

Evaluation:

Sei $G = (V, E, \eta, \lambda, v)$ ein Property-Graph und $r = (z_1, \ldots, z_m) \leftarrow a_1(x_1, y_1), \ldots, a_n(x_n, y_n)$ eine Conjunctive Graph Query. Eine Abbildung für r auf G ist eine Funktion μ mit Definitionsbereich $\mathcal{V}$ und Wertebereich V, sodass $\forall \; 1 \leq i \leq n \; \exists$ Kante$_i \in E : \eta($Kante$_i) = (\mu(x_i), \mu(y_i)) \; \wedge \; a_i \in \lambda($Kante$_i)$. Die Evaluation von r auf G ist die m-stellige Relation $[\![r]\!]_G \subseteq \underbrace{V \times \cdots \times V}_{m \text{ mal}}$, welche wie folgt definiert ist:

$$[\![r]\!]_G = \{(\mu(z_1), \ldots, \mu(z_m)) \mid \mu \text{ ist eine Abbildung von } r \text{ auf } G\}$$

Beispiel Die Anfrage

$$q = (a_1, e_2) \leftarrow :\text{studiert}(a_1, a_2), :\text{betreut}(e_1, a_1), :\text{beinhaltet}(a_2, e_2), :\text{lehrt}(e_1, e_2)$$

auf Graph G aus Abbildung 4.2 ergibt

$$[\![q]\!]_G = \{(10, 15)\}$$

Hierbei werden alle Knotenpaare gesucht, welche durch Kanten mit den Labels :studiert, :betreut, :beinhaltet und :lehrt verbunden sind. Dabei gilt das Kriterium, dass die Zielknoten von :betreut ebenfalls Quellknoten von :studiert, die Quellknoten von :lehrt ebenfalls Quellknoten von :betreut und die Zielknoten von :lehrt ebenfalls Zielknoten von :beinhaltet sein

müssen. Diese Einschränkungen stellen damit einen Subgraphen von G dar. Als Ergebnis werden dann Knotenpaare aus den Quellknoten von :studiert und Zielknoten von :beinhaltet gebildet. Durch die genannten Prädikate ist sichergestellt, dass sich diese Knoten im Subgraphen befinden und direkt oder indirekt miteinander verbunden sind.

4.1.3. Conjunctive Regular Path Queries

Conjunctive Regular Path Queries (CRPQ) kombinieren RPQs und CQs zu einer gemeinsamen Anfrage. Damit bilden CRPQs eine Obermenge von sowohl RPQs als auch CQs (siehe Abbildung 4.1). Die Definition gleicht der von CQs, wobei die Anfrage nun nicht mehr nur Labels sondern auch direkt RPQs enthalten kann.

Definition 4.3 (Conjunctive Regular Path Queries) *Sei $\mathcal{V}$ eine Menge von Knotenvariablen. Dann ist eine CRPQ ein Ausdruck der Form*

$$(z_1, \ldots, z_m) \leftarrow \alpha_1(x_1, y_1), \ldots, \alpha_n(x_n, y_n)$$

wobei

- $m \geq 0, n > 0$,
- $x_1, y_1, \ldots, x_n, y_n \in \mathcal{V}$,
- $\alpha_1, \ldots, \alpha_n \in RPQ$,
- $z_i \in \{x_1, y_1, \ldots, x_n, y_n\} \; \forall \; 0 < i \leq m$.

Evaluation:

Sei $G = (V, E, \eta, \lambda, v)$ ein Property-Graph und $r = (z_1, \ldots, z_m) \leftarrow \alpha_1(x_1, y_1), \ldots, \alpha_n(x_n, y_n)$ eine Conjunctive Regular Path Query. Eine Abbildung für r auf G ist eine Funktion μ mit Definitionsbereich $\mathcal{V}$ und Wertebereich V, sodass $\forall \; 1 \leq i \leq n : (\mu(x_i), \mu(y_i)) \in [\![\alpha_i]\!]_G$.

Beispiel Da es sich bei RPQs und CQs um Teilmengen von CRPQ handelt, sind die Beispiele aus Abschnitten 4.1.1 und 4.1.2 auch hier gültig. Umgekehrt gilt dies jedoch nicht: die folgende Anfrage q kann so nur in CRPQ gestellt werden:

$q = (n, e) \leftarrow :\text{studiert}(n, a_1), :\text{beinhaltet}^+(a_1, e), :\text{kennt}(n, a_2), :\text{betreut}(a_2, a_3), :\text{korrigiert}(a_3, e)$

Ausgewertet auf Graph G aus Abbildung 4.2 ergibt diese Anfrage

$$[\![q]\!]_G = \{(10, 16)\}$$

Diese Anfrage vereint Elemente aus RPQ und CQ. Es wird wie in Beispiel 4.1.2 ein Subgraph gebildet und aus diesem Graphen Knotenpaare als Ergebnis ausgeben. Allerdings sind nun zur Bildung des Graphen auch Ausdrücke aus RPQ zulässig, beispielsweise transitive Ausdrücke wie :beinhaltet$^+$, welche auch im Beispiel aus Absatz 4.1.1 beschrieben sind.

4.1.4. Unions of Conjunctive Regular Path Queries

UCRPQs erweitern CRPQs durch Disjunktionen. Dies bedeutet, dass zur Findung der Ergebnismenge an Knoten mehrere CRPQs verwendet werden können. Ein Ergebnisknoten muss dann eine dieser Anfragen erfüllen. UCRPQs stellen die Grundlage der in den nächsten Abschnitten näher beschriebenen Graphenalgebren dar.

Definition 4.4 (Unions of Conjunctive Regular Path Queries) *Eine UCRPQ ist eine endliche, nicht-leere Menge $R \subseteq CRPQ$, wobei jedes Element der Menge dieselbe Arität m aufweist.*

Evaluation:

Sei $G = (V, E, \eta, \lambda, v)$ ein Property-Graph. Für $R \in UCRPQ$ mit Arität m ist die Evaluation von R auf G die m-stellige Relation $[\![R]\!]_G \subseteq \underbrace{V \times \cdots \times V}_{m \text{ mal}}$, welche wie folgt definiert ist:

$$[\![R]\!]_G = \bigcup_{r \in R} [\![r]\!]_G$$

Beispiel Die folgende UCRPQ auf Graph G aus Abbildung 4.2 evaluiert alle Personen x, welche eine Person in der Mitte kennen oder mit ihr arbeiten (welche nicht unbedingt von x unterschiedlich sein muss), welche eine Person betreut (welche nicht unbedingt von x oder der Person in der Mitte unterschiedlich sein muss):

$$(x) \leftarrow :\text{kennt}/:\text{betreut}(x, y)$$
$$(x) \leftarrow :\text{arbeitetMit}/:\text{betreut}(x, y)$$

Das Ergebnis dieser Anfrage ist

$$x = \{10, 13\}$$

4.2. Regular Property Graph Algebra

Zu einer besseren Untersuchung von Anfragesprachen und der Konstruktion von Anfragen bietet es sich an, eine algebraische Repräsentation aller möglichen Anfragen zu definieren. Das relationale Datenbankmodell besitzt mit der relationalen Algebra bereits eine solche

Repräsentation [27]. Bonifati Et Al. stellen in [1] einen Vorschlag für eine solche Algebra für Property-Graphen-Datenbankmodelle vor. Dabei wird zunächst ein logischer Teil, die Regular Property Graph Logic (RPGLog) vorgestellt, welche auf den im vorherigen Abschnitt erläuterten regulären Anfragen UCRPQ und RA basiert. Anschließend wird die Regular Property Graph Algebra (RPGQ) eingeführt, welche logisch äquivalent und gleichmächtig zur RPGLog ist und mit welcher Property-Graphen-Anfragen untersucht werden können.

Definition 4.5 (Regular Property Graph Algebra) *Unter der Regular Property Graph Algebra können alle und nur solche Anfragen konstruiert werden, welche aus Elementen von $\mathcal{L}$ bestehen und die transitive Hülle, die Vereinigung oder den Verbund als Operatoren nutzen. Dabei definieren wir zunächst die Grammatik der Menge von Sub-Anfragen (subRPGA):*

$$e \quad := \quad \ell \mid e^* \mid e \cup e \mid \bowtie^{\Phi,c}_{pos_i,pos_j} (e, \ldots, e)$$

wobei

- $l \in \mathcal{L}$*;*
- $(e, \ldots, e)$ *von der Länge* $n > 0$*;*
- $c \in C$ *ist ein Kontext-Identifikator;*
- $pos_i, pos_j \in \{src_1, trg_1, \ldots, src_n, trg_n\}$*;*
- Φ *ist eine Verkettung einer endlichen Anzahl an Termen der Form:*
  - $\lambda(pos) = \ell$ *für* $pos \in \{src_1, trg_1, \ldots, src_n, trg_n\}$ *und* $\ell \in \mathcal{L}$*,*
  - $pos_i.p \circ pos_j.q$ *oder* $pos_i.p \circ val$*,*
    für $pos_i, pos_j \in \{src_1, trg_1, \ldots, src_n, trg_n, edge_1, \ldots, edge_n\}, p, q \in \mathcal{K}, val \in \mathcal{N}$ *und* $\circ \in \{=, \neq, <, >, \leq, \geq\}$*,*
  - $pos_i = pos_j$ *für* $pos_i, pos_j \in \{src_1, trg_1, \ldots, src_n, trg_n\}$

Ausdrücke $e \in RPGA$ entsprechen dann allen und nur allen Ausdrücken der Form

$$\bowtie^{\Phi,c}_{\overline{pos}} (e_1, \ldots, e_n),$$

wobei $n > 0$, jedes e_i ein subRPGQ-Ausdruck $(1 \leq i \leq n)$, Φ wie oben definiert und $\overline{pos}$ eine Liste mit Länge ≤ 0 und Elementen der Form $\{src_1, trg_1, \ldots, src_n, trg_n\}$ ist. Diese Definition hält für Property-Graphen, in welchen Kanten genau ein Label besitzen.

Evaluation:

Sei $G = (V, E, \eta, \lambda, v)$ ein Property-Graph und $e \in subRPGA$. Dann wird e auf G, bezeichnet als $[\![e]\!]_G$, wie folgt ausgewertet:

- *Fall* $e = \ell \in \mathcal{L}$*:* $[\![e]\!]_G = \{\{\eta(Kante), Kante\} \mid Kante \in E \wedge \lambda(Kante) = \ell\}$*;*

- *Fall* $e = f \cup g$: $[\![f]\!]_G \cup [\![g]\!]_G$;
- *Fall* $e = f^*$: $[\![e]\!]_G = \{(x,x) \mid x \in V\} \cup \{(x,y) \mid x,y \in V \wedge x \xrightarrow{[\![f]\!]_G} y\}$, *wobei* $x \xrightarrow{[\![f]\!]_G} y$ *bedeutet, dass für* $n > 0$ *Ausdrücke* $e_1, \ldots, e_n \in [\![f]\!]_G$ *und* $v_1, \ldots v_{n+1} \in V$ *existieren, sodass* $x = v_1$, $y = v_{n+1}$ *und* $1 \leq i \leq n$, $(v_i, v_{i+1}) \in e_i$;
- *Fall* $e = \bowtie^{\Phi,c}_{pos_i,pos_j} (e_1, \ldots, e_n)$: *Für* $i \leq i \leq n$, *sei* $(src_i, trg_i, edge_i)$ *das Schema von* $[\![e_i]\!]_G$, *wobei für alle* $t = \{(x,y),z\} \in [\![e_i]\!]_G$ $t.src_i = x$, $t.trg_i = y$ *und* $t.edge_i = z$ *bekannt ist. Dann ist* $[\![e]\!]_G = \{\{(t.pos_i, t.pos_j), \omega_G(t.pos_i, t.pos_j, c)\} \mid t \in \sigma_\Phi([\![e_1]\!]_G \times \cdots \times [\![e_n]\!]_B\}$, *wobei für den Filter* Φ *folgendes gilt:*

  - „$\lambda(pos) = \ell$“ *ist wahr ausschließlich wenn* $\lambda(t.pos) = \ell$;
  - „$pos_i.p \circ pos_j.q$“ *ist wahr ausschließlich wenn* $v(t.pos_i, p) \circ v(t.pos, q)$;
  - „$pos_i.p \circ val$“ *ist wahr ausschließlich wenn* $v(t.pos, p) \circ val$;
  - „$pos_i = pos_j$“ *ist wahr ausschließlich wenn* $t.pos_i = t.pos_j$.

  σ *bezeichnet hier den Standard-Selektions-Operator in der relationalen Algebra, welcher in Definition 4.6 näher definiert wird.*

Aus diesen Fällen ergibt sich die Evaluation eines RPGA-Ausdrucks $\bowtie^{\Phi}_{pos_1,\ldots,pos_m} (e_1, \ldots, e_n) \in$ *RPGA auf G:*

$$[\![\bowtie^{\Phi}_{pos_1,\ldots,pos_m} (e_1, \ldots, e_m)]\!]_G = \{(t.pos_1, \ldots, t.pos_m) \mid t \in \sigma_\Phi([\![e_1]\!]_G \times \cdots \times [\![e_n]\!]_G)\}$$

Diese Algebra ist ausreichend, um alle möglichen Anfragen, welche an einen Property-Graphen gestellt werden können, abzubilden. Ein Property-Graph ist sehr grob mit einer relationalen Datenbank vergleichbar: jeder Knoten entspricht einem eigenen Schema der Datenbank. Dadurch wird deutlich, dass der Verbund $\bowtie$ hier die grundlegendste Operation darstellt. Selektion geschieht durch Filter in der Anfrage, welche innerhalb von Φ definiert werden. Ein Unterschied zu relationalen Datenbanken besteht in der fehlenden Projektion auf einzelnen Knoten: da kein festes Schema existiert, an welches sich alle Knoten orientieren, ist eine Projektion auf Attributnamen nicht ohne weiteres möglich.

Durch die rekursive Definition der Algebra kommt die Frage nach den Definitions- und Wertebereichen der einzelnen Operatoren auf. Diese ergeben sich aus der Reihenfolge der Auswertung der einzelnen *subRPGA*s. Die innersten Ausdrücke werden dabei zuerst ausgewertet. Diese entsprechen den gültigen Ausdrücken einer UCRPQ. Alle folgenden Ausdrücke und Operatoren, welche sich auf die bereits ausgewerteten *subRPGA*s beziehen, besitzen als Definitionsbereich alle Knoten und Kanten des Subgraphens, welcher von den ausgewerteten Operatoren aufgestellt wurde, sowie deren Labels und Properties. Der Wertebereich von Operatoren ist demnach folgend eine unechte Teilmenge ihres Definitionsbereiches. Somit wird

Bezeichner	Erklärung
e	Genereller Ausdruck, kann einer der nachfolgenden Ausdrücke sein.
ℓ	Label aus der Menge aller Label $\mathcal{L}$.
e^*	Transitive Hülle eines Ausdrucks e, definiert wie in Abschnitt 4.1.1.
$e \cup e$	Vereinigung der Ergebnisse zweier Ausdrücke, definiert wie in Abschnitt 4.1.4.
$\bowtie^{\Phi,c}_{pos_i,pos_j}(e,\ldots,e)$	subRPQ. Unteranfrage, welche in die Elternanfrage geschachtelt wird. Ermöglicht Erstellen und Eingrenzen von Teilgraphen in Anfragen.
c	Kontext-Identifikator aus der Menge aller Kontext-Identifikatoren $\mathcal{C}$. Ermöglicht die Identifikation oder Trennung von Kanten, welche durch unterschiedliche Anfragen erstellt und angefragt wurden. Dies spielt vor allem für Anfragen an unterschiedliche Graphen eine Rolle. Diese Technik wird von GQL mit composite statements unterstützt.
Φ	Menge an Filtern, welche die angefragten Knoten und Kanten durch Prädikate weiter einschränken.

Tabelle 4.1.: Übersicht über die einzelnen Algebra-Bestandteile

der Bereich, auf dem die Operatoren wirken können, immer weiter eingeschränkt und ist ursprünglich der Definitions- und Wertebereich einer UCRPQ auf dem betrachteten Graphen. Aus dieser Definition ergibt sich schließlich, dass die Ein- und Ausgaben aller möglichen Operatoren und Operationen immer gültige Graphen, genauer Teilgraphen des untersuchten ganzen Graphen der Graphdatenbank, darstellen, was ein relevantes Kriterium für die logische Konsistenz der Graphenalgebra darstellt.

4.2.1. Weitere Operatoren

In der Optimierung von Datenbanken-Anfragen ist es Standard, Selektionen möglichst spät auszuführen [27, 28]. Um solche Selektionen einfacher und besser erkennen zu können, ist es sinnvoll, einen Selektions-Operator zu definieren:

Definition 4.6 (Selektions-Operator) *Sei Ψ ein Prädikat $\lambda(pos) = \ell$ oder $pos.p \circ val$. Dann ist $\sigma_\Psi(e)$ ein Werte-Selektions-Operator für subRPGA, welcher dem Ausdruck $\bowtie^{\Psi}_{src_1,trg_1}(e)$ entspricht.*

Eine weitere sinnvolle Ergänzung stellt der Pfad-Projektions-Operator $\pi(e)$ dar. Mit diesem lassen sich Knoten auswählen, welche eine ausgehende Kante an einen Knoten besitzen. Das Filter-Kriterium ist dabei e.

Definition 4.7 (Pfad-Projektions-Operator) *Der Pfad-Projektions-Operator $\pi(e)$ wird wie folgt evaluiert:*

$$[\![\pi(e)]\!]_G = \{(s,s) \mid \exists t \in V : (s,t) \in [\![e]\!]_G\}$$

Ein Beispiel hierfür wäre die Anfrage $q = \pi(:\text{kennt}^+/:\text{arbeitetFür})$ an den Graphen aus Abbildung 2.2, welche als Ergebnis $\{(11,11)\}$ zurückgibt. Diese Anfrage sucht alle Knoten, welche eine Kante mit Label „:kennt“ mit einem Knoten teilen, welcher wiederum eine Kante mit Label „:arbeitetFür“ mit einem anderen (nicht notwendigerweise unterschiedlichen) Knoten teilt.

4.3. Erweiterung auf Hypergraphen

Bei genauer Betrachtung des Property-Graphen-Modells aus Definition 2.3 und des Hypergraphen-Modells von PowerGraph aus Abschnitt 3.2 wird deutlich, dass nur wenige Anpassungen an die Graphenalgebra des vorherigen Abschnitts notwendig sind. So entsprechen die Typen von PowerGraph den Labels eines Property-Graphen und die Attribute den Properties. Daher bedarf es hier keiner Anpassung. Jedoch benennen wir $\ell \in \mathcal{L}$ zur besseren Übersicht und als Abgrenzung zur RPGA in $g \in \Gamma$ und $\lambda(pos) = \ell$ in $\gamma(pos) = g$ um.

$$g \in \Gamma$$
$$\gamma(pos) = g$$

Hierbei muss darauf geachtet werden, dass Knoten und Kanten in PowerGraph nicht nur Element der Knoten- und Kantenmengen sondern auch durch eine Rolle definiert werden: $v \in V \times R, e \in A \times R$. Da in PowerGraph sowohl Knoten als auch Kanten Attribute aufweisen können, erweitern wir den Definitionsbereich von γ wie folgt:

$$\gamma(pos) = g, \quad pos \in (V \times R) \cup (A \times R)$$

Ein weiterer Unterschied ergibt sich aus den unterschiedlichen Definitionen von Kanten. Während bei Property-Graphen eine Kante maximal zwei Knoten miteinander verbindet, können Hyperkanten beliebig viele Knoten beinhalten. Daher können Kanten als eine Menge von Knoten betrachtet werden. Es ergibt daher Sinn, die Start- und Endknoten einer Anfragen nicht mehr nur als einzelne Knoten zu betrachten, sondern auch Mengen von Start- und Endknoten zuzulassen. So kann dieselbe Syntax für Anfragen sowohl für Knoten als auch für Kanten verwendet werden.

$$pos_i, pos_j \subseteq \{\text{src}_1, \text{trg}_1, \ldots, \text{src}_n, \text{trg}_n, \text{edge}_1, \ldots, \text{edge}_n\}$$

Dies bedeutet, dass sich bei einem Filter mit Verknüpfung $\circ$ die Attribute von pos_i und pos_j nun nicht mehr auf nur einen Knoten oder eine Kante, sondern auf die Menge der Attribute innerhalb der Menge der Knoten und Kanten bezieht.

Die Menge der Typen von Knoten und Hyperkanten sind in PowerGraph disjunkt. Dies ermöglicht es, Typen mit gleichem Namen für beide Mengen einzuführen. Da die angepasste Syntax für Anfragen sowohl Knoten als auch Kanten verwendet, ist es notwendig zu spezifizieren, auf welche Typenmenge sich ein angegebener Typ bezieht. Dies ermöglichen wir mit einem zusätzlichen Qualifikator, den wir, falls notwendig, vor den Typbezeichner stellen:

$$g \in \Gamma, \text{falls Auswahl von Typen für Knoten und Kanten}$$
$$node.g \in \Gamma^V, \text{falls Auswahl von Knotentyp}$$
$$edge.g \in \Gamma^A, \text{falls Auswahl von Kantentyp}$$

Aus diesen Erweiterungen ergibt sich schließlich die folgende Grammatik für eine Regular Hypergraph Algebra:

Definition 4.8 (Regular Hypergraph Algebra)

$$e \quad := \quad g \mid e^* \mid e \cup e \mid \ \bowtie^{\Phi,c}_{pos_i,pos_j} (e, \ldots, e),$$

wobei

- $g \in \Gamma, node.g \in \Gamma^V, edge.g \in \Gamma^A$*;*
- $(e, \ldots, e)$ *von der Länge* $n > 0$*;*
- $c \in C$ *ist ein Kontext-Identifikator;*
- $pos_i, pos_j \subseteq \{src_1, trg_1, \ldots, src_n, trg_n, edge_1, \ldots, edge_n\}$*;*
- $edge \in \{edge_1, \ldots, edge_n\}$*;*
- Φ *ist eine Verkettung einer endlichen Anzahl an Termen der Form:*
  - $\gamma(pos) = g$ *für* $pos \subseteq \{src_1, trg_1, \ldots, src_n, trg_n\}$ *und* $g \in \Gamma$*;*
  - $pos_i.p \circ pos_j.q$ *oder* $pos_i.p \circ val$*,*
    für $pos_i, pos_j \subseteq \{src_1, trg_1, \ldots, src_n, trg_n, edge_1, \ldots, edge_n\}, p, q \in \mathcal{K}, val \in \mathcal{N}$
    und $\circ \in \{=, \neq, <, >, \leq, \geq\}$*,*
  - $pos_i = pos_j$ *für* $pos_i, pos_j \subseteq \{src_1, trg_1, \ldots, src_n, trg_n\}$

PowerGraph besitzt wie Property-Graph-Modelle auch Properties, welche sowohl Knoten als auch Kanten zugewiesen werden können. Der Selektions-Operator Ψ kann somit unverändert aus Definition 4.6 übernommen werden, da die erweiterte Hypergraphen-Algebra keine Änderungen an diesen Properties vornimmt.

Ähnlich verhält es sich mit dem Pfad-Projektions-Operator π aus Definition 4.7. Das sowohl Property-Graphen- als auch Hypergraphen-Modelle in ihren Grundlagen Graphen darstellen und die Labels aus Property-Graphen den Typen aus PowerGraph entsprechen, ist auch hier ein solcher Operator sinnvoll. Allerdings muss darauf geachtet werden, dass es sich bei den Start- und Endknoten nun nicht mehr um einzelne Knoten sondern Mengen von Knoten handelt. Dies führt dazu, dass s und t in der Definition von π nun Teilmengen von V sind. Somit wird die Hyperkanten-Eigenschaft von PowerGraph abgebildet.

$$[\![\pi(e)]\!]_G = \{(s,s) \mid \exists t \subseteq V : (s,t) \in [\![e]\!]_G\}$$

Im Gegensatz zur relationalen Algebra wird in dieser Arbeit vorerst kein Projektionsoperator für Attributprojektionen definiert. Dies ist darin begründet, dass für Hyperkanten keine einfache Projektion auf einzelne Attribute stattfinden kann, da unter Umständen auf Konten unterschiedlichen Typs projiziert wird. Dies würde zu einer inkonsistenten Definition des Definitions- und Wertebereichs führen, da unterschiedliche Knotentypen unterschiedliche Attribute besitzen können. Bei einer Projektion auf einen festen Knotentypen eignet sich ein Projektionsoperator vom Typ

$$\Pi(\text{pos}) = (p_1, \ldots, p_n),\ p \in \mathcal{K}$$

Diese Art von Projektion wird in der Implementation dieser Arbeit verwendet, da HyperGQL nur Projektionen auf einen bestimmten Knotentypen erlaubt. Für Knotenmengen mit Knoten unterschiedlichen Typs bestehen hier Weiterentwicklungsmöglichkeiten der Graphenalgebra.

4.3.1. Beispiele

Die folgenden Beispiele können mit der Hypergraphenalgebra aus dem vorherigen Abschnitt dargestellt werden. Die Knoten- und Kantentypen sind dabei dem WossiDiA-Projekt entnommen, die Anfragen können so also tatsächlich an das System gestellt werden. Weitere Beispiele finden sich in Tabelle 4.2 sowie in Abschnitt 5.4.

Beispiel Selektiere alle Knoten vom Typ `am_place`, bei welchen das Attribut `name` „Zapel“ und das Attribut `near` „Crivitz“ entspricht.

$$\ltimes^{\emptyset}_{src_1} (\sigma_{\Psi}(node.\text{am\_place}))$$
$$\Psi := \text{name} = \text{“Zapel”},\ \text{near} = \text{“Crivitz”}$$

Beispiel Selektiere alle Wortknoten, welche über eine `content`-Hyperkante mit einem Ort verbunden sind und die Attribute `name` = „Zapel“ und `near` = „Crivitz“ besitzen.

$$\ltimes^{\emptyset}_{trg_1} (\ltimes^{\emptyset}_{src_1,trg_1} (\sigma_{\Psi}(edge.\text{content})))$$
$$\Psi := \gamma(src_1) = \text{am\_place},\ \gamma(trg_1) = \text{w},\ src_1.name = \text{“Zapel”},\ src_1.near = \text{“Crivitz”}$$

4.4. Verbindung zu HyperGQL

HyperGQL [23] stellt eine Erweiterung der GQL-Grammatik. Dabei wird die GQL-Semantik wesentlich vereinfacht und um PowerGraph-Konzepte erweitert. Der folgende Abschnitt beschreibt die wichtigsten Änderungen.

Die erste Abänderung von HyperGQL stellt der Verzicht auf gerichtete Kanten in einzelnen Patterns dar. Dies wird damit begründet, dass bei Angabe von Quell- und Zielknoten immer die zugrundeliegende ID im Voraus bekannt sein muss. Somit ist eine Suche nach gerichteten Kanten hinfällig, da ohne Angabe einer eindeutigen ID Anfragen kein Ergebnis liefern würden.

Eine weitere Einschränkung ist der Verzicht auf die Angabe von mehreren Graphen in Anfragen, da die PowerGraph-Datenbank aus nur einem einzigen Graphen besteht. Somit wären Anfragen auf mehrere unterschiedliche Graphen hinfällig.

HyperGQL erlaubt keine lokalen Deklarationen, da Unteranfragen als Eingabegraphen definiert werden können. Darüber hinaus war es auch Ziel, eine simple Anfragesprache zu schaffen, welche Aspekte wie Funktionen oder die Definition von bevorzugten Pfaden außer Acht lässt. Ebenso fehlen Gruppierungs-, Aggregations- und Sortierungsfunktionalitäten, wie sie beispielsweise aus SQL bekannt sind und auch im ersten Definitionsvorschlag von GQL [24] vorkommen.

Zuletzt erlaubt HyperGQL die direkte Anfrage von Knoten und Kanten über ihren Typen. Hierfür werden die Schlüsselwörter **`NODES`** und **`EDGES`** definiert, welche über eine Angabe des gesuchten Typs die Ausgabe aller Knoten und Kanten in der PowerGraph-Datenbank unterstützen.

4.4.1. Abbildung auf RHGA

HyperGQL lässt sich fast direkt auf die relationale Hypergraphenalgebra aus Definition 4.8 abbilden. Dies liegt zum einen daran, dass sowohl HyperGQL als auch RHGA für den Einsatz auf dem WossiDiA-PowerGraph-Modell konzipiert sind. Zum anderen stellt RHGA eine Ableitung und Erweiterung der relationalen Property-Graphen-Algebra aus Definition 4.5 dar. Bei der Erstellung dieser Algebra durch Bonifati et al. wurde besonderer Wert auf die Abbildung kontemporärer Graphanfragesprachen wie beispielsweise G-CORE auf Algebra-Ausdrücke gelegt, da ein wesentlicher Motivationsgrund die mathematische Untersuchung

solcher Anfragesprachen war. HyperGQL „erbt“ diese Eigenschaften der RPGA, wodurch eine unkomplizierte Umformung von HyperGQL-Anfragen zu RHGA-Ausdrücken und umgekehrt möglich ist.

Die Abbildung von HyperGQL-Bestandteilen in RHGA-Ausdrücke ist wie folgt:

- **`FROM`** `g` ist wie in den oberen Abschnitten beschrieben implizit;
- Pattern Matching erfolgt mithilfe von Joins und Ausdrücken aus der in Definition 4.4 beschriebenen UCRPQ, beispielsweise

```
(a:KnotenA) -[:Kante]- (b:KnotenB)
```

zu

$$\bowtie_{a,b}^{\Phi,x} (Kante),$$
$$\Phi := \gamma(a) = KnotenA,\ \gamma(b) = KnotenB$$

- Filter werden mithilfe von Filter-Prädikaten umgesetzt, das heißt **`WHERE`** $= \Phi$;
- RHGA unterstützt nur die Rückgabe von ganzen Knoten, d.h. jeder **`RETURN`**-Ausdruck wird als **`RETURN`** `*` interpretiert.

In Tabelle 4.2 sind einige Beispiele aus [23] in HyperGQL aufgelistet und ihre RHGA-Äquivalente dargestellt.

Test	HyperGQL	RHGA

Node	`FROM g`
	`NODES 10`
	`WHERE name = 'Zapel';`
		$\bowtie^{\emptyset}_{\mathrm{src}_1} (\sigma_{\Psi}(node.10))$,
		$\Psi := \gamma(\mathrm{src}_1) = node.10,\ \mathrm{src}_1.name = \text{`Zapel}'$

Edge	`FROM g` `EDGES fieldtrip;`	$\bowtie^{\emptyset}_{\mathrm{src}_1} (\sigma_{\Psi}(edge.fieldtrip))$, $\Psi := \gamma(\mathrm{src}_1) = edge.fieldtrip$

Pattern Correctness	`FROM g`
	`MATCH (a:am_place) -[:content]- (c:18)`
	`WHERE a.name = 'Zapel'`
	`AND a.near = 'Crivitz'`
	`AND c.word = 'Mittelpunkt'`
	`RETURN a.name, a.near, c.word;`
		$\bowtie^{\Phi,x}_{\mathrm{src}_1,\mathrm{trg}_1} (\sigma_{\Psi}(edge.content))$,
		$\Phi := \gamma(\mathrm{src}_1) = node.am\_place,\ \gamma(\mathrm{trg}_1) = node.18$
		$\Psi := \mathrm{src}_1.name = \text{`Zapel}',\ \mathrm{src}_1.near = \text{`Crivitz}',$
		$\mathrm{trg}_1.word = \text{`Mittelpunkt}'$

Pattern Free	`FROM g`
	`MATCH (p:am_place) -[:content]- (w:18)`
	`WHERE p.name = 'Zapel'`
	`AND p.near = 'Crivitz'`
	`RETURN w.word;`
		$\bowtie^{\emptyset}_{\mathrm{trg}_1} (\bowtie^{\Phi,x}_{\mathrm{src}_1,\mathrm{trg}_1} (\sigma_{\Psi}(edge.content)))$,
		$\Phi := \gamma(\mathrm{src}_1) = node.am\_place,\ \gamma(\mathrm{trg}_1) = node.18$
		$\Psi := \mathrm{src}_1.name = \text{`Zapel}',\ \mathrm{src}_1.near = \text{`Crivitz}'$

Return All	`FROM g`
	`MATCH (r:am_place) -[:content]- (w:am_word)`
	`WHERE w.word = 'Werwolf'`
	`RETURN r.*;`
		$\bowtie^{\emptyset}_{\mathrm{trg}_1} (\bowtie^{\Phi,x}_{\mathrm{src}_1,\mathrm{trg}_1} (\sigma_{\Psi}(edge.content)))$,
		$\Phi := \gamma(\mathrm{src}_1) = node.am\_place,\ \gamma(\mathrm{trg}_1) = node.18$
		$\Psi := \mathrm{trg}_1.word = \text{`Werwolf}'$

Tabelle 4.2.: Beispiel-Anfragen in HyperGQL und regulärer Hypergraphen-Algebra

5. Implementierung

Das folgende Kapitel beschreibt eine Implementierung der im vorherigen Kapitel erarbeiteten Konzepte, namentlich der regulären Hypergraphenalgebra und ihrer Abbildung auf die in [23] vorgestellte modifizierte Version von GQL, HyperGQL. Dabei wird zunächst auf die Struktur der WossiDiA-REST-Schnittstelle eingegangen, welche als Anfrageschnittstelle zwischen der Graphdatenbank WossiDiA und dieser Implementierung dient. Darauf folgt eine Einführung in den Parser-Generator ANTLR, welcher zum Parsen der HyperGQL-Grammatik verwendet wurde. Anschließend wird die Struktur der Implementierung beschrieben und ihre Funktionalitäten anhand von beispielhaften Anfragen getestet. Schließlich folgt eine Reihe von Verbesserungsvorschlägen an die REST-API, welche im Verlaufe des Implementierungsprozesses aufgefallen sind.

5.1. Die WossiDiA-REST-API

Die in diesem Kapitel beschriebene Implementierung nutzt zur Anfrage der Daten der PowerGraph-Datenbank eine sogenannte REST-Schnittstelle. REST ist ein Akronym für **RE**presentational **S**tate **T**ransfer und stellt eine Form von Architektur von öffentlichen Schnittstellen für Hypermedia-Systeme dar. Häufig handelt es sich bei dieser Art von Schnittstellen um HTTP-Schnittstellen. Wie in [29] beschrieben werden folgende Prinzipien bei der Konzipierung von REST-Schnittstellen verwendet:

- Einheitliches Interface: Die Schnittstelle muss klar und hierarchisch strukturiert sein.
- Client-Server-Architektur
- Zustandslose Anfragen: Der Server darf keine Daten über mehrere Anfragen hinweg zur Auswertung weiterer Anfragen speichern.
- Cachebare Anfragen

Die WossiDiA-REST-API ist zum Zeitpunkt dieser Arbeit über die URL `http://api.wossidia.de` zu erreichen und stellt Endpunkte bereit, welche die Anfrage an Knoten- und Kantentypen sowie Knoten und Kanten mit optionalen Filterkriterien ermöglichen. Außerdem besteht die Möglichkeit, sich über spezielle Pfadausdrücke Teilgraphen der Datenbank ausgeben zu lassen, wovon in dieser Arbeit allerdings nicht Gebrauch gemacht wurde. Die folgende Tabelle 5.1 gibt einen Überblick über die genutzten Endpunkte in der Implementierung

sowie ihr Nutzen innerhalb der Umsetzung der Graphenalgebra. Eine genauere Beschreibung der Vorgehensweise und der einzelnen Verarbeitungsschritte und Anfragen folgt in Abschnitt 5.3.

Algebra-Ausdruck	**REST-Pfad**	**Pfad-Argumente**
$\gamma(pos) = g$	`/nodes/{type}`	`{type}` $= g$
$pos_i.p \circ val$	`/nodes/{type}/{cond}`	`{type}` $= g$ aus Typ-Prädikat, `{cond}` $=$ `@`$p\circ$`'`val`'`
$pos_i.p \circ pos_j.q$	`/nodes/{type}/{cond}`	`{type}` $= g$ aus Typ-Prädikat, `{cond}` $=$ `@`$p\circ$`'`val`'` für $q = val$
$pos_i = pos_j$	`/nodes/{type}/{cond}`	`{type}` $= g$ aus Typ-Prädikat, `{cond}` $=$ `id='`g`'` für $\gamma(pos_j) = g$

Tabelle 5.1.: Abbildung von Algebra-Filtern auf Endknoten der REST-API

5.2. ANTLR

Die in dieser Arbeit beschriebene Implementierung nutzt zum Parsen der HyperGQL-Eingaben den Open-Source-Parser-Generator ANTLR [30]. Dieser parst mithilfe einer vorher festgelegten Grammatik den HyperGQL-Eingabetext des Nutzers in Tokens und schließlich einen Parse-Tree. Anschließend werden Java-Klassen generiert, welche die Weiterverarbeitung der Parser-Ergebnisse erlauben. Abbildung 5.1 stellt die einzelnen Verarbeitungsschritte des ANTLR-Parser-Generators dar.

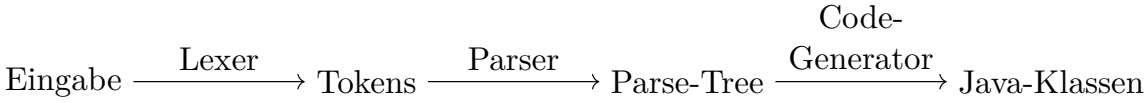


Abbildung 5.1.: Einzelne Verarbeitungsschritte von ANTLR

Grammatik Zu Beginn der Nutzung von ANTLR wird eine Grammatik definiert. Diese beschreibt die zu parsende Sprache und folgt vom Tool vordefinierten Prinzipien. Eine Grammatik besteht aus zwei Arten von Regeln: Lexer-Regeln und Parser-Regeln. Diese Regeln werden vom jeweiligen Verarbeitungsschritt ANTLRs genutzt um Muster im Eingabetext zu erkennen und zu verarbeiten. Die Regeln des jeweiligen Schrittes können dabei aufeinander aufbauen, um komplexe Grammatiken erstellen zu können. Dies wird dadurch erreicht, dass einzelne Regeln beliebig häufig wiederholt und aneinandergereiht werden können, um wiederum neue Regeln zu definieren. Die Implementation dieser Arbeit nutzt dabei die in [23] definierte ANTLR-Grammatik für HyperGQL.

Lexer Der Lexer, kurz für Lexical Analyzer, ist der erste Verarbeitungsschritt des Parser-Generators. Er nimmt als Eingabe den unstrukturierten Text des Nutzers und erzeugt mithilfe der in der Grammatik definierten Lexer-Regeln sogenannte Tokens. Diese Tokens stellen eine grobe Typisierung von Abschnitten der Zeichenkette des Nutzerinputs dar. So kann die Zeichenkette „FROM“ in das Token `FROM` umgewandelt werden. Damit ist definiert, dass es sich bei dieser Zeichenkette um das Schlüsselwort **`FROM`** handelt.

Parser Der Parser erhält als Eingabe die Tokens des Lexers und wendet auf diese die Parser-Regeln der Grammatik an. Zunächst wird geprüft, ob die Reihenfolge und Typen der Tokens den syntaktischen Regeln der Grammatik folgen. Anschließend wird ein Baum generiert, der sogenannte Parse-Tree. Dieser Parse-Tree repräsentiert eine strukturierte Anordnung der einzelnen Tokens und ordnet diese in die aufeinander aufbauenden Regeln der Grammatik ein. Abbildung 5.2 zeigt ein sehr einfaches Beispiel für einen solchen Parse-Tree für die Eingabe `(a:am_place)`. Mithilfe dieses Baumes erzeugt der Code-Generator schließlich Java-Klassen, welche zur weiteren Verarbeitung der Parser-Ergebnisse genutzt werden können.

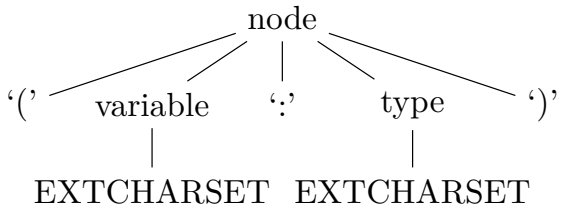


Abbildung 5.2.: Parse-Tree des Knotenausdrucks `(a:am_place)`

5.3. Ablauf einer Anfrage

Im folgenden wird der interne Ablauf einer Anfrage anhand eines Beispiels durchlaufen. Dabei wird auf die einzelnen Schritte eingegangen, welche von einer HyperGQL-Nutzereingabe zu einem auswertbaren Ergebnis führen. Die einzelnen Abschnitte erläutern hierbei interne Implementationsmechanismen sowie in Betracht gezogene alternative Ansätze. Die Auswertung der JSON-Antworten der REST-Schnittstelle erfolgt dabei durch das JSON-Serilisations-Deserilisations-Werkzeug GSON [31], welches JSON-Objekte und Arrays in Java-Objekte umformen kann.

5.3.1. Parsen der HyperGQL-Eingabe

```
FROM g
MATCH (a:am_place) -[:content]- (c:18)
WHERE a.name = 'Zapel'
    AND a.near = 'Crivitz'
    AND c.word = 'Mittelpunkt'
```

```
RETURN a.name, a.near, c.word;
```

Listing 5.1: Beispiel HyperGQL

Die Anfrage unseres Beispiels beginnt zunächst mit einer Texteingabe des Nutzers in Form eines HyperGQL-Ausdrucks, beispielsweise der Ausdruck in Listing 5.1. Nach Klicken des Buttons `Evaluate Query`, dargestellt in Abbildung 5.3a, nimmt das Programm die Eingabe des Textfeldes „Input“ entgegen und startet einen Parse-Vorgang der im Textfeld enthaltenen HyperGQL-Anweisung. Dieser Vorgang findet einem Objekt der Klasse `RequestBuilder` statt, welcher als Parameter den Typenkatalog des WossiDiA-Projektes entgegennimmt, welcher immer beim Start der Anwendung geladen und von dort an für weitere Anfragen zur Verfügung steht, und erzeugt als Ausgabe über die Funktion `RequestBuilder::toRequest()` ein im nächsten Abschnitt näher beschriebenes `Request`-Objekt. Für den Parse-Vorgang wird das in Abschnitt 5.2 beschriebene Werkzeug ANTLR verwendet. Dabei wird von ANTLR zunächst ein Parse-Baum erstellt. Anschließend wird durch mittels eines Automaten aus den Elementen des Baumes ein neuer `Request` zusammengestellt und wiedergegeben.

5.3.2. Erstellen einer RHGA-Anfrage

```
GraphVariable a = new GraphVariable("a",
        new GraphType(GraphType.ElementType.NODE, "am_place"));
GraphVariable c = new GraphVariable("c",
        new GraphType(GraphType.ElementType.NODE, 18));
GraphVariable edge = new GraphVariable("@0",
        new GraphType(GraphType.ElementType.EDGE, "content"));

Response response = new Request(catalog)
        .join(new JoinOperator(a, c, edge))
        .select(new TypePredicate(a))
        .select(new TypePredicate(c))
        .select(new OperatorValuePredicate(a, "name", "Zapel", OperatorType.EQUAL))
        .select(new OperatorValuePredicate(a, "near", "Crivitz", OperatorType.EQUAL))
        .select(new OperatorValuePredicate(c, "word", "Mittelpunkt", OperatorType.EQUAL))
        .project(new ProjectionOperator("a", "name"))
        .project(new ProjectionOperator("a", "near"))
        .project(new ProjectionOperator("c", "word"))
        .evaluate();
```

Listing 5.2: Beispielanfrage

Das im vorherigen Abschnitt beschriebene, von `RequestBuilder::toRequest()` zurückgegebene `Request`-Objekt ist in Listing 5.2 dargestellt. Hierbei sind zwei Tatsachen auffällig: Zum einen wird jede Graphenalgebra-Operation als eine eigene Methode des Objektes dargestellt. So kann beispielsweise ein neues Selektionsprädikat mit der Methode `Request::select()` hinzugefügt werden. Dazu wird ein neues Objekt erstellt, welches dem Selektionstypen entspricht. Anschließend werden diesem Objekt die Eigenschaften der Selektion übergeben. In aktuellen Beispiel sollen unter anderem Knoten ausgewählt werden, welche vom Typ „node.am_place“ sind und das Attributpaar „name = Zapel“ besitzen. Hierzu wird

zuerst die Variable a angelegt, welche die Knotenvariable des RHGA-Ausdrucks repräsentiert. Anschließend wird diese Variable durch die Prädikate

$$\gamma(a) = node.\text{am\_place und } a.name = \text{‘Zapel’}$$

weiter eingeschränkt, welche in Java mit den Ausdrücken

```
.select(new TypePredicate(a))
.select(new OperatorValuePredicate(a, "name", "Zapel", OperatorType.EQUAL))
```

beschrieben werden. Auf dieselbe Weise verhält es sich mit Projektionen, welche dem Objekt durch eine Angabe des Variablennamens und es gewünschten Attributs mit der Funktion `Request::project()` hinzugefügt werden können.

Die andere Tatsache, welche auffällt, ist, dass der Bau von Anfragen durch diese Oberfläche recht nahe an der mathematischen Definition von RHGA-Ausdrücken ist. Es sind fast keine Kenntnisse zu den internen Abläufen des Objekt notwendig, um eine neue Anfrage in Java per Hand zu definieren. Die HyperGQL-Anfragesprache bietet nur eine deklarative Abstraktionsebene, welche die Definition von `Request`-Objekten weiter vereinfacht. Auch ist die Reihenfolge der Operatoren für die Auswertung der Anfrage unerheblich, da die Implementation per Einschränkung von HyperGQL momentan nur einzelne Matchings erlaubt.

Die Anfrage kann schließlich durch Aufruf der `Request::evaluate()`-Methode ausgewertet werden, welche im folgenden Abschnitt beschrieben ist.

5.3.3. Auswertung der API-Antwort

Durch Aufruf der Methode `Request::evaluate()` wird die im vorherigen Abschnitt definierte Anfrage schließlich ausgewertet. Hierbei wird in folgender Reihenfolge vorgegangen:

1. Anfrage und Evaluation aller Verbunde;
2. Sortierung aller Selektionsprädikate nach ihren Variablen;
3. Auswertung aller Typenselektionen vom Typ $\gamma(\text{pos}_i) = g$;
4. Auswertung aller Attributsselektionen von Konstanten vom Typ $\text{pos}_i.p \circ val$;
5. Auswertung aller Attributsselektionen von anderen Knoten vom Typ $\text{pos}_i.p \circ \text{pos}_j.q$;
6. Auswertung aller Knotenselektionen vom Typ $\text{pos}_i = \text{pos}_j$;
7. Zusammenführung der Ergebnisse von Verbunden und Selektionen;
8. Auswertung von Projektionen.

Aus dieser Liste sind Punkte 5 und 6 nicht umgesetzt. Dies ist damit begründet, dass HyperGQL diese Art von Anfragen noch nicht unterstützt.

Anfrage und Evaluation aller Verbunde Zunächst werden alle definierten Verbunde ausgewertet. Hierzu wird die gesamte, in der Anfrage spezifizierte Kante von der API heruntergeladen und nach den Knotentypen der teilnehmenden Knoten gefiltert. Dabei ist dieser Teil häufig der Flaschenhals der gesamten Anfrage, da eine komplette Kante des WossiDiA-Systems heruntergeladen werden muss, was bei Kantentypen mit vielen teilnehmenden Kanten wie beispielsweise dem Typen `content` einen großen Aufwand des Systems erfordert. Auf das laufende Beispiel bezogen hätte eine solche Anfrage die URL `http://api.wossidia.de/edges/content`. Eine mögliche Optimierung stellt der API-Endpunkt `/graph/` dar, mit welchem sich einzelne Pfadausdrücke in der Graphdatenbank abfragen lassen. So könnte eine optimierte Form der Anfrage die URL `http://api.wossidia.de/graph/{Knoten-ID}/<content>18` darstellen, welche nur Knoten wiedergibt, die mit dem Knoten mit einer spezifischen Knoten-ID über die Kante `content` verbunden sind. Diese Art der Anfrage ist allerdings nur sehr schlecht dokumentiert und wurde daher in der Implementation nicht genutzt.

Auswertung von Selektionsprädikaten Als nächstes werden alle genannten Selektionsprädikate ausgewertet. Dies geschieht unabhängig von der Ergebnismenge der Verbundoperationen, da die WossiDiA-API Knoten-Anfragen mit Filtereigenschaften erlaubt, beispielsweise `http://api.wossidia.de/nodes/10/@name='Zapel'`. Somit kann eine Menge an Knoten erstellt werden, welche als Elemente alle Knoten enthält, die den Selektionsprädikaten entsprechen. Diese Menge wird später mit der Ergebnismenge der Verbundoperation vereinigt.

Zusammenführung der Ergebnisse von Verbunden und Selektionen Nachdem alle Verbunde und Selektionen evaluiert wurden, müssen die Ergebnisse dieser zusammengeführt werden. Es handelt sich hier in gewissem Maße um einen weiteren Verbund aller Teilergebnisse. Hierzu wird das Verfahren der Nested-Loop-Joins verwendet, welches eine sehr simple und ineffiziente Form von Verbundauswertung darstellt. Daher besteht hier ebenfalls die Möglichkeit der Optimierung, in dem effizientere Verfahren eingesetzt werden. Hierfür sind verschiedene Ansätze aus der relationalen Datenbanktechnik bekannt, beispielsweise Merge-Joins oder Hashverbunde [28]. Diese Verfahren erfordern allerdings komplexere und aufwändigere innere Datenstrukturen des Anfrageauswerters, zum Beispiel Hashtabellen oder spezielle Baumstrukturen, von welchen in dieser Implementierung kein Gebrauch gemacht wurde.

Auswertung von Projektionen Schließlich werden alle angegebenen Projektionen ausgewertet. Hierzu wird über alle Variablen iteriert, von welchen Attribute ausgegeben werden müssen. Anschließend werden die gewünschten Attribute in eine Ergebnismenge eingetragen

und vom `Request`-Objekt in Form eines `Response`-Objektes zurückgegeben. Dieses `Reponse`-Objekt kann nun vom Anwender ausgewertet werden. Im Prototypen geschieht dies durch die sortierte Ausgabe der Ergebnisse im Textfeld „Output“. Abbildung 5.5 veranschaulicht diese Ausgabe.

5.4. Tests

Der folgende Abschnitt testet die vorgestellte Implementierung anhand einer Reihe von Tests. Dabei werden alle möglichen Anfragetypen (Node-Queries, Edge-Queries und Pattern-Queries) getestet und die Ergebnisse anhand der vom WossiDiA-Projekt bereitgestellten Werkzeuge validiert. Dabei orientiert sich diese Arbeit an den von Manthey in [23] vorgeschlagenen Anfragen. Dies sei zum einen damit begründet, dass ihre Tauglichkeit zum Testen der Anfragetypen bereits gründlich eruiert wurde. Zum anderen handelt es sich bei dieser Arbeit um eine alternative Implementierung des selben Sprachvorschlags, HyperGQL. Es ist daher sinnvoll, die Fähigkeiten der Implementierungen an denselben Anfragen zu testen, um eine Vergleichbarkeit zwischen beiden Ansätzen zu schaffen.

Es muss darauf hingewiesen werden, dass es sich bei WossiDiA um ein System handelt, welches sich aktiv in der Entwicklung befindet. Daher können die hier vorgestellten Tests bei zukünftigen Ausführungen möglicherweise zu anderen Ergebnissen kommen. Die beschriebenen Tests wurden am 05. November 2022 ausgeführt.

5.4.1. Node-Query

Zunächst soll die Ausführung von Node-Queries getestet werden. Hierzu wird die Anfrage aus Listing 5.3 an das System gestellt. Ziel ist es, alle Knoten des Typs `am_place` zurückzugeben, welche das Attribut `name` mit dem Wert `Zapel` besitzen. Es werden also alle dokumentierten Orte mit dem Ortsnamen „Zapel“ im WossiDiA-System gesucht.

```
FROM g
NODES 10
WHERE name = 'Zapel';
```

Listing 5.3: Beispiel Node-Query HyperGQL

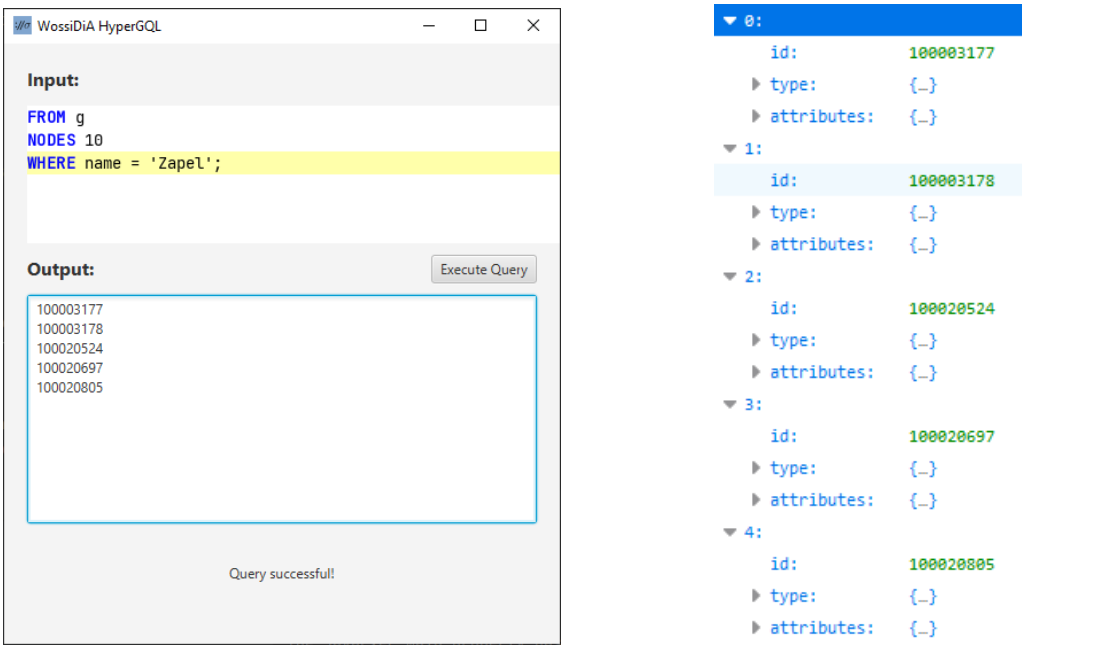
Die Implementierung parst die Anfrage aus Listing 5.3 und übersetzt sie in eine normalisierte RHGA-Anfrage. Diese ist in Listing 5.4 dargestellt. Dabei zeigt sich, dass eine Node-Query problemlos in einen RHGA-Ausdruck übersetzt werden kann, indem eine simple Typselektion nach dem angegebenen Knotentyp vorgenommen wird. Da HyperGQL nur Anfragen zwischen maximal zwei Knoten unterstützt, kann auch eine Attributsprojektion problemlos realisiert werden. Diese fehlt, wie im vorherigen Abschnitt beschrieben, in der in Definition 4.8 vorgestellten Graphenalgebra, ist in der Implementierung dennoch realisiert.

```
GraphVariable node = new GraphVariable("@0",
        new GraphType(GraphType.ElementType.NODE, 10));

Response response = Request(catalog)
        .select(new TypePredicate(node))
        .select(new OperatorValuePredicate(a, "name", "Zapel", OperatorType.EQUAL))
        .project(new ProjectionOperator("*"))
        .evaluate();
```

Listing 5.4: Beispiel Node-Query Java

Abbildung 5.3a zeigt die Ergebnisse der Anfrage. Hierbei werden die IDs der Ergebnisknoten im unteren Textfeld angezeigt. Die Beschränkung auf die Anzeige von IDs zur Ausgabe von Knoten ist gerade bei Anfragen an Knotentypen mit vielen Knoten und keiner Selektion sinnvoll, da eine zusätzliche Anzeige aller Knotenattribute schnell unübersichtlich werden würde. Diese Daten sind dennoch in der Antwort der WossiDiA-Schnittstelle mitenthalten und werden von der Implementierung ausgewertet. Sie können also durch eine Erweiterung des Quelltextes abgerufen werden.

Zur Validierung der Ergebnisse wird auf die REST-Schnittstelle der WossiDiA-API zurückgegriffen. Diese unterstützt, wie in den vorherigen Kapiteln beschrieben, Anfragen an Knotentypen mit der zusätzlichen Filterung von Attributen. Tatsächlich nutzt die Implementierung selber intern bereits diese Art von Anfragen. Die URL für die Anfrage aus Listing 5.3 lautet `http://api.wossidia.de/nodes/10/@name='Zapel'`. Diese gibt nach Aufruf ein JSON-Array zurück, welches in Abbildung 5.3b dargestellt ist. Andere Eigenschaften der Knoten sind der Übersichtlichkeit halber ausgeblendet. Die IDs beider Ergebnisse stimmen überein, was den Erfolg dieses Tests bestätigt.



(a) Ergebnisse der Anfrage in der Implementierung (b) Ergebnisse der Anfrage im Browser

Abbildung 5.3.: Verifikation der Ergebnisse der Pattern-Query aus Listing 5.3

5.4.2. Edge-Query

Eine weitere Art von Anfrage stellen die Edge-Queries dar. Diese wählen alle Kanten eines Typs mit optionalen Selektionsprädikaten aus. Das Testanfrage dient hier Listing 5.5, welche alle Kanten des Typs `fieldtrip` zurückgibt. Damit wird nach allen Kanten gesucht, welche eine Reise Wossidlos, die im Rahmen seiner Befragungen durchgeführt wurde, beschreiben. Manthey nennt in [23] als Grund für die Auswahl dieser Kante die sehr kleine Ergebnismenge der Anfrage, welche leicht verifiziert werden kann.

```
FROM g
EDGES fieldtrip;
```

Listing 5.5: Beispiel Edge-Query HyperGQL

Die Anfrage wird zunächst in die in Listing 5.6 übersetzt. Anschließend folgt eine in Abschnitt 5.3 beschriebene Auswertung des `Request`-Objekts. Ähnlich wie bei den Node-Queries aus dem vorherigen Abschnitt ist auch hier zu sehen, dass eine Edge-Query problemlos in einen RHGA-Ausdruck umgeformt werden kann. Dazu wird als gesuchter Typ der Kantentyp angegeben. Anschließend wird auf alle Attribute des gesuchten Elements projiziert. Da Hyperkanten in PowerGraph keine Attribute besitzen, werden als Ergebnisse allein die IDs der gefundenen Kanten zurückgegeben.

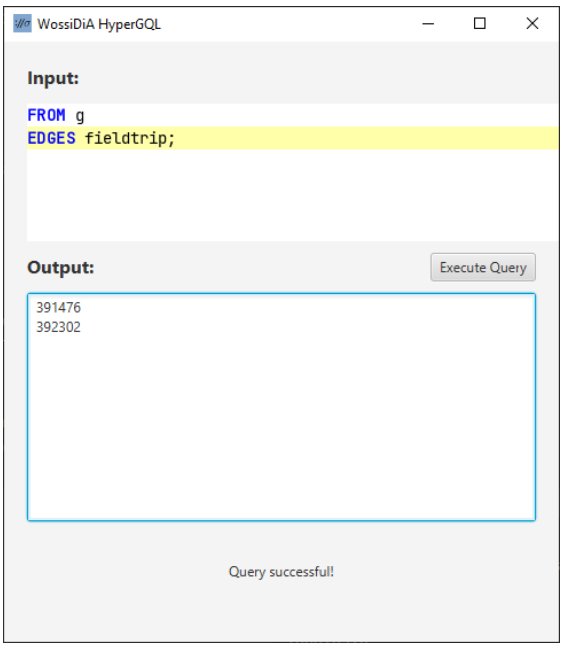
```
GraphVariable edge = new GraphVariable("@0",
        new GraphType(GraphType.ElementType.EDGE, "fieldtrip"));

Response response = Request(catalog)
        .select(new TypePredicate(edge))
        .project(new ProjectionOperator("*"))
        .evaluate();
```

Listing 5.6: Beispiel Edge-Query Java

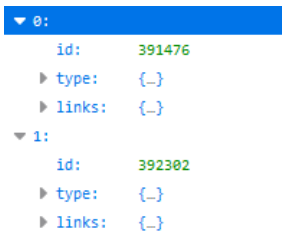
Die Ergebnisse der Anfrage sind in Abbildung 5.4 dargestellt. Dabei zeigt die linke Abbildung wieder das Ergebnis im Prototypen, während die rechte Abbildung das Ergebnis der Anfrage `http://api.wossidia.de/edges/fieldtrip` im Browser darstellt. Zur besseren Übersicht sind auch hier wieder die weiteren Eigenschaften der Kanten minimiert. Die Ergebnismenge ist bei beiden Ausführungen gleich, was einen Erfolg des Tests bedeutet.

5.4.3. Pattern-Query

Die dritte Art von Anfragen stellen die Pattern-Queries dar. Durch diese lassen sich nach Knoten suchen, welche durch eine spezifische Kante miteinander verbunden sind. Als Beispiel dient die Anfrage aus Listing 5.7. Diese enthält ein **`MATCH`**-Statement, welches ein Pattern beinhaltet. Dieses beschreibt die beiden Knoten `a:am_place` und `w:18`, welche über die Hyperkante `content` miteinander verbunden sind. Die Identifikatoren vor den Doppelpunkten



(a) Ergebnisse der Anfrage in der Implementierung



(b) Ergebnis der Anfrage im Browser

Abbildung 5.4.: Verifikation der Ergebnisse der Pattern-Query aus Listing 5.5

der Knoten stellen dabei Variablen dar, in welchen die Ergebnisse der Anfrage gespeichert werden. Diese lassen sich in einem **RETURN**-Statement auf einzelne Attribute einschränken, womit **RETURN**-Klauseln das Äquivalent zu Projektionen aus der Datenbanktechnik darstellen. Des Weiteren definiert die Anfrage mehrere **WHERE**-Ausdrücke, welche die Knoten auf spezifische Attributwerte selektieren.

Zusammenfassend sucht die Anfrage nach allen Ortsknoten mit dem Namen „Zapel“, welche sich in der Nähe von „Crivitz“ befinden. Dabei müssen die Knoten über eine Hyperkante vom Typ „Inhalt“ mit Wortknoten verbunden sein. Schließlich die Worte der von allen gefundenen Wortknoten zurückgegeben werden.

```
FROM g
MATCH (a:am_place) -[:content]- (w.18)
WHERE a.name = 'Zapel'
    AND a.near = 'Crivitz'
RETURN w.word;
```

Listing 5.7: Beispiel Pattern-Query HyperGQL

Die Umformung in ein `Request`-Objekt ist in Listing dargestellt. Hierbei ist die Funktion `Request::join()` zu beachten, welche die Knoten- sowie die Kantenvariablen des Patterns entgegennimmt. Die weiteren Operatoren verhalten sich wie in den in den vorherigen Abschnitten erläuterten Tests.

```
GraphVariable a = new GraphVariable("a",
        new GraphType(GraphType.ElementType.NODE, "am_place"));
GraphVariable w = new GraphVariable("w",
        new GraphType(GraphType.ElementType.NODE, 18));
GraphVariable edge = new GraphVariable("@0",
        new GraphType(GraphType.ElementType.EDGE, "content"));

Response response = Request(catalog)
        .join(new JoinOperator(a, w, edge))
        .select(new TypePredicate(a))
        .select(new TypePredicate(c))
        .select(new OperatorValuePredicate(a, "name", "Zapel", OperatorType.EQUAL))
        .select(new OperatorValuePredicate(a, "near", "Crivitz", OperatorType.EQUAL))
        .project(new ProjectionOperator("w", "word"))
        .evaluate();
```

Listing 5.8: Beispiel Pattern-Query Java

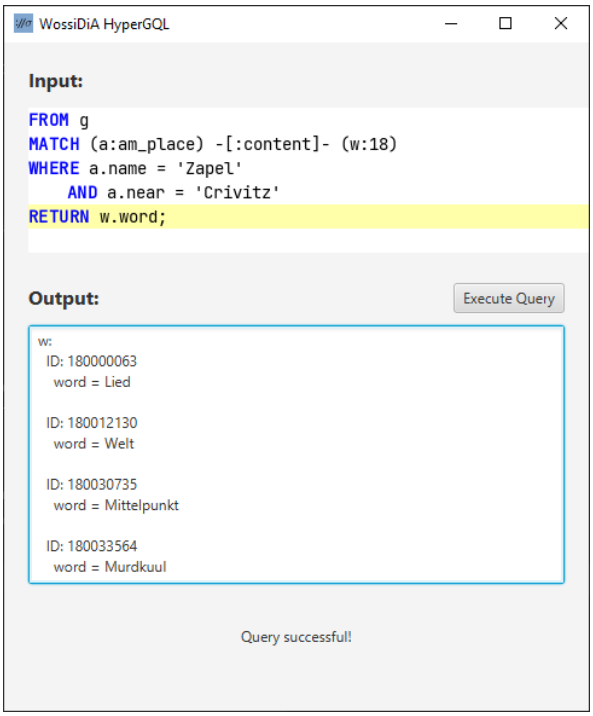


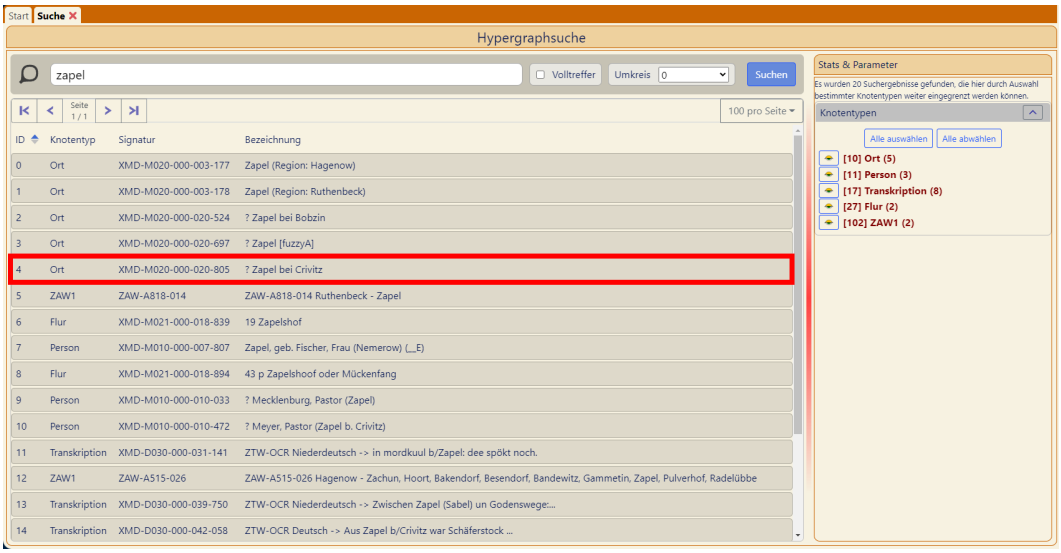
Abbildung 5.5.: Pattern-Query in der Anwendung

Das Ergebnis der Anfrage ist in Abbildung 5.5 dargestellt. Im Gegensatz zu den vorherigen Tests wurde nun nach einer bestimmten Variable, `w`, gefiltert. Diese wird den Ergebnissen vorangestellt. Bei Anfragen mit mehreren verschiedenen Variablen ist so eine Zuordnung der einzelnen Werte zu ihren Variablen möglich. Weiterhin wurde nach einem bestimmten Attribut gesucht, welches für jeden Knoten gelistet ist.

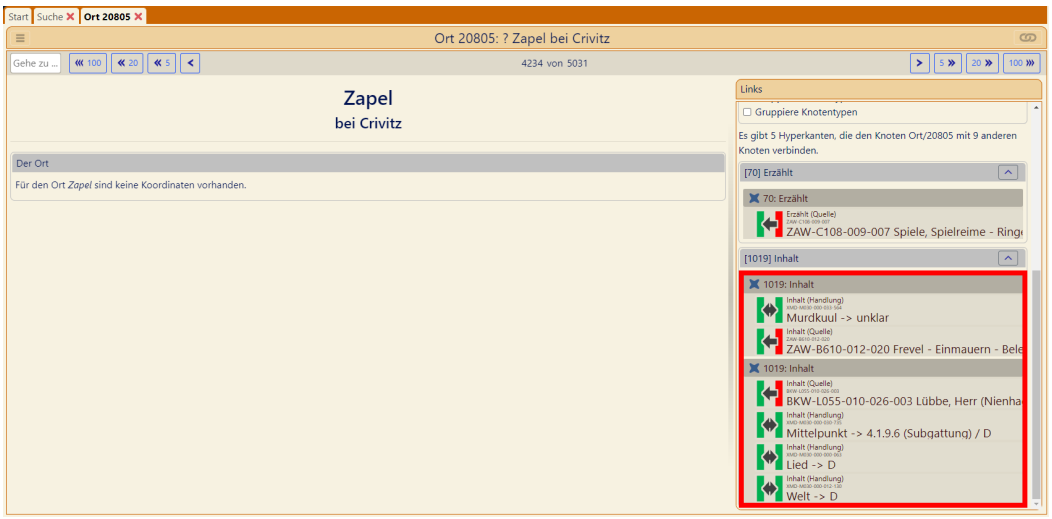
Zur Überprüfung der Ergebnisse Pattern-Query-Tests eignet sich die Weboberfläche des WossiDiA-Projektes, welche unter `http://apps.wossidia.de` erreichbar ist. Hierbei wird im Suchfeld der Webanwendung nach dem Begriff „Zapel“ gesucht. Diese Anfrage listet jeden im Hypergraphen gespeicherten Knoten, in welchem „Zapel“ als Teil des Bezeichners vorkommt. Der relevante Knoten der Suchanfrage ist in Abbildung 5.6a markiert. Bei Aus-

wahl dieses Knotens lassen sich alle Verbindungen zu anderen Knoten über die jeweiligen Kantentypen anzeigen. Dabei sind nur bidirektionale Kanten relevant, da HyperGQL keine Auswahl an gerichteten Kanten unterstützt. Die verbundenen Knoten werden in Abbildung 5.6b dargestellt. Dabei sind die bidirektionalen Knoten an ihren grünen Symbolen zu erkennen.

Es lässt sich feststellen, dass die Ausgabeknoten der Weboberfläche den Ausgabeknoten des Prototyps gleichen. Dies ist daran zu erkennen, dass die Werte für das Attribut „Wort“ bei beiden Knoten gleich sind. Damit ist auch dieser Test erfolgreich.



(a) Ergebnis der Suche nach „Zapel“ in der WossiDiA-Weboberfläche. Der relevante Knoten ist rot markiert.



(b) Anzeige der Ergebnisknoten der Anfrage (rot markiert). Die relevanten Knoten sind bidirektional, zu erkennen an den grünen Symbolen.

Abbildung 5.6.: Verifikation der Ergebnisse der Pattern-Query aus Listing 5.7

5.5. API-Verbesserungsvorschläge

Durch die intensive Auseinandersetzung mit der REST-API des WossiDiA-Projektes sind im Verlauf der Implementierung einige Schwachstellen und mögliche Verbesserungen der Schnittstelle aufgefallen, welche im nachfolgenden Abschnitt genauer beschrieben werden.

Dokumentation Die Schnittstelle für manche Endpunkte momentan nur äußerst spärlich dokumentiert. So ist für die Endpunkte aus der Dokumentation heraus nicht ersichtlich, wie

```
/nodes/{type}/{cond}
/edges/{type/{cond}
```

die Filterkondition zu gestalten ist. Hiermit werden diese Endpunkte für Anwender ohne Wissen um die internen Abläufe der PowerGraph-Schnittstelle praktisch nutzlos. Gleiches gilt für den Endpunkt

```
/graph/{pnid}/{path}
```

mit welchem sich ein Subgraph des Hypergraphen ausgeben lässt. Allerdings ist der Dokumentation nicht zu entnehmen, wie der Pfadausdruck `{path}` auszusehen hat. Eine ausführlichere Dokumentation dieser Endpunkte, vielleicht auch mit Beispielanfragen, wäre hier also äußerst wertvoll.

HTTP-Statuscodes Ein Verbesserungswunsch, welcher im Verlauf der Implementation bereits umgesetzt wurde, ist die Rückgabe korrekter HTTP-Statuscodes für API-Anfragen. So wurden zu Beginn der Arbeit für jede nicht korrekte Anfrage der Statuscode 500 zurückgegeben, was einen internen Fehler des Servers signalisiert. Allerdings war es auch möglich, Anfragen an die Datenbank zu stellen, welche eine leere Knoten- oder Kantenmenge zurücklieferte. Auch bei solchen Anfragen wurde ein interner Serverfehler signalisiert. Nach der korrekten Implementation von HTTP-Statuscodes wird nun bei leeren Antworten der Statuscode 404 zurückgegeben, was ein Fehlen der angefragten Ressource signalisiert.

Neue API-Endpunkte Mit dem aktuellen Stand der API ist es nur sehr umständlich möglich, alle Knoten und Kanten der Datenbank gleichzeitig anfragen. Theoretisch besteht zwar die Möglichkeit, zunächst via `/catalog` alle Knoten- und Kantentypen des Graphen herunterzuladen und danach für jeden Typen eine eigene Anfrage via `/nodes/{type}` und `/edges/{type}` zu stellen. Allerdings bedeutet dies in der Praxis einen enormen Mehraufwand, da hierduch tausende einzelne HTTP-Anfragen gestellt werden müssen. Diese Arbeit schlägt daher die neuen Endpunkte

```
/nodes
/edges
```

vor, welche alle Knoten- beziehungsweise Kanten der Datenbank gleichzeitig in einer einzelnen HTTP-Anfrage anfragen. Bei größeren Datenbeständen könnte solch eine Anfrage allerdings zu sehr großen Antworten führen.

Filteranfragen von Elementen ohne Typ Aktuell ist es mit der API nicht möglich, nach Knoten filtern, ohne vorher ihren Typen zu kennen. Werden zum Beispiel alle Knoten mit dem Attributpaar `name = Crivitz` gesucht, muss vorher der mögliche Typ für diese Knoten ermittelt werden. Erst dann kann mit der Anfrage `/nodes/{type}/@name='Crivitz'` die gewünschte Knotenmenge gefunden werden. Dies macht eine solche Filteranfrage umständlich. Eine mögliche Ergänzung zur aktuellen Schnittstelle stellen somit die Endpunkte

```
/nodes/{cond}
/edges/{cond}
```

dar, welche alle Knoten- und Kanten mit dem gewünschten Attributpaar ermitteln.

Alle Attributwerte eines Typen Für das Filterkriterium $pos_i.p = pos_j.q$ in der Hypergraphenalgebra aus Definition 4.8 ist es sinnvoll, alle Attributwerte eines Attributs für einen bestimmten Knotentyp anfragen zu können. Somit können leichter Vergleiche zwischen verschiedenen Knotentypen gestellt werden. Hierfür müssten die Filter in den bereits bestehenden Endpunkten

```
/nodes/{type}/{cond}
/edges/{type}/{cond}
```

erweitert werden, beispielsweise durch `{cond}` := `attr = 'attrName'`, ähnlich wie bei einer Anfrage einer ID. Die Ausgabe des Endpunktes würde dann alle Attributpaare der Knoten des angegeben Typs darstellen, eventuell verknüpft mit seiner ID.

Übertragen auf das vorherige Beispiel könnte eine solche Anfrage wie folgt aussehen:

```
/nodes/10/attr='name'
```

Quellknotenmengen in Graph-Anfragen Der in der Implementierung dieser Arbeit nicht genutzte API-Endpunkt `/graph/` unterstützt momentan nur einen Quellknoten pro Anfrage. Wenn alle Knoten gefunden werden sollen, die über eine Hyperkante miteinander verbunden sind, muss daher zunächst für jeden Quellknoten eine eigene Anfrage über `/nodes/{type}/{cond}` gestellt werden, worauf folgend für jeden Knoten aus der Antwort dieser Anfrage eine neue `/graph/{pnid}/{path}`-Anfrage gestellt werden muss. Diese beiden

Anfragetypen könnten zu einer neuen Anfrage vereinigt werden, in der `/graph/` auch die Filterung von Quell- und Zielknoten nach Typen und Attributen unterstützt, beispielsweise

```
/graph/{cond}/{path}/{cond}
```

`{cond}` entspricht hier den Filterkonditionen von `/nodes/{type}/{cond}` mit der zusätzlichen Option zur Filterung von Typen, beispielsweise mit `type={type}`.

Tabelle 5.1 zeigt eine Zusammenfassung der vorgeschlagenen Ergänzungen und Verbesserung im Stile der offiziellen WossiDiA-REST-API-Dokumentation.

Returns	**Path**	**Description**	**Path Parameters**
`application/json`	`/nodes`	Returns all WossiDiA-PowerGraph nodes.	
`application/json`	`/edges`	Returns all WossiDiA-PowerGraph edges.	
`application/json`	`/nodes/{cond}`	Obtains nodes with the specified conditions.	`cond`
`application/json`	`/edges/{cond}`	Obtains edges with the specified conditions.	`cond`
`application/json`	`/nodes/{type}/{cond}`	Obtains nodes of the specified type with the specified conditions, now also includes attr.	`type` `cond`
`application/json`	`/graph/{cond}/{path}/{cond}`	Obtains a graph, starting from a list of nodes that satisfy the conditions specified in the `cond` statements and returns a list of nodes that share a hyperedge specified by `edge`.	`cond` `path`

Tabelle 5.2.: Vorschlag zur Erweiterung der REST-API im Stile der bestehenden Dokumentation

6. Zusammenfassung und Ausblick

Dieses Kapitel stellt den inhaltlichen Abschluss dieser Bachelorarbeit der und beinhaltet eine Zusammenfassung und Reflektion der erarbeiteten Ergebnisse sowie eine Betrachtung zukünftiger möglicher Verbesserungen und Weiterentwicklungen der vorgestellten Algebra sowie der begleitenden Implementation.

6.1. Zusammenfassung

Als Teil des WossiDiA-Projektes entstand die Hypergraphendatenbank Hydra.PowerGraph, welche die weltweit erste Implementierung eines Hypergraphendatenbankmodells darstellt. Für diese Datenbank existiert noch keine deskriptive Anfragesprache vergleichbar mit dem relationalen Datenbankäquivalent SQL. Darüber hinaus ist auch noch keine theoretische Grundlage für eine solche Sprache in Form einer Algebra bekannt.

Der erste Schritt dieser Bachelorarbeit bestand deshalb daraus, bestehende Ansätze und Beschreibungen von Graphenalgebren anderer Graphdatenbankmodelle zu recherchieren und zu evaluieren. Teil dieser Aufgabe war es auch, ihre Tauglichkeit zur Beschreibung des neuen Graphanfragesprachstandards GQL zu untersuchen. Da sich dieser Standard zum Zeitpunkt dieser Arbeit noch in Entwicklung befindet, wurde auf den vorliegenden Vorschlag zur Formulierung der Sprache [24] aus A.2 zurückgegriffen. Die vorliegende Literatur zur Untersuchung von Graphanfragesprachen durch Graphenalgebren ist nicht sehr umfangreich, was dem noch jungen Alter solcher Datenbankmodelle geschuldet ist. Dennoch konnte ein Ansatz [1] gefunden und beschrieben werden. Dieser Schritt kann also als Erfolg angesehen werden.

Das zweite Ziel der Arbeit war es, die evaluierten Graphenalgebren aus Schritt eins um eine angepasste Version an das Hydra.PowerGraph-System zu erweitern und mit der angepassten Version von GQL, HyperGQL [23] in Einklang zu bringen. Dabei war es Ziel der Arbeit, sich auf den reinen Anfrageteil einer solchen Algebra zu beschränken - andere Operatoren wie etwa das Einfügen und Modifizieren neuer oder bereits bestehender Knoten oder Kanten wurden bewusst nicht betrachtet. Die aus dieser Arbeit entstandene Graphenalgebra ist in Abschnitt 4.3 beschrieben, ihre Verknüpfung zu HyperGQL lässt sich in Abschnitt 4.4 nachvollziehen.

Drittes Ziel dieser Bachelorarbeit war schließlich die prototypische Implementierung der beschriebenen Graphenalgebra sowie der Anfragesprache HyperGQL. Hierzu wurde die vom WossiDiA-Projekt öffentlich bereitgestellte REST-Schnittstelle sowie eine modifizierte Version der HyperGQL-ANTLR-Grammatik aus [23] verwendet. Die Implementation nimmt als Eingabe HyperGQL-Anfragen entgegen, welche mithilfe des ANTLR-Sprachübersetzers in gültige Hypergraphenalgebra-Ausdrücke umgeformt werden. Anschließend werden diese Ausdrücke in URL-Anfragen umgeformt und an die REST-API gestellt. Schließlich werden die Antworten der Schnittstelle ausgewertet, aufbereitet und dem Benutzer zurückgegeben. Es ist dabei möglich, die Übersetzung von HyperGQL in RHGA zu umgehen und direkt programmatisch Anfragen zu formulieren. Durch die teilweise fehlende oder nur sehr sparsame Dokumentation von einzelnen Endpunkten der WossiDiA-API konnten einzelne Anfrageteile nur über ineffiziente Umwege implementiert werden, was die Auswertung von Anfragen deutlich verlangsamt.

Zum Testen ausgewählter Anfragen wurde die Web-Suchoberfläche des WossiDiA-Projektes verwendet. Hierbei muss angemerkt werden, dass es sich bei diesen Tests um Stichproben handelt, welche keinen Anspruch auf vollständige Korrektheit der Implementation erheben. Vielmehr sollte mit diesen Tests eine allgemeine Funktionstüchtigkeit der Implementierung durch einfache Anfragen demonstriert werden.

Abschließend wurden Verbesserungen der REST-API vorgeschlagen, welche sich allerdings teilweise mit undokumentierten aber bereits bestehenden Funktionen der API überschneiden.

6.2. Ausblick

Nach der Zusammenfassung der Ergebnisse der Arbeit im vorherigen Abschnitt dieses Kapitels folgt nun eine Reihe von möglichen Verbesserungs- und Weiterentwicklungvorschlägen sowohl der Graphenalgebra-Definition als auch der Implementation.

Die offensichtlichste und zugleich wichtigste Erweiterung der Graphenalgebra stellt die Einführung von Attributsprojektionen dar, welche die Auswahl einzelner Attribute einer Knoten- und Kantenmenge erlauben. Dies könnte mithilfe neuer Operatoren, beispielsweise der Form $\Pi(\text{pos}) = \{p_1, \ldots, p_n\},\ p \in \mathcal{K}$ geschehen, die eine Menge von Knoten und Kanten weiter eingrenzen.

Der bisherige Algebra-Entwurf beinhaltet keine Operationen zur Erstellung und Modifikation von Knoten und Kanten. Weiterentwicklungen könnten demnach neue Operatoren zur Unterstützung dieser Anfragen beinhalten. Dies könnte wiederum Konsequenzen in anderen Bereichen nach sich ziehen. Bonifati Et Al. [1] unterstützen Projektionen beispielsweise mit der Erstellung neuer (temporärer) Knoten und Kanten.

Weiterhin bietet es sich an, die Implementierung durch Abbildung weiterer RHGA-Ausdrücke in HyperGQL zu erweitern. So sind zum Beispiel noch keine Verkettungen von Anfragen zulässig, was durch den GQL-Standard allerdings definiert in RHGA grundsätzlich möglich ist. Auch können noch keine anderen Einschränkungen als Typ-Einschränkungen an Hyperkanten zu treffen. Für eine Auflistung anderer möglicher Erweiterungen sei auf Manthey [23] verwiesen.

In Algebren existieren häufig Äquivalente von bestimmten Ausdrücken. So kann eine Aneinanderreihung bestimmter Operatoren mit bestimmten Parametern wiederum einen anderen Operator ergeben. Dies ist für die Optimierung von Anfragen von Interesse, bei der eine möglichst geringe Anzahl an Operationen erwünscht ist. Es wäre daher interessant, solche möglichen Äquivalenzen innerhalb der Algebra zu finden und zu identifizieren.

Außer Hydra.PowerGraph sind noch keine weiteren Hypergraphendatenbankmodelle bekannt. Allerdings kann davon ausgegangen werden, dass in Zukunft auch andere Hypergraphendatenbanken konzipiert und implementiert werden. Diese werden zwar das allgemeine Konzept von Hypergraphen miteinander teilen, sich aber vermutlich auch in einzelnen Details des zugrunde liegenden Modells unterscheiden. So sind beispielsweise auch Hypergraphen mit Hyperknoten denkbar. Sobald sich eine genügende Anzahl anderer Hypergraphendatenbankmodelle etabliert hat, könnte eine erneute Evaluation der in dieser Arbeit vorgestellten Algebra vorgenommen werden. Dadurch könnte eine angepasste oder neue Algebra auch Basis dieser Version entwickelt werden, welche die Konzepte anderer Datenbankmodelle beinhaltet und berücksichtigt.

7. Leitfaden für die digitale Version

Dieses Kapitel verschafft einen kurzen Überblick über die digitalen Dateien, welche die Ergebnisse dieser Arbeit darstellen.

Ordnerstruktur & -inhalte Die gesamte Bachelorarbeit befindet sich in einem ZIP-Archiv mit dem Namen „`ba_matthias_losch_215205549.zip`“. Dieses enthält als grundlegende Struktur die folgenden Ordner:

- Dokument
- Implementierung
- Literatur

Der Ordner „Dokument“ beinhaltet sowohl die LaTeX-Dateien, welche zur Erstellung der Bachelorarbeit verwendet wurden, als auch alle Abbildungen, welche in das Dokument eingebunden wurden. Die TeXdateien sind dabei im Unterordner „Kapitel“ zu finden, während die Abbildungen im Unterordner „Abbildungen“ abgelegt sind. Als Einstiegspunkt dient die Datei „`main.tex`“. Das Dokument kann mit einer einfachen Ausführung der Datei „`build.bat`“ beziehungsweise „`build.sh`“ erstellt werden. Hierfür wird eine aktuelle Version des Zeichensetzungsprogramms LuaLaTeX sowie der Literaturverwaltungsprogramme Biblatex und Biber benötigt.
Der Ordner „Implementierung“ beinhaltet die Quelltext-Dateien der dieser Arbeit begleitenden Implementierung kompilierte und ausführbare Versionen für Windows und Linux. Die Quelltexte befinden sich dabei im Verzeichnis „hypergql“, welches als Projektverzeichnis der Implementation dient. Das Projekt lässt sich einfach über das Build-Management-Automatisierungswerkzeug Gradle kompilieren und ausführen. Hierzu werden die Skripte „`gradlew.bat`“ beziehungsweise „`gradlew`“ ausgeführt. Die vorkompilierten Versionen des Programms lassen sich in den Verzeichnissen „`HyperGQL_{OS}`“ finden, wobei „{OS}“ für das jeweilige Betriebssystem steht. Darüber hinaus befindet sich eine Dokumentation des Quelltexts in Form von Javadoc-HTML-Dateien im Unterordner „Dokumentation“.
Im Order „Literatur“ befinden sich alle schriftlichen Quellen in Form von PDF-Dateien, welche im Rahmen dieser Bachelorarbeit ausgewertet und zitiert wurden. Die Dateinamen der Quellen folgt dabei dem Schema `{Autor}{EtAl}{Jahr}{Buchstabe}.pdf`, wobei bei „EtAl“ bei mehreren Autoren und „Buchstabe“ bei mehreren Veröffentlichung desselben Autors innerhalb eines Jahres genutzt wird.

Literatur

[1] Angela Bonifati u. a. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018. DOI: 10.1007/978-3-031-01864-0.

[2] Todd J. Green u. a. *Datalog and Recursive Query Processing*. Now Foundations und Trends, 2013. ISBN: 978-1-601-98752-5. DOI: 10.1561/1900000017.

[3] Juan L. Reutter und Moshe Y. Vardi. „Regular Queries on Graph Databases“. In: Theory of Computing Systems 61 (Juli 2017), S. 31–83. DOI: 10.1007/s00224-016-9676-2.

[4] Gábor Szárnyas u. a. „Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance“. In: *CoRR* abs/1806.07344 (Aug. 2018). DOI: 10.48550/arXiv.1806.07344.

[5] *Information technology — Database languages — SQL*. eng. Standard ISO/IEC 9075-1:2016. Genf, CH: International Organization for Standardization, Dez. 2016. URL: https://www.iso.org/standard/63555.html.

[6] *Information technology — Database languages — GQL*. eng. Standard ISO/IEC CD 39075.2. Genf, CH: International Organization for Standardization, 2022. URL: https://www.iso.org/standard/76120.html.

[7] Alf-Christian Schering u. a. „From Box to Bin – Semi-automatic Digitization of a Huge Collection of Ethnological Documents“. In: *Digital Libraries: For Cultural Heritage, Knowledge Dissemination, and Future Creation*. Bd. 7008. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Okt. 2011, S. 168–171. ISBN: 978-3-642-24826-9. DOI: 10.1007/978-3-642-24826-9_22.

[8] Reinhard Diestel. *Graphentheorie*. 5. Aufl. Springer, 2017. ISBN: 978-3-662-53633-9.

[9] Giorgio Gallo u. a. „Directed hypergraphs and applications“. In: *Discrete Applied Mathematics* 42.2 (1993), S. 177–201. ISSN: 0166-218X. DOI: 10.1016/0166-218X(93)90045-P.

[10] Keqin Feng und Wen-Ch'ing (Winnie) Li. „Spectra of Hypergraphs and Applications“. In: *Journal of Number Theory* 60.1 (1996), S. 1–22. ISSN: 0022-324X. DOI: 10.1006/jnth.1996.0109.

[11] Giorgio Ausiello, Alessandro D'Atri und Domenico Saccà. „Minimal Representation of Directed Hypergraphs“. In: *SIAM Journal on Computing* 15.2 (1986), S. 418–431. DOI: 10.1137/0215029.

[12] Giorgio Ausiello. „Directed Hypergraphs: Data structures and applications“. In: *CAAP '88, 13th Colloquium on Trees in Algebra and Programming, Nancy, France, March 21-24, 1988*. Bd. 299. Lecture Notes in Computer Science. Springer, S. 295–303. ISBN: 978-3-540-38930-9. DOI: 10.1007/BFb0026111.

[13] Giorgio Ausiello und Luigi Laura. „Directed Hypergraphs: Introduction and fundamental algorithms - A survey“. In: *Theoretical Computer Science* 658 (2017), S. 293–306. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2016.03.016.

[14] Renzo Angles. „The Property Graph Database Model“. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*. Bd. 2100. CEUR-WS.org, Mai 2018.

[15] Justin J. Miller. „Graph Database Applications and Concepts with Neo4j“. In: *SAIS 2013 Proceedings*. Bd. 24. 2013.

[16] Jaroslav Pokorný. „Conceptual and Database Modelling of Graph Databases“. In: *Proceedings of the 20th International Database Engineering & Applications Symposium*. IDEAS '16. Association for Computing Machinery, 2016, S. 370–377. DOI: 10.1145/2938503.2938547.

[17] Dominik Tomaszuk u. a. „Serialization for Property Graphs“. In: *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*. Springer, 2019, S. 57–69. ISBN: 978-3-030-19093-4. DOI: 10.1007/978-3-030-19093-4_5.

[18] Holger Meyer, Alf-Christian Schering und Andreas Heuer. „The Hydra.PowerGraph System“. In: *Datenbank Spektrum* 17.2 (Juli 2017), S. 113–129. DOI: 10.1007/s13222-017-0253-x.

[19] Nadime Francis u. a. *Formal Semantics of the Language Cypher*. Techn. Ber. März 2018. DOI: 10.48550/arXiv.1802.09984.

[20] Nadime Francis u. a. „Cypher: An Evolving Query Language for Property Graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ' 18. Association for Computing Machinery, Juli 2018, S. 1433–1445. DOI: 10.1145/3183713.3190657.

[21] Oskar van Rest u. a. „PGQL: A Property Graph Query Language“. In: GRADES '16. Association for Computing Machinery, Juni 2016, S. 1–6. ISBN: 978-145-03478-08. DOI: 10.1145/2960414.2960421.

[22] Renzo Angles u. a. „G-CORE: A Core for Future Graph Query Languages“. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Association for Computing Machinery, Juni 2018, S. 1421–1432. ISBN: 978-145-03470-37. DOI: 10.1145/3183713.3190654.

[23] Erik Manthey. „Querying Hypergraphs: Entwurf einer deskriptiven Anfragesprache“. Masterarbeit. Universität Rostock, Institut für Informatik, Aug. 2021.

[24] Neo4j Query Languages Standards and Research Team. *GQL Scope and Features*. Techn. Ber. 3. Neo4j Inc., Dez. 2018.

[25] Edgar F. Codd. „A Relational Model of Data for Large Shared Data Banks“. In: *Communications of the ACM* 13.6 (Juni 1970), S. 377–387. DOI: 10.1145/362384.362685.

[26] Pierre Bourhis, Markus Krötzsch und Sebastian Rudolph. „How to Best Nest Regular Path Queries“. In: *Informal Proceedings of the 27th International Workshop on Description Logics*. Juli 2014.

[27] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II*. Computer Science Press, Inc., 1988.

[28] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken - Implementierungstechniken*. 4. Aufl. mitp, 2019. ISBN: 978-3-958-45779-7.

[29] Roy Thomas Fieldings. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. University of California, Irvine, 2000. ISBN: 978-0-599-87118-2.

[30] ANTLR/Terence Parr. *ANTLR*. 2022. URL: https://www.antlr.org (besucht am 07. 11. 2022).

[31] Google. *GSON: A Java serialization/deserialization library to convert Java Objects into JSON and back*. 2022. URL: https://github.com/google/gson (besucht am 07. 11. 2022).

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

A. Anhang

⟨request⟩ := ⟨query⟩ | ⟨functioncall⟩ ;
⟨query⟩ := ⟨patternquery⟩ | ⟨nodeedgequery⟩ ;
⟨patternquery⟩ := **FROM**, ⟨graph⟩, ⟨matchstatementlist⟩, ⟨returnstatement⟩ ;
⟨nodeedgequery⟩ := **FROM**, ⟨graph⟩, ⟨nodes⟩ | ⟨edges⟩, ⟨type⟩, [⟨conditionlist⟩] ;
⟨functioncall⟩ := **CALL**, ⟨functionname⟩, ⟨paramlist⟩ ;
⟨graph⟩ := (‘**(**’, ⟨request⟩, ‘**)**’) | ⟨identifier⟩ ;
⟨matchstatementlist⟩ := ⟨matchstatement⟩, connector, matchstatement ;
⟨matchstatement⟩ := **MATCH**, ⟨pattern⟩, ‘**,**’, ⟨pattern⟩, [wherestatementlist] ;
⟨wherestatementlist⟩ := **WHERE**, wherestatement, connector, wherestatement ;
⟨wherestatement⟩ := [**NOT**], ⟨variable⟩, [‘**.**’, ⟨attribute⟩], ⟨operator⟩, ⟨value⟩ ;
⟨returnstatement⟩ := **RETURN** ⟨projection⟩ ;
⟨pattern⟩ := ⟨node⟩, ⟨edge⟩, ⟨node⟩ ;
⟨functioname⟩ := ⟨identifier⟩ ;
⟨paramlist⟩ := ‘**(**’, [⟨paramlist⟩, ‘**,**’, param], ‘**)**’ ;
⟨param⟩ := ⟨identifier⟩ ;
⟨conditionlist⟩ := condition, connector, condition ;

Listing A.1: Grammatik von HyperGQL in EBNF, übernommen aus [23]

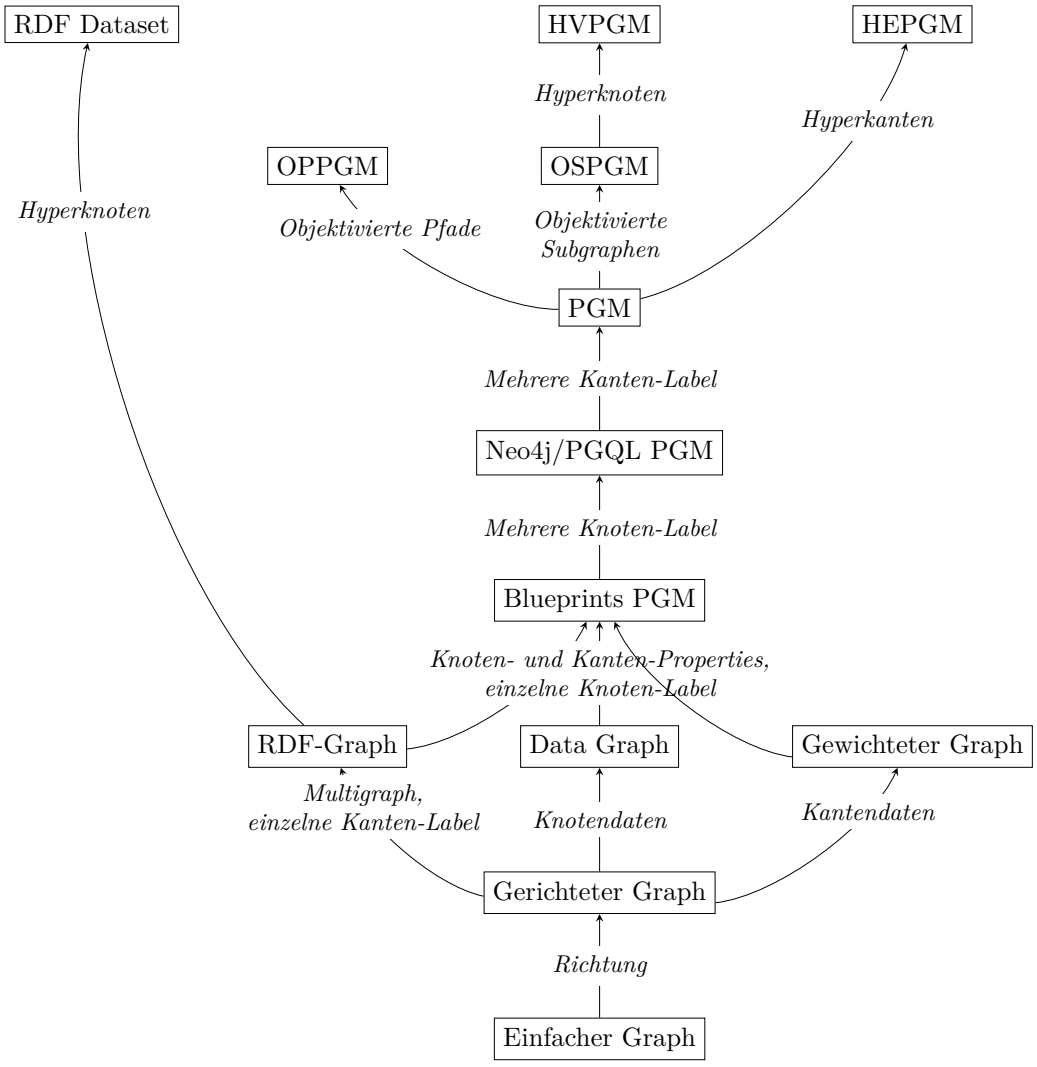


Abbildung A.1.: Hierarchie von PGM-Varianten und -Erweiterungen (übernommen aus [1]). PowerGraph stellt dabei eine Form von HEPGM dar.

⟨request⟩ := [⟨preamble⟩], (⟨procedure⟩ | ⟨catalog procedure⟩) ;
⟨procedure⟩ := ⟨local declarations⟩, ⟨procedure body⟩ ;
⟨local declarations⟩ := ⟨local declaration⟩ ;
⟨procedure body⟩ := ⟨composite statement⟩, ⟨combinator⟩, ⟨composite statement⟩ ;
⟨composite statement⟩ := ⟨statement list⟩ | ⟨nested procedure⟩ ;
⟨statement list⟩ := ⟨statement⟩ - ;
⟨nested procedure⟩ := '{', ⟨procedure⟩, '}' ;
⟨local declaration⟩ := **QUERY**, ⟨identifier⟩, [**AS**, '{', ⟨procedure⟩, '}']
| **PATH**, ⟨identifier⟩, **AS**, ⟨pattern⟩
| **GRAPH**, ⟨identifier⟩, [**AS**, '{', ⟨procedure⟩, '}']
| **PARAM**, ⟨identifier⟩, **AS**, ⟨expression⟩
| ... ;
⟨statement⟩ := [**FROM** ⟨identifier⟩ | ⟨named procedure call⟩],
[**OPTIONAL**|**MANDATORY**] **MATCH**, ⟨pattern⟩, [⟨**WHERE**⟩]
| **CALL**, ⟨call arguments⟩, [**YIELD**, ⟨non-empty variable list⟩]
| **OPTIONAL**, ⟨call arguments⟩
| **MANDATORY**, ⟨call arguments⟩
| **WITH**, ⟨projection arguments⟩
| **INSERT**, ⟨pattern⟩
| **SET**, ...
| **REMOVE**, ...
| [**DETACH**] **DELETE**, ⟨identifier⟩ -
| **TRUNCATE**, ⟨identifier⟩
| **RETURN**, ⟨projection arguments⟩
| ... ;
⟨where⟩ := **WHERE**, ⟨predicate⟩ ;
⟨call arguments⟩ := ⟨nested procedure⟩
| ⟨named procedure call⟩
| ⟨table name⟩ ;
⟨non-empty variable list⟩ := ⟨identifier⟩ [**AS** ⟨identifier⟩], '**,**', ⟨identifier⟩ [**AS** ⟨identifier⟩] ;
⟨named procedure call⟩ := ⟨identifier⟩, '**(**', [⟨expression⟩ '**,**', ⟨expression⟩], '**)**' ;
⟨projection arguments⟩ := ⟨expression⟨, [**AS**, ⟨identifier⟩] -, [⟨where⟩], [⟨group by⟩],
[⟨order by⟩], [⟨skip⟩], [⟨limit⟩] ;
⟨catalog procedure⟩ := ⟨catalog statement list⟩ ;
⟨catalog statement list⟩ := ⟨catalog statement⟩ - ;
⟨catalog statement⟩ := **CREATE**, **QUERY**, ⟨identifier⟩, [**AS**, '{', ⟨procedure⟩, '}']
| **CREATE**, **PATH**, ⟨identifier⟩, [**AS**, '{', ⟨pattern⟩, '}']
| **CREATE**, **SCHEMA**, ⟨schema⟩
| **CREATE**, **GRAPH**, ⟨identifier⟩, [**AS**, '{', ⟨procedure⟩, '}']
| **ALIAS**, ⟨identifier⟩, **TO**, ⟨identifier⟩
| **DROP**, ⟨identifier⟩
| **RENAME**, ⟨identifier⟩, **TO**, ⟨identifier⟩ ;

Listing A.2: Grammatik von GQL in EBNF, übernommen aus [24]

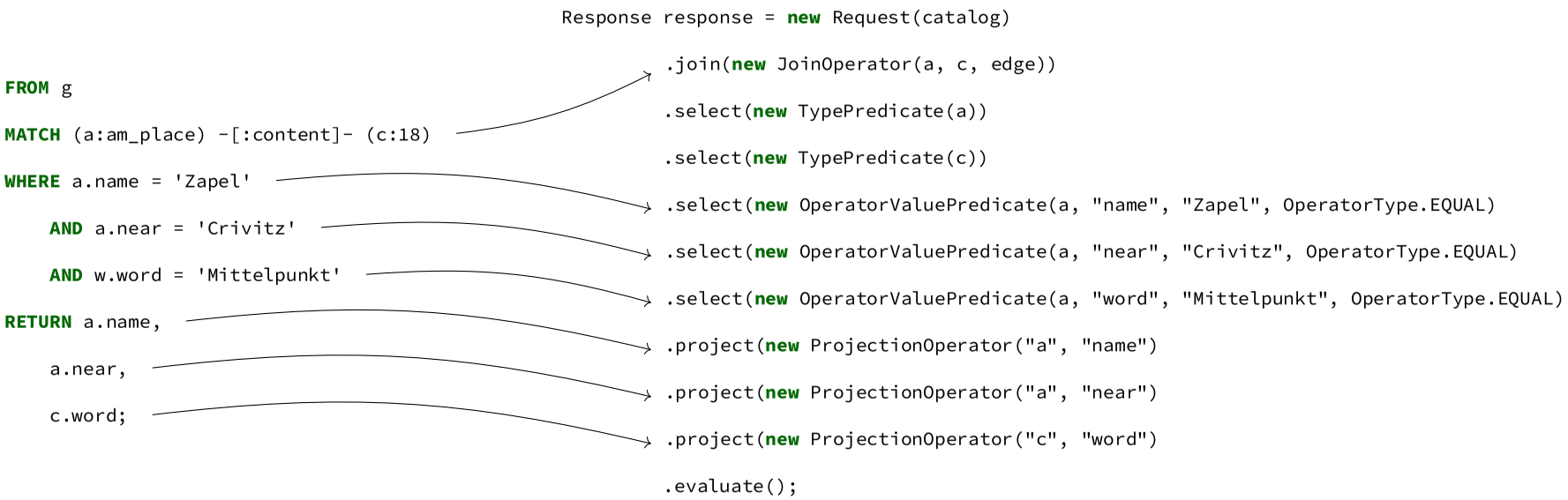


Abbildung A.2.: Beispiel einer Umformung einer HyperGQL-Anfrage in eine RHGA-Anfrage. Die Deklarationen der `GraphVariable`-Objekte für `a` und `c` wurden der Übersichtlichkeit halber weggelassen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Rostock, den 08.11.2022