Universität Regensburg

# Schema-Mapping von Neo4j nach ProSA

Bachelorarbeit im Fach Medieninformatik

am Institut für Information und Medien, Sprache und Kultur (I:IMSK)


| | |
|---|---|
| Vorgelegt von: | Timo Hanöffner |
| E-Mail (Universität): | timo.hanoeffner@stud.uni-regensburg.de |
| Matrikelnummer: | 2296209 |
| Erstgutachter: | Prof. Dr. Niels Henze |
| Zweitgutachter: | Prof. Dr.-Ing. Meike Klettke |
| Betreuer: | Dr.-Ing. Tanja Auge; Dominique Hausler |
| Laufendes Semester: | SS 2025 |
| Abgegeben am: | 24. April 2025 |

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quellcodeverzeichnis

## Zusammenfassung

Mit dem zunehmenden Bedarf an Tools zur Datenanalyse wächst die Relevanz strukturierter Datenverfolgung. Ein solches System ist beispielsweise ProSA. ProSA ermöglicht die Rückverfolgbarkeit von SQL-Anfragen auf relationalen Datenbanken. Für Graphdatenbanken fehlen jedoch vergleichbare Tools. Diese Arbeit stellt ein Framework vor, das Graphdatenbanken aus Neo4j automatisiert in eine relationale Datenbank transformiert. Dabei werden Entitäten, Beziehungsinformationen und Multilabel-Knoten extrahiert und strukturell korrekt in einer PostgreSQL-Datenbank abgebildet, damit eine Weiterverarbeitung in ProSA gewährleistet ist. Das System ist somit in der Lage, ein bestehendes Werkzeug, das für relationale Datenbanken entwickelt wurde, ebenfalls für Graphdatenbanken nutzbar zu machen. Anschließend wird das Framework um das Konzept der Schema-Evolution erweitert und exemplarisch vorgestellt.

## Abstract

With the increasing need for data analysis tools, the relevance of structured data tracking is growing. One such system is ProSA. ProSA enables the traceability of SQL queries on relational databases. However, there is a lack of comparable tools for graph databases. This paper presents a framework that automatically transforms graph databases from Neo4j into a relational database. In the process, entities, relationship information and multi-label nodes are extracted and mapped structurally correctly in a PostgreSQL database so that further processing in ProSA is guaranteed. The system is therefore able to utilise an existing tool that was developed for relational databases for graph databases as well. The framework is then extended to include the concept of schema evolution and presented as an example.

## Aufgabenstellung

Ziel der Bachelorarbeit ist die Implementierung eines Konzeptes zur Anwendung von ProSA auf Graphdatenbanken. Hierfür muss zunächst das Schema der Graphdatenbank extrahiert und dieses in ein relationales Schema überführt werden. Anschließend kann die Datenbank in ProSA ausgewertet und weiterverarbeitet werden und abschließend zurück in eine Graphdatenbank transformiert werden. Zur Überprüfung der korrekten Arbeitsweise von ProSA sollen die Anfrageergebnisse verschiedener SQL-Anfragen vor und nach der Schema-Evolution in ProSA mit denen aus der Graphdatenbank verglichen werden. Schema-Evolution findet über Evolutionsoperatoren wie *add*, *rename*, *delete* und *copy* auf Schemaebene statt.

- Einarbeitung in die Thematik von Graphdatenbanken, Schema-Extraktion und Schema-Mapping
- Erstellung der Beispieldatenbanken in Neo4j und PostgreSQL
- Implementierung von Schema-Extraktion auf der Graph-DB und Schema-Mapping von Neo4j auf PostgreSQL
- Durchführung der Schema-Evolution auf beiden Datenbanken
- Evaluierung des Konzepts

# 1. Einleitung

Relationale Datenbankmanagementsysteme (RDBMS) spielen in modernen Informationssystemen eine zentrale Rolle. Sie zeichnen sich durch ein definiertes Schema, strikte Konsistenzregeln und eine leistungsfähige Abfragesprache aus, wodurch sie sich für eine Vielzahl an Anwendungen etabliert haben - insbesondere in Bereichen wie Unternehmenssoftware, Verwaltungssystemen und Finanzwesen. Dennoch stoßen relationale Datenbanken bei der Modellierung und Verarbeitung komplex vernetzter oder unstrukturierter Daten zunehmend an ihre Grenzen. In Anwendungsfeldern wie sozialen Netzwerken oder Empfehlungssystemen gewinnen daher spezialisierte NoSQL-Datenbanken, insbesondere Graphdatenbanken, zunehmend an Bedeutung. Dazu zählt unter anderem auch das Graphdatenbank-System Neo4j. Graphdatenbanken ermöglichen es, Informationen als Knoten und Kanten direkt abzubilden und so komplexe Beziehungsstrukturen zu analysieren.

Trotz dieser Vorteile bleibt der Zugriff auf fortgeschrittene Analysewerkzeuge aus dem relationalen Umfeld für Graphdatenbanken aufgrund ihrer strukturellen Unterschiede und der fehlenden Kompatibilität zu relationalen Modellen weitgehend verwehrt. Dies führt dazu, dass Systeme wie ProSA für Graphdatenbanken nicht einsetzbar sind. ProSA (Provenance Management using mappings Annotations) ist ein System zur Verwaltung und Analyse von Datenherkunft (Provenance) innerhalb relationaler Datenbanken. ProSA ist in der Lage, Anfrageergebnisse nach ihrer Herkunft, Reproduzierbarkeit und Nachvollziehbarkeit zu analysieren. Dazu wird die CHASE-Technik zur schematischen Datenbanktransformation mit Methoden des Provenance-Managements kombiniert. ProSA setzt jedoch ein relationales Datenmodell voraus, um die jeweiligen Analyseprozesse anwenden zu können (Auge et al. (2022), Auge & Heuer (2019)). Aus diesem Grund befasst sich diese Bachelorarbeit mit der Entwicklung und Umsetzung eines strukturierten Verfahrens zum

Schema-Mapping einer Neo4j-Graphdatenbank in ein relationales Datenbanksystem, das sich nahtlos in ProSA integrieren lässt.

**Aufbau der Arbeit:** Die vorliegende Bachelorarbeit gliedert sich in mehrere aufeinander aufbauende Kapitel. Im Anschluss folgt in Kapitel 3 eine Einführung in die zentralen Begriffe und Konzepte relationaler sowie graphbasierter Datenbanksysteme. Kapitel 4 und 5 beleuchten den aktuellen Stand der Technik und Forschung. Darauf aufbauend wird in Kapitel 5 die methodische Vorgehensweise erläutert, welche in Kapitel 6 in ein konkretes Konzept und dessen technische Umsetzung überführt wird. Abschließend erfolgt in Kapitel 7 eine Evaluation des Systems, bei der die Korrektheit, Effizienz und Robustheit des Transformationsverfahrens bewertet wird und im Anschluss in Kapitel 8 die Ergebnisse diskutiert werden.

Zum besseren Verständnis der entwickelten Konzepte wird in dieser Arbeit ein Beispieldatensatz verwendet. Dieser bildet reale Strukturen ab und umfasst unter anderem Städte, Länder, Bundesstaaten und wirtschaftliche Zentren, die über verschiedene Beziehungstypen miteinander verknüpft sind.

# 2. Motivation

Die Fähigkeit, Daten nicht nur effizient zu speichern, sondern auch deren Entstehung und Entwicklung nachvollziehbar zu machen, gewinnt in vielen Anwendungsfeldern, insbesondere in der Forschung, an Relevanz. Werkzeuge wie ProSA ermöglichen eine gezielte Analyse der Datenherkunft (Provenance) und der zugrunde liegenden Transformationsschritte durch sogenannte Schema-Evolution. Jedoch ist ein solches System wie ProSA auf relationale Datenbanken ausgelegt und kann nicht mit Graphdatenbanken interagieren.

Graphdatenbanken wie Neo4j spielen eine zunehmend wichtige Rolle, insbesondere bei der Modellierung komplexer Knoten- und Kantenstrukturen. Um diese in das bestehende Provenance-Werkzeug integrieren zu können, bedarf es eines strukturierten Mappings von Graphdatenbanken in ein relationales Schema.

In der aktuellen Forschung existieren bereits erste Ansätze, die sich mit der Extraktion von Schemata aus Graphdatenbanken beschäftigen. Dieser Schritt stellt eine zentrale Voraussetzung dar, wenn eine Transformation in ein relationales Modell erfolgen soll. Dennoch handelt es sich hierbei nur um wenige Ansätze, die sich auch mit dem Aspekt von Multilabel-Knoten beschäftigen. Multilabeling ist ein zentrales Merkmal vieler Graphmodelle, welche die gleichzeitige Zugehörigkeit eines Knotens zu mehreren Labels beschreibt (Definition 3.2.8). Analog zur Schema-Extraktion zeigt sich auch im Bereich des Schema-Mappings eine vergleichbare Forschungslücke. Zwar existieren Verfahren, die sich mit der Abbildung von Graphdatenbankschemata in relationalen Strukturen befassen, jedoch wird auch hier der Aspekt von Multilabeling nicht berücksichtigt. Diese Ansätze bilden dennoch sowohl für die Schema-Extraktion als auch für das Mapping eine wertvolle Grundlage.

Die vorliegende Bachelorarbeit setzt an dieses Problem an und verfolgt das Ziel,

ein Transformationsverfahren zu entwickeln, das insbesondere auch den Umgang mit Multilabel-Knoten in der Graphdatenbank berücksichtigt. Durch die gezielte Erweiterung bestehender Ansätze zur Behandlung von Knotenstrukturen wird eine strukturierte und verlustfreie Überführung in ein relationales Datenbankschema ermöglicht. Auf diese Weise wird die Kompatibilität zwischen dem Graphdatenbanksystem Neo4j und relationalen Datenbanksystemen wie PostgreSQL gezielt verbessert. Außerdem ermöglicht dieses Verfahren den Einsatz von Provenance-Werkzeugen wie ProSA auch an Graphdatenbanken, die bislang ausschließlich für relationale Datenmodelle ausgelegt sind. Somit schafft das entwickelte Verfahren eine methodische Grundlage, um auch graphbasierte Datenmodelle in bestehende Nachverfolgbarkeitssysteme zu integrieren.

# 3. Grundlagen

Das Kapitel Grundlagen widmet sich der Klärung zentraler Begriffe und Konzepte im Kontext relationaler Datenbanken und Graphdatenbanken. Es werden wesentliche Definitionen, theoretische Grundlagen sowie relevante Zusammenhänge erläutert, um eine fundierte Basis für die nachfolgenden Untersuchungen und Analysen zu schaffen.

## 3.1. Relationale Datenbanken

Relationale Datenbanken bilden die Grundlage vieler klassischer Informationssysteme. Sie organisieren Daten in Tabellen (Relationen), die über Schlüsselbeziehungen miteinander verknüpft sind. Die folgenden Definitionen erläutern zentrale Bestandteile des relationalen Modells:

**Definition 3.1.1** (Relationale Datenbank,Palagashvili & Stupnikov (2023))**.** Eine *relationale Datenbank*

$D_R = \{r_1, r_2, \ldots, r_n\}$ besteht aus einer Menge von *Relationen* $r$.

**Definition 3.1.2** (Attribut, Heuer et al. (2018))**.** Sei $U$ eine nicht-leere, endliche Menge, das *Universum* der Attribute. Ein Element $A \in U$ heißt *Attribut*.

Das heißt: Ein Attribut ist eine Spalte in einer Tabelle einer relationalen Datenbank, die eine bestimmte Eigenschaft oder ein Merkmal der gespeicherten Daten beschreibt.

**Definition 3.1.3** (Attributwert, Heuer et al. (2018))**.** Ein *Attributwert* ist ein Wert $w$, den ein Attribut $A$ annehmen kann. Dabei gehört $w$ zur Domäne $dom(A)$, also zum Wertebereich, der für $A$ definiert ist.
Das heißt: Ein konkreter Wert, der in einer Zelle einer Tabelle steht und ein Attribut eines Datensatzes beschreibt.

**Definition 3.1.4** (Relation, Heuer et al. (2018))**.** Eine *Relation* $r$ über $R = \{A_1, \ldots, A_n\}$ (kurz: $r(R)$) mit $n \in \mathbb{N}$ ist eine endliche Menge von Abbildungen, die *Tupel* $t$ genannt werden, wobei $t(A) \in dom(A)$ gilt.

**Definition 3.1.5** (Entität, Silberschatz et al. (2010))**.** Eine *Entität* ist ein eindeutig identifizierbares Objekt der realen Welt, das in einer Datenbank gespeichert werden kann. In relationalen Datenbanken wird jede Entität durch ein Tupel in einer Relation (Tabelle) dargestellt. Eine Entität besitzt bestimmte Eigenschaften, die durch Attribute beschrieben werden.

**Definition 3.1.6** (Relationenschema, Heuer et al. (2018))**.** Eine Menge $R \subseteq U$ heißt *Relationenschema*. Sie beschreibt die Struktur einer Tabelle. Es legt die Anzahl, Bezeichnung und Typisierung der Spalten fest. Das heißt: Strukturierte Datenmenge in Tabellenform.

**Definition 3.1.7** (Datenbankschema, Heuer et al. (2018))**.** Eine Menge von Relationenschemata $S := \{R_1, \ldots, R_p\}$ mit $p \in \mathbb{N}$ heißt *Datenbankschema*.

**Definition 3.1.8** (Instanz, Heuer et al. (2018))**.** Für ein Relationenschema $R$ ist eine *Instanz* $r(R)$ eine konkrete Menge von Tupeln, die dem Schema $R$ entsprechen, wobei $r(R) \subseteq A^n$ ist. Dabei enthält jedes Tupel genau einen Wert pro Attribut des Schemas.
Das heißt: Eine Instanz beschreibt den aktuellen Satz an Tupeln in einer Tabelle zu einem bestimmten Zeitpunkt.

**Definition 3.1.9** (Beziehungstypen, Heuer et al. (2018))**.** Es gibt drei grundlegende *Beziehungstypen*, die die Verbindungen zwischen Tabellen beschreiben. Diese basieren auf den *Fremdschlüsseln*, die eine Tabelle mit einer anderen verbindet:

1. Bei einer *1:1-Beziehung* existiert zu jedem Datensatz in der ersten Tabelle genau ein Datensatz in der zweiten Tabelle
2. Bei einer *1:N-Beziehung* existieren zu einem Datensatz in der ersten Tabelle mehrere Datensätze in der zweiten Tabelle

3. Bei einer *N:M-Beziehung* können jedem Datensatz aus der einen Tabelle mehrere passende Datensätze aus der zweite Tabelle zugeordnet sein und umgekehrt.

**Definition 3.1.10** (Primärschlüssel, Heuer et al. (2018))**.** Für eine *Relation* $r(R)$, wobei $R$ eine Menge von Attributen ist, ist ein *Primärschlüssel* $P \subseteq R$ eine Menge von Attributen, die jeden Datensatz in einer Tabelle eindeutig identifiziert. Der Primärschlüssel wird zudem für die Zuordnung von Beziehungen zwischen Tabellen verwendet.

**Definition 3.1.11** (Fremdschlüssel, Heuer et al. (2018))**.** Ein *Fremdschlüssel* ist eine Liste von Attributen $X$ in einem Relationenschema $R_1$, wenn es in einem zweiten Schema $R_2$ eine passende Attributliste $Y$ gibt, die als Primärschlüssel fungiert. Dabei müssen alle Werte von $X$ in $r_1(R_1)$ auch in den entsprechenden Spalten $Y$ von $r_2(R_2)$ enthalten sein. Ein *Fremdschlüssel* stellt eine referenzielle Verknüpfung zwischen zwei Tabellen her. Er verweist auf einen existierenden Primärschlüssel in einer anderen Tabelle und stellt somit sicher, dass nur gültige Datenbeziehungen bestehen.

**Definition 3.1.12** (Fremdschlüsselbedingung, Heuer et al. (2018))**.** Eine *Fremdschlüsselbedingung* liegt für eine Relation $r_1(R_1)$ dann vor, wenn ein Ausdruck der Form $X(R_1) \rightarrow Y(R_2)$ existiert, wobei $X \subseteq R_1$ und $Y \subseteq R_2$ ist. In diesem Fall wird $X$ als *Fremdschlüssel* von $R_1$ bezüglich $Y$ in $R_2$ bezeichnet.

Eine Datenbankinstanz $d$ erfüllt diese Bedingung genau dann, wenn eine Relation $r_2(R_2)$ mit $Y$ als Primärschlüssel existiert und für alle Tupel $t$ aus $r_1$ gilt:

$$\{t(X) \mid t \in r_1\} \subseteq \{t(Y) \mid t \in r_2\}$$

Das heißt: Die *Fremdschlüsselbedingung* beschreibt eine Regel, dass ein Fremdschlüssel in einer Tabelle immer auf einen gültigen Datensatz in einer anderen/derselben Tabelle verweisen muss. Der Wert eines Fremdschlüssels in einer Tabelle muss einen Primärschlüsselwert in einer anderen Tabelle entsprechen.

Beispielsweise beschreibt die Tabelle *City* die Entität *City* mit den Attributen *id* und *name*. Jeder Datensatz wie $(8, \text{'Munich'})$ ist ein Tupel und entspricht einer Instanz der Relation.

## 3.2. Graphdatenbanken

Graphdatenbanken bieten ein flexibles Datenmodell, das auf der expliziten Speicherung von Beziehungen basiert. Sie ermöglichen eine intuitive Darstellung und Analyse vernetzter Informationen.

**Definition 3.2.1** (Graphdatenbank,Palagashvili & Stupnikov (2023))**.** Eine *Graphdatenbank* ist ein Datenbanksystem $D_G = (V, E)$, das aus einer Menge von *Knoten* $V = \{N_1, N_2, N_3, \dots\}$ und *Kanten* $E = \{E_1, E_2, E_3, \dots\}$ besteht.

**Definition 3.2.2** (Knoten, Palagashvili & Stupnikov (2023) )**.** Ein *Knoten* $N_i = (L, a, v)$ besteht aus einem bestimmten Label $L$, einer Menge von Attributen $a$ und den dazugehörigen Werten $v$.

**Definition 3.2.3** (Kante, Palagashvili & Stupnikov (2023) )**.** Eine *Kante* $E_i = (L, a, v, \text{-}node_a, node_b)$ wird durch ein Label $L$, einer Menge von Attributen $a$ mit ihren Werten $v$ und zwei miteinander verbundenen Knoten $node_a$ und $node_b$ definiert. Diese werden als Start- und Zielknoten $(node_a \rightarrow node_b)$ bezeichnet.

**Definition 3.2.4** (Label, Palagashvili & Stupnikov (2023) )**.** Ein *Label* $L$ wird definiert als eine Bezeichnung, die einer Entität wie einem Knoten oder einer Kante zugeordnet ist, um deren Zugehörigkeit zu einer Klasse zu markieren.

**Definition 3.2.5** (Graph, Frozza et al. (2020) )**.** Ein *Graph* $G = (V, E)$ ist ein Tupel bestehend aus einer Menge von Knoten $V$ und Kanten $E$. Die Knoten repräsentieren Entitäten, während Kanten die Beziehungen zwischen diesen Knoten beschreiben.

**Definition 3.2.6** (Property Keys, Frozza et al. (2020) )**.** Ein *Property Key* ist der Bezeichner einer Eigenschaft und dient dazu, innerhalb eines Key-Value-Paares auf den zugehörigen Wert zuzugreifen. Er stellt somit den Schlüssel dar, unter dem eine bestimmte Information gespeichert ist.

**Definition 3.2.7** (Key-Value-Pair, Yousaf & Wolter (2019) )**.** Eine Datenstruktur aus zwei Elementen:

- *Key* (Schlüssel): Eindeutiger Bezeichner für ein Datenelement
- *Value* (Wert): Zugehörige Information, die mit dem Key verbunden ist

**Definition 3.2.8** (Multilabel-Knoten, Frozza et al. (2020), )**.** Ein *Multilabel-Knoten* ist ein Knoten $N = (P, L)$, wobei $P$ die Menge der Eigenschaften (Properties) und $L$ die Menge der zugewiesenen Labels ist. Gilt $L > 1$, so besitzt der Knoten mehrere semantische Typisierungen und wird als *Multilabel-Knoten* bezeichnet. Dabei steht jedes Label für eine unterschiedliche konzeptuelle Zuordnung, sodass ein Knoten gleichzeitig mehreren Entitätstypen angehören kann.

**Definition 3.2.9** (Type, Palagashvili & Stupnikov (2023) )**.** Der *Type* beschreibt die semantische Kategorie, die ein Knoten oder eine Kante innerhalb eines Graphen einnimmt.

**Definition 3.2.10** (Knotenklasse, Frozza et al. (2020))**.** Eine *Knotenklasse* $NC$ fasst Knoten mit identischen Labels zusammen.

**Definition 3.2.11** (Kantenklasse, Frozza et al. (2020))**.** Eine *Kantenklasse* $EC$ beschreibt eine Menge von Kanten, die dieselbe Beziehung zwischen Knotenklassen modellieren .

**Definition 3.2.12** (Schema, Vicknair et al. (2010) )**.** Ein *Schema*
$S = (NC, EC, L_E, L_M, A_N, A_E)$ beschreibt die Struktur der Daten und definiert die Knotenklassen $NC$ und Kantenklassen $EC$, deren Labels $L_N, L_E$ und Attribute $A_N, A_E$. Graphdatenbanken verfolgen den Ansatz des schemafreien Modells. Die Struktur der Daten ist flexibel und kann immer wieder geändert werden.

Beispielsweise stellt in der verwendeten Beispieldatenbank ein Knoten mit dem Label *City* und den Eigenschafte *id:8* sowie *name: "Munich"* eine Entität der Klasse *City* dar. Sollte ein Knoten wie *Munich* zusätzlich das Label *EconomicHub* besitzen, handelt es sich um einen Multilabel-Knoten.

Damit bildet dieses Kapitel eine zentrale Grundlage für die nachfolgende Bachelorarbeit. Die definierten Konzepte und Begriffe ermöglichen ein präzises Verhältnis der im weiteren Verlauf dargestellten Methoden.

# 4. Stand der Technik

In diesem Kapitel werden die technologischen Grundlagen betrachtet, die für die Umsetzung dieser Arbeit relevant sind. Dazu gehören ProSA, ein System zur Sicherung der Data Provenance und zur Reproduzierbarkeit von Datenbankabfragen (Kapitel 4.1), Neo4j als Graphdatenbank zur Speicherung und Abfrage vernetzter Daten (Kapitel 4.2) sowie PostgreSQL als relationales Datenbanksystem mit leistungsfähigen Mechanismen zur strukturierten Datenspeicherung (Kapitel 4.3). Diese Technologien bieten unterschiedliche Ansätze zur Datenorganisation und -verarbeitung, die im weiteren Verlauf der Arbeit für die Modellierung und Umsetzung des Mappings zwischen Graph- und relationalen Datenbanken von Bedeutung sind.

## 4.1. ProSA und ihre Funktionen

In relationalen Datenbanken kann es durch Projektionen oder andere Operatoren dazu kommen, dass identische Datensätze zu einer einzigen aggregierten Darstellung zusammengefasst werden. Dies geschieht insbesondere bei Abfragen, die nur bestimmte Attribute (Definition 3.1.2) auswählen, während die ursprünglichen Daten mit zusätzlichen Informationen erhalten bleiben. Nach einer Aktualisierung oder Erweiterung der Datenbank kann es jedoch vorkommen, dass diese zusammengefassten Datensätze nicht mehr eindeutig rekonstruierbar sind, da die Differenzierungsmerkmale fehlen. Dies führt zu Problemen bei der Nachverfolgbarkeit und Reproduzierbarkeit von Analysen.

Die Konsistenz und Reproduzierbarkeit von Datenbankauswertungen erfordert Methoden, die es ermöglichen, die ursprüngliche Datenstruktur auch nach Änderungen nachzuvollziehen und wiederherzustellen. ProSA adressiert diese Problematik, indem es die Provenance der Daten sichert und eine minimale, aber den-

noch reproduzierbare Teilmenge der Originaldatenbank berechnet. Dadurch wird sichergestellt, dass trotz Löschung oder Veränderung einzelner Datensätze alle notwendigen Informationen für eine spätere Rekonstruktion erhalten bleiben. Die zugrundeliegende CHASE-Technik ermöglicht es, Abfragen auf Basis historischer Abhängigkeitsstrukturen nachzuvollziehen und somit die ursprüngliche Datenbasis für eine Reanalyse oder erneute Berechnung zu rekonstruieren (Auge (2023)) .

Insbesondere in wissenschaftlichen Disziplinen, in denen Datenbanken regelmäßig überarbeitet werden, spielt die Reproduzierbarkeit von Analysen eine entscheidende Rolle. Ohne geeignete Mechanismen zur Nachverfolgung von Änderungen könnten Forschungsergebnisse durch inkonsistente oder unvollständige Daten verzerrt werden. ProSA ermöglicht es, selbst nach Datenbankmodifikationen jene Daten zu identifizieren, die zur ursprünglichen Berechnung eines Ergebnisses beigetragen haben, und trägt somit zur wissenschaftlichen Validität und Transparenz bei.

### 4.1.1. Die CHASE-Technik und ihre Anwendung in ProSA

Die CHASE-Technik ist ein grundlegendes Werkzeug in der Datenbankforschung und dient der Anwendung von Abhängigkeiten und Transformationen auf Datenbanken. Diese Technik wird verwendet, um:

- Abfragen zu optimieren,
- Daten zwischen unterschiedlichen Datenbankmodellen zu transformieren,
- Datenintegrität durch Bereinigung und Konsistenzprüfungen sicherzustellen (Benedikt et al. (2017)).

ProSA nutzt den CHASE nicht nur zur Vorwärtsausführung von Abfragen, sondern kombiniert diese mit einer sogenannten *Backchase*-Phase. Diese Rückwärtsausführung ermöglicht es, den Ursprung von Abfrageergebnissen nachzuvollziehen und die minimalen Teilmengen der Daten zu identifizieren, die für deren Berechnung erforderlich sind (Auge & Heuer (2019)).

### 4.1.2. Funktionen und Merkmale von ProSA

ProSA bietet eine Vielzahl von Funktionen. Zu den wichtigsten Funktionen gehören:

**Datenminimierung:** Eine der Kernfunktionen von ProSA ist die Berechnung einer minimalen Teilmenge der ursprünglichen Datenbank. Diese Teilmenge, auch als *Sub-Datenbank* bezeichnet, enthält nur die Daten, die für die Reproduktion einer konkreten Anfrage notwendig sind. Dabei wird sichergestellt, dass die resultierende Datenbank sowohl platzsparend als auch vollständig ist (Auge (2023)).

**Datenherkunft (Provenance):** ProSA unterstützt verschiedene Arten der Datenherkunftsanalyse:

- *Why-Provenance*: Identifiziert die spezifischen Daten, die zur Berechnung eines Ergebnisses beigetragen haben (Herschel et al. (2017)).
- *How-Provenance*: Dokumentiert die Sequenz von Operationen, durch die ein Ergebnis erzielt wurde (Herschel et al. (2017)).
- *Where-Provenance*: Gibt die ursprünglichen Speicherorte der verwendeten Daten an (Herschel et al. (2017)).

Durch die Nutzung dieser Provenance-Informationen kann nachvollzogen werden, wie Ergebnisse zustande gekommen sind, und sie gegebenenfalls validieren oder korrigieren (Auge & Heuer (2019); Auge et al. (2022)).

### 4.1.3. Die ProSA-Pipeline

Die ProSA-Pipeline ist ein strukturierter Prozess, der in mehrere Schritte unterteilt ist:

1. **Parsen der Abfragen:** Die ursprüngliche Abfrage wird in eine Menge von Abhängigkeiten umgewandelt (Auge (2023)).
2. **Integration von Provenance-IDs:** Jede Datenbankzeile erhält eine eindeutige Provenance-ID, die für die Rückverfolgung genutzt wird (Auge (2023)).

3. **CHASE-Phase:** Die Abfrage wird auf die Datenbank angewendet, um das Abfrageergebnis zu berechnen (Auge (2023)).
4. **Backchase-Phase:** Eine Rückwärtsanalyse wird durchgeführt, um die minimale Sub-Datenbank zu identifizieren (Auge (2023)).
5. **Anonymisierung und Validierung:** Die minimale Sub-Datenbank wird überprüft und anonymisiert, bevor sie archiviert oder veröffentlicht wird (Auge (2023)).

ProSA bietet eine robuste und flexible Lösung für das Management von Forschungsdaten. Durch die Kombination moderner Datenbanktechnologien mit umfangreichen Provenance-Mechanismen ermöglicht ProSA die Minimierung der zu speichernden Datenmengen, ohne die wissenschaftliche Nachvollziehbarkeit zu beeinträchtigen.

## 4.2. Neo4j

Neo4j ist eine Graphdatenbank (Definition 3.2.1), die speziell für die effiziente Speicherung und Analyse stark vernetzter Daten entwickelt wurde (Neo4j Inc. (2025)). Durch die Verwendung eines Knoten-Kanten-Modells, wo mehrere Knoten und Kanten einen Graph (Definition 3.2.5) bilden, eignet sich Neo4j besonders für Anwendungen, bei denen die Beziehungen zwischen den Daten im Vordergrund stehen. Beispiele hierfür sind soziale Netzwerke, Betrugserkennung und Empfehlungssysteme. Als native NoSQL-Datenbank, die von Grund auf speziell für ein nicht-relationales Datenmodell entwickelt wurde, implementiert Neo4j typischerweise das Property-Graph-Modell. Dabei handelt es sich um eine flexible Datenstruktur, bei der sowohl Knoten (Definition 3.2.2) als auch Kanten (Definition 3.2.3) beliebige Eigenschaften in Form von Schlüssel-Wert-Paaren (Definition 3.2.7) enthalten können. Die Modellierungskonzepte unterscheiden sich teilweise hinsichtlich unterstützter Datentypen, Multilabels (Definition 3.2.8), Kantenrichtung und Komplexität der Properties (Frozza et al. (2020)).

In der biomedizinischen Forschung wird Neo4j zunehmend eingesetzt, um komplexe Datenbeziehungen zu modellieren und zu analysieren. Ein Beispiel für den

Einsatz von Neo4j ist die Entwicklung eines Graphmodells der Zelle durch Knowing Health. Dieses Modell wird vom Deutschen Zentrum für Diabetesforschung (DZD) genutzt, um biologische Daten besser zu verstehen und neue Erkenntnisse in der Diabetesforschung zu gewinnen (Preusse (2017)).

Ein weiteres Beispiel für den Einsatz von Neo4j ist im Bereich der Genomik und Bioinformatik. Hier wurde das Framework *phyloDB* entwickelt, das auf Neo4j basiert und die effiziente Speicherung, Verarbeitung und Analyse phylogenetischer Daten ermöglicht. Dieses Framework unterstützt Forscher bei der Durchführung großflächiger phylogenetischer Analysen und hat sich als leistungsfähiges Werkzeug in der Forschung etabliert (Lourenço et al. (2023)).

Diese Beispiele verdeutlichen, wie Neo4j in verschiedenen Bereichen der biomedizinischen Forschung eingesetzt wird, um komplexe Datenbeziehungen zu modellieren und neue wissenschaftliche Erkenntnisse zu gewinnen.

## 4.3. PostgreSQL

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem (ORDBMS), das für seine Erweiterbarkeit und Leistung bekannt ist. Es bietet eine Vielzahl moderner Funktionen wie zum Beispiel Transaktionen oder Fremdschlüsselunterstützung (Definition 3.1.11), Mehrversionen-Kontrolle (MVCC) und Unterstützung für benutzerdefinierte Datentypen. Diese Eigenschaften machen es besonders geeignet für anspruchsvolle Datenmanagement-Anwendungen (Stonebraker & Rowe (1991)).

Ein wesentlicher Vorteil von PostgreSQL liegt in seiner zuverlässigen Transaktionsverarbeitung sowie der effizienten Verwaltung umfangreicher Mengen an strukturierten und semistrukturierten Daten. Zudem unterstützt PostgreSQL das Erstellen von materialisierten Sichten, komplexen Aggregationen und Indexstrukturen, was die Eignung von PostgreSQL für leistungsintensive Anwendungen unterstreicht (Silberschatz et al. (2010)).

Ein Beispiel für den Einsatz von PostgreSQL ist seine Verwendung bei der Speicherung und Analyse von Big-Data-Workloads in der Genomforschung. Durch die Integration von Erweiterungen wie PostGIS und PL/Python können komplexe räum-

liche und algorithmische Berechnungen direkt innerhalb der Datenbank ausgeführt werden. Dies reduziert den Bedarf an externen Analysen und erhöht die Effizienz. Eine Studie zeigte, dass PostgreSQL-basierte Pipelines bei der Analyse von Genomdaten sowohl hinsichtlich Leistung als auch Genauigkeit mit spezialisierten Tools konkurrieren können (Sanderson et al. (2021)).

Darüber hinaus wird PostgreSQL häufig in der Wirtschaft und Wissenschaft als Grundlage für hochgradig skalierbare und replizierbare Datenbanken verwendet. Erweiterungen wie TimescaleDB, die auf PostgreSQL basieren, bieten Funktionen zur Verwaltung von Zeitreihendaten, die besonders in der IoT-Datenanalyse und bei wissenschaftlichen Langzeitstudien nützlich sind (Team (2018)).

Auf Basis dieser technologischen Konzepte erfolgt im nächsten Kapitel die Auseinandersetzung mit bestehenden wissenschaftlichen Ansätzen und Entwicklungen im Kontext der Transformation zwischen Graph- und relationalen Datenmodellen sowie der Handhabung von Schema-Evolution.

# 5. Stand der Forschung

Dadurch, dass ProSA nur relationale Datenbankanfragen verarbeiten kann, muss die Graphdatenbank in ein relationales Schema (Definition 3.1.7) transformiert werden, damit das Provenance-Konzept von ProSA angewendet werden kann. Hierfür werden mehrere Schritte benötigt. Zuerst muss das Schema der Graphdatenbank (Definition 3.2.12) extrahiert werden (Kapitel 5.2), um im Anschluss das gewonnene Schema auf eine relationale Datenbank mappen zu können (Kapitel 5.3). Nach der erfolgreichen Transformation kann die Schema-Evolution auf der relationalen Datenbank durchgeführt werden, um Änderungen an der Datenstruktur vorzunehmen. Nachdem die Schema-Evolution durchgeführt wurde, wird die relationale Datenbank wieder zurück in eine Graphdatenbank transformiert. Sowohl bei der Schema-Extraktion, beim Mapping als auch bei der Schema-Evolution gibt es bereits verschiedene Ansätze, wie man dort vorgehen kann.

## 5.1. Schema Evolution

Schema-Evolution ist ein essenzielles Konzept in der Datenbankverwaltung, das sich mit der kontinuierlichen Anpassung des Schemas einer Datenbank befasst. In relationalen Datenbanken erfolgt diese Evolution häufig durch Änderungen an Tabellenstrukturen, wie das Hinzufügen oder Entfernen von Spalten, während in NoSQL -Systemen das Schema oft implizit bleibt und sich durch die Speicherung neuer Datenstrukturen verändert (Chillón et al. (2024)). Um diese Veränderungen beschreiben und umsetzen zu können, werden verschiedene Arten von Schemaänderungen durch sogenannte Evolutionsoperatoren klassifiziert. Ein zentraler Aspekt der Schema-Evolution ist daher die Einteilung der Evolutionsoperatoren in *uniäre* und *binäre* Operatoren.

Uniäre Operationen sind solche, die sich ausschließlich auf eine einzelne Entität

beziehen (Definition 3.1.5). Beispiele hierfür sind das Hinzufügen (*add*) eines neuen Knotentyps oder einer Eigenschaft, das Umbenennen (*rename*) einer bestehenden Entität, das Löschen einer Eigenschaft (*remove*) oder das Löschen aller Knoten eines Labels (*delete*). Diese Operatoren verändern lediglich die betroffene Entität, ohne dass weitere Schema-Elemente direkt modifiziert werden.

Binäre Operationen hingegen betreffen zwei Schema-Elemente und führen zu strukturellen Veränderungen, bei denen bestehende Entitäten direkt beeinflusst oder neue erzeugt werden. Dazu gehören beispielsweise das Kopieren einer Entität (*copy*), wodurch eine neue Instanz (Definition 3.1.8) mit identischen Eigenschaften erstellt wird. Ebenso zählen das Zusammenführen (*merge*) zweier Entitäten oder das Aufteilen (*split*) einer Entität in mehrere spezialisierte Entitäten zu dieser Kategorie. (Hausler (2024)).

Im Rahmen des PRISM-Projekts von Carlo Curino et al. wurde die Sprache *Schema Modification Operators* (*SMOs*) definiert, die typische Schemaveränderungen in einer Datenbank formal beschreibt. SMOs erfassen typische Evolutionsoperatoren wie ADD COLUMN, DELETE TABLE oder RENAME COLUMN.

Ein zentrales Merkmal dieser SMOs ist nicht nur die Beschreibung der Operation, sondern auch die Definition genauer Input- und Output-Zustände des Schemas. So wird beispielsweise bei einem RENAME-Operator auf Attributebene definiert, dass im Input-Zustand eine Tabelle Person(Name, Alter) existiert und im Output-Zustand das Attribut *Name* zu *Vorname* umbenannt wurde, sodass das resultierende Schema Person(Vorname, Alter) lautet. Somit lässt sich nachvollziehen, welche Teile des Schemas vor und nach der Anwendung eines SMOs betroffen sind und wie sie sich verändert haben (Curino et al. (2008)).

## 5.2. Schema Extraktion von NoSQL-Datenbanken

NoSQL-Datenbanken sind Datenbanksysteme, die speziell für die Verarbeitung von großen, verteilten und häufig unstrukturierten Datenmengen entwickelt wurden. Im Gegensatz zu relationalen Datenbanken verzichten sie auf ein fest vorgegebenes Schema. Ein wesentlicher Vorteil besteht darin, dass durch den Verzicht auf

starre Kontrollmechanismen zur Festlegung der Datenstruktur der administrative Aufwand signifikant verringert wird. Allerdings bringt dieses Konzept auch den Nachteil mit sich, dass das Fehlen eines festen Schemas die Extraktion einer klaren und strukturierten Datenrepräsentation erheblich erschwert. In der Forschung wird daher verstärkt versucht, Lösungen und Methoden zur Schema-Extraktion von NoSQL-Datenbanken zu entwickeln. Zu diesem Thema wurden bereits mehrere Papers verfasst, die jeweils verschiedene Methoden für die Schema-Extraktion von Graphdatenbanken vorstellen.

In der Arbeit von Frozza et al. wird ein Ansatz beschrieben, bei dem als Input eine NoSQL-Graphdatenbank verwendet wird, um ein Schema zu extrahieren und in einem JSON-Schema-Format zu repräsentieren. Hierbei werden auch Knoten mit mehreren Labels, sogenannten Multilabels, berücksichtigt. Multilabels ermöglichen es, Knoten gleichzeitig mehreren Kategorien zuzuordnen, was die Modellierung komplexer Datenstrukturen erheblich erleichtert. Dies bedeutet, dass ein Knoten nicht nur ein einzelnes Label wie *Person* besitzt, sondern zudem noch ein weiteres Label wie *Actor* besitzen kann, etwa um auszudrücken, dass eine Person als Schauspieler fungiert. Dadurch wird die semantische Repräsentation der Daten erweitert, da ein Knoten mehrere Rollen oder Eigenschaften gleichzeitig abbilden kann, ohne einen separaten Knoten anlegen zu müssen. Diese flexible Struktur erlaubt eine effizientere Abfrage und Analyse, da Multilabels Beziehungen und Gemeinsamkeiten zwischen verschiedenen Kategorien direkt integrieren. Um diese Informationen in ein relationales Modell zu überführen, müssen zunächst die relevanten Strukturen aus der Graphdatenbank identifiziert und aufbereitet werden. Der Schema-Extraktionsprozess besteht aus mehreren Schritten:

- **Grouping**: In diesem Schritt werden die Knoten der Datenbank untersucht und für jede vorkommende Label-Kombination eine eigene Gruppe generiert. Dabei werden Knoten mit denselben Labels in eine Gruppe zusammengefasst. Eigenschaften und Kanten werden zudem auch in den Gruppen gespeichert.
- **Factoring**: Im Anschluss werden die Gruppen von Knoten analysiert, um Gruppen mit mehr als einem Label (Multilabels) in Gruppen von Knoten mit nur

einem Label umzuwandeln. Dabei werden Labels identifiziert, die in mehreren Multilabel-Knoten vorkommen und somit als Supertype fungiert. Gemeinsame Eigenschaften und Beziehungen werden nur im Supertype gespeichert und im dazugehörigen Label auf den Supertype referiert.

- **Parsing**: Im letzten Schritt wird für jedes Schema eines einzelnen Knotens und einer Kante ein JSON-Schema-Dokument erstellt (Frozza et al. (2020)).

Ein weiterer Ansatz, der von Wattiau et al. erarbeitet wurde, fokussiert sich auf das modellgetriebene Reverse Engineering von NoSQL-Property-Graph-Datenbanken, am Beispiel von Neo4j. In ihrem Ansatz wird die Struktur der Datenbank analysiert und mithilfe von Transformationsregeln in ein erweitertes Entity-Relationship-Modell (ER) überführt. Das Reverse Engineering beschreibt den Ansatz, von einer bestehenden Implementierung zurück zu einem konzeptionellen/abstrakten Modell. Dabei nutzen sie Cyber-Abfragen, um Beziehungen und Knoten zu identifizieren und diese in ein konzeptionelles Modell zu transformieren. Cypher ist die Abfragesprache von Neo4j, die speziell für die Arbeit mit Graphdaten entwickelt wurde. Um es nun auf das ER-Schema zu mappen, werden Transformationsregeln benutzt. Diese legen fest, wie eine Beziehung oder eine Entität in einem anderen Modell umgewandelt werden kann (Comyn-Wattiau & Akoka (2017)).

In der Arbeit von Bonifati et al. und ihren Kollegen beschäftigen auch sie sich mit der Schema-Extraktion. Sie entwickelten eine Anwendung namens *DiscoPG*. Ihr Fokus liegt hierbei auf der automatisierten Entdeckung von Graphschemata. Mithilfe von hierarchischem Clustering und einem Gaussian Mixture Model können Knoten und ihre Eigenschaften analysiert werden, um Muster zu erkennen und Knoten in Gruppen oder Cluster zu unterteilen. Im Gegensatz zum Reverse-Engineering-Ansatz basiert die Extraktion nicht auf festen Transformationsregeln, sondern extrahiert das Schema dynamisch (Bonifati et al. (2022).

## 5.3. Schema-Mapping

Das Mapping von Graphdatenbanken auf relationale Datenbanken stellt den nächsten Schritt nach der Extraktion der Daten dar. Es dient dazu, die in der Graphdaten-

bank vorhandenen Knoten und Beziehungen in ein relationales Modell zu überführen, ohne die Struktur und Semantik der ursprünglichen Daten zu verlieren. Dieser Prozess ist essenziell, um die Daten in einer relationalen Umgebung nutzen und verarbeiten zu können. Dabei gibt es verschiedene Ansätze und Methoden, die in der Forschung beschrieben werden. Neben der Transformation von Graphdatenbanken in relationale Strukturen existieren auch Verfahren für das Mapping von relationalen Datenbanken auf Graphdatenbanken. Diese Ansätze werden für die Arbeit ebenfalls berücksichtigt, da im Anschluss an die Schema-Evolution eine Rücktransformation in ein Graphmodell erfolgt.

Ein Ansatz, der von Palagashvilia et al. beschrieben wird, fokussiert sich auf die Möglichkeit, Daten bidirektional zu konvertieren. Dabei wird eine Methode vorgestellt, die es erlaubt, Daten sowohl von graphbasierte in relationale Modelle als auch umgekehrt zu übertragen. Dieses reversible Mapping gewährleistet, dass die ursprüngliche Datenstruktur vollständig rekonstruiert werden kann. Bei der Überführung des Graphschemas auf ein relationales Schema werden Knoten als Entitätstabellen mit ihren Eigenschaften abgebildet, während Kanten als separate Beziehungstabellen modelliert werden. Bei der Rücktransformation werden alle Relationen, die keine Many-to-Many-Beziehungen darstellen, zu Knotenklassen und deren Tupel bilden die einzelnen Knoten ab. Alle Many-to-Many-Relationen, die genau zwei Fremdschlüssel besitzen, werden als Kantenklassen dargestellt. Die einzelnen Tupel werden zu Kanten zwischen den Knoten, auf die sie referenzieren. Dadurch wird eine flexible Nutzung beider Datenmodelle ermöglicht. Die Arbeit hebt hervor, wie wichtig es ist, die Eigenschaften und Beziehungen von Knoten in einer Graphdatenbank korrekt in Tabellen und Beziehungstabellen zu überführen (Palagashvili & Stupnikov (2023)).

Ein weiterer Forschungsbeitrag von Virgilio et al. beschäftigt sich primär mit der Konvertierung relationaler Daten in Graphmodelle. Dabei werden Schema und Integritätsbedingungen wie Primär- und Fremdschlüssel genutzt, um eine effiziente Struktur zu erzeugen und SQL-Anfragen in Graphabfragen zu übersetzen. Anstatt für jedes Tupel einen eigenen Knoten zu erzeugen, gruppiert der Ansatz logisch

zusammengehörige Daten in gemeinsame Knoten. Während in einer relationalen Datenbank zwei Tabellen wie User und Comment über einen Fremdschlüssel verknüpft sind, werden diese Informationen im Graphmodell in einem gemeinsamen Knoten zusammengefasst, wenn sie häufig gemeinsam abgefragt werden. Somit wird die Anzahl der Knoten und Kanten reduziert und beschleunigt komplexere Abfragen (De Virgilio et al. (2013)).

Die Arbeit von Wardani et al. legt den Schwerpunkt auf die Berücksichtigung semantischer Beziehungen. Ziel ist es, beim Mapping von relationalen zu Graphmodellen keine semantischen Informationen zu verlieren, indem die ursprünglichen Bedeutungen von Entitäten und Beziehungen aus dem ER-Modell berücksichtigt werden. In einer relationalen Datenbank wird eine Many-To-Many-Beziehung (Definition 3.1.9) wie *Casting* zwischen *Actor* und *Movie* als eigene Tabelle dargestellt. Das semantische Mapping erkennt diese Struktur korrekt als Beziehung und modelliert sie im Graphen als Kante zwischen den beiden Knoten *Actor* und *Movie*. Dadurch bleibt die ursprüngliche semantische Abstraktion erhalten (Wardani & Kiing (2014)).

Damit sind alle erforderlichen theoretischen und technischen Grundlagen geschaffen, um im nächsten Kapitel auf deren Basis den methodischen Ansatz des Transformationssystems vorstellen zu können.

# 6. Methodik

Dieses Kapitel gibt einen Überblick über die methodische Herangehensweise bei der Umsetzung der Bachelorarbeit. Grundlage bildet der in Abbildung 1 dargestellte Ablauf, in dem die einzelnen Schritte der Transformation, Schema-Evolution und Rückübertragung abgebildet sind. Beginnend mit einer Graphdatenbank wird diese in ein relationales Schema überführt, und im nächsten Schritt überprüft, ob das Mapping erfolgreich war. Nach dem Vergleich beider Datenbanken erfolgt das Einbinden der gemappten relationalen Datenbank in ProSA. Dabei wird sowohl in ProSA als auch auf der Graphdatenbank eine identische Schema-Evolution durchgeführt. Nach dem Abgleich beider Datenbanken wird die relationale Datenbank wieder in ein Graphschema zurücktransformiert.

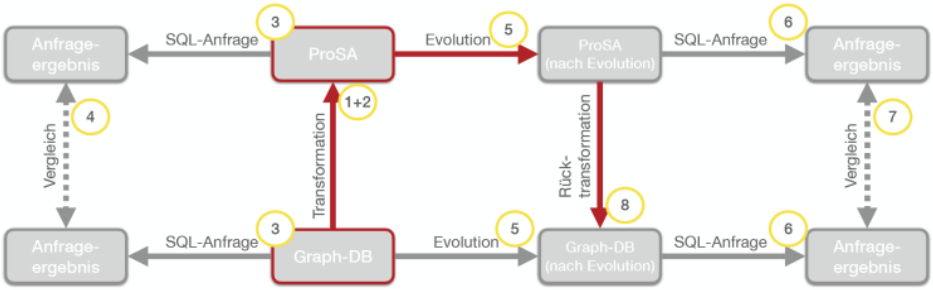



Abbildung 1: Ablauf des Schema-Mappings und Schema-Evolution zwischen Graph- und relationaler Datenbank.

## 6.1. Beispieldatenbanken

In diesem Kapitel werden die verwendeten Beispieldatenbanken beschrieben. Die Graphdatenbank dient als Ausgangspunkt für die Schema-Extraktion, während die relationale Datenbank das Zielmodell darstellt. Beide Datenbanken basieren auf demselben Datenbestand zu Städten, Hauptstädten, Ländern und wirtschaftlichen

Zentren sowie deren Beziehungen untereinander, sind jedoch aufgrund des jeweiligen Datenbanksystems unterschiedlich strukturiert.

### 6.1.1. Beispieldatensatz als Graphdatenbank

Die Graphdatenbank modelliert die Daten und deren Beziehungen in Form von Knoten und Kanten. Dabei repräsentieren die Knoten (Definition 3.2.2) die Entitäten wie Städte, Bundesstaaten, Länder und wirtschaftliche Zentren. Die Kanten (Definition 3.2.3) beschreiben die Verbindungen zwischen diesen Entitäten und erlauben eine flexible und effiziente Abbildung der Beziehungen. Diese Graphdatenbank wird in Neo4j mit der Sprache Cypher erstellt. Da es in Neo4j die Modellierung von Multilabel-Knoten unterstützt, enthält die Datenbank auch Beispiele von Knoten, die mehr als ein Label besitzen.


Abbildung 2: Graphdatenbankschema, mit Knoten und ihren Kanten

Jeder Knoten, der in Abbildung 2 abgebildet ist, enthält spezifische Eigenschaften, die die Entitäten genauer definieren:

- **City (Stadt):** enthält Attribute wie Name und eine eindeutige ID

- **State (Bundesstaat):** enthält Attribute wie Name und eine eindeutige ID
- **Country (Land):** enthält Attribute wie Name und eine eindeutige ID
- **EconomicHub (Wirtschaftszentrum):** enthält Attribute wie das Bruttoinlandsprodukt (GDP), um wirtschaftliche Zentren zu modellieren und eine eindeutige ID

Folgende Multilabel-Knoten sind in der Graphdatenbank enthalten:

- **City, Economichub:** Diese Multilabel-Knoten repräsentieren Städte, die gleichzeitig wirtschaftliche Zentren sind.
- **Country, Economichub:** Diese Knoten stellen Länder dar, die als wirtschaftliche Zentren fungieren.

Die Kanten (*Edges*) definieren die Beziehungen zwischen den Knoten. Dazu gehören:

- **data:** Diese Kanten verbinden Knoten mit sich selbst, um zusätzliche Informationen oder Eigenschaften innerhalb eines Knotens zu speichern. Sie speichert Werte wie Temperatur und den Monat
- **is_in:** Diese Kanten beschreiben hierarchische oder geografische Beziehungen, wie beispielsweise die Zugehörigkeit einer *City* zu einem *State* oder eines *State* zu einem *Country*
- **has:** Diese Kanten repräsentieren Besitztümer oder spezifische Zuordnungen. Ein Beispiel hierfür ist ein *State*, der eine *City* als Hauptstadt hat

### 6.1.2. Anforderungen

Damit die Transformation mit diesem System erfolgreich durchgeführt werden kann, muss die Graphdatenbank folgende Anforderungen erfüllen:

- Jeder Knoten muss eine eindeutige ID besitzen. Dafür kann die automatischgenerierte ID aus Neo4j benutzt werden, die als ganzzahliger Wert vergeben wird und innerhalb der Datenbank eindeutig ist.
- Jeder Beziehungstyp muss eine eindeutige ID besitzen.

### 6.1.3. Beispieldatensatz als Relationale Datenbank

Nach Abschluss des Mappings liegt die ursprüngliche Graphdatenbank in relationaler Form vor. Die resultierende relationale Datenbank entspricht dabei strukturell der zuvor manuell in PostgreSQL implementierten Zielstruktur.

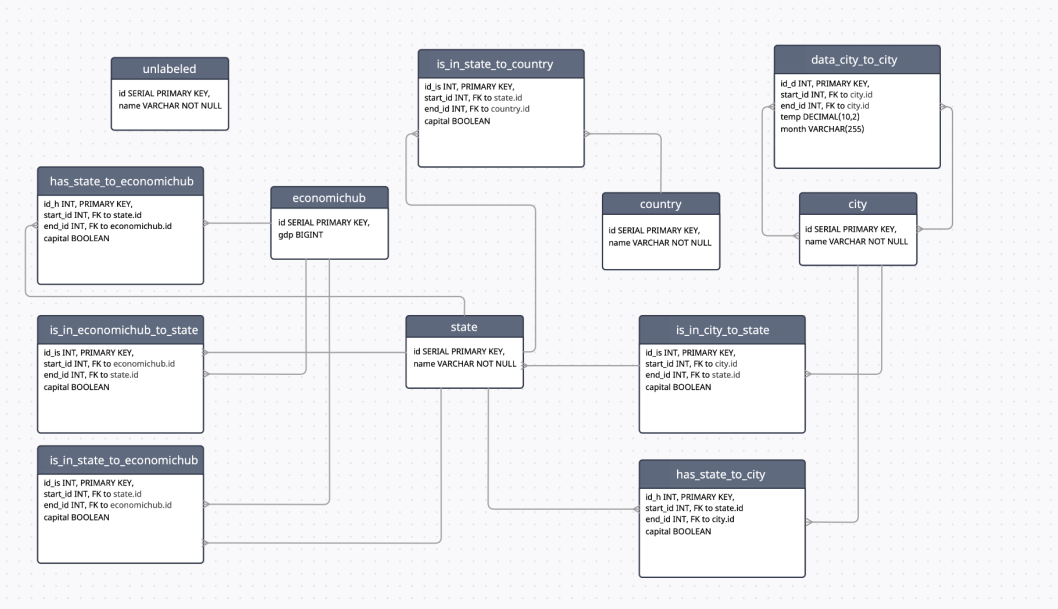



Abbildung 3: Datenbankdiagramm mit Entitäts- und Beziehungstabellen

Diese Datenbank besteht aus fünf Entitätstabellen und sieben Beziehungstabellen (Abbildung 3). Im relationalen Modell wird jede Entität (Definition 3.1.5) durch eine eigene Tabelle dargestellt, wobei jede Tabelle eine eindeutige ID-Spalte besitzt, die als Primärschlüssel (Definition 3.1.10) dient. Die Entitätstabellen in der relationalen Datenbank:

- **City (Stadt):** Speichert Informationen über Städte, z. B. *ID* (eindeutige ID) und *name* (Name der Stadt).
- **State (Bundesstaat):** Modelliert Bundesstaaten mit Attributen wie *ID* (eindeutige ID des Bundesstaates) und *name* (Name des Bundesstaates).
- **Country (Land):** Enthält Informationen zu Ländern, z. B. *ID* (eindeutige ID) und *name* (Name des Landes).
- **EconomicHub (Wirtschaftszentrum):** Repräsentiert wirtschaftliche Zentren mit *ID* (eindeutige ID) und *gdp* (Bruttoinlandsprodukt).

- **Unlabeled:** In dieser Tabelle werden alle Knoten gespeichert, die keine Labels besitzen.

Die Beziehungen zwischen den Entitäten werden durch zusätzliche Tabellen dargestellt. Dabei enthält jede Tabelle die Spalten *start_id* und *end_id*. In diesen beiden Spalten werden die IDs der beteiligten Entitäten gespeichert. Zudem definieren sie Fremdschlüsselbedingungen (Definition 3.1.12) zu den jeweils referenzierten Tabellen. Diese finden sich auch im Namen der Tabellen wieder. Diese folgen einem bestimmten Schema und machen damit sowohl die Richtung als auch den semantischen Kontext der Beziehung deutlich.

Im Folgenden werden die Beziehungstabellen vorgestellt. Die enthaltenen Attribute zur Beschreibung der Eigenschaften entsprechen dabei denen des zugrunde liegenden Graphmodells:

- **data_city_to_city:** Diese Tabelle repäsentiert selbstreferenzielle Beziehungen innerhalb der Entität *City* und enthält temperaturbezogene Informationen. Beziehungsart:*1:1-Beziehung.*
- **has_state_to_city:** Diese Beziehungstabelle bildet die Eigenschaften von Städten in Bezug auf ihren Bundesstaat ab, z. B. ob eine Stadt Hauptstadt ist. Beziehungsart: *1:N-Beziehung*.
- **has_state_to_economichub:** Diese Tabelle beschreibt Eigenschaften von wirtschaftlichen Knotenpunkten im Zusammenhang mit Bundesstaaten. Beziehungsart: *1:N-Beziehung.*
- **is_in_city_to_state:** Diese Tabelle speichert Zugehörigkeitsbeziehungen, indem sie darstellt, welche *City* zu welchem *State* gehört. Beziehungsart: *1:N-Beziehung.*
- **is_in_state_to_country:** Hier werden die Beziehungen zwischen *State* und *Country* abgebildet. Beziehungsart: *1:N-Beziehung*.
- **is_in_state_to_economichub:** Diese Tabelle verknüpft Bundesstaaten mit wirtschaftlichen Knotenpunkten. Beziehungsart: *N:M-Beziehung.*
- **is_in_economichub_to_state:** Diese Tabelle beschreibt die umgekehrte Beziehung zwischen *Economichub* und *State*. Diese Beziehung entsteht, da es einen

Multilabel-Knoten *City, Economichub* gibt. Beziehungsart: *N:M-Beziehung.*

## 6.2. Extraktion des Schemas

Im nächsten Schritt erfolgt die Extraktion des Schemas aus der Graphdatenbank. Dies bildet einen essenziellen Schritt bei der Überführung einer Graphdatenbank in ein relationales Modell. Dabei werden zu Beginn die strukturellen Eigenschaften der Graphdatenbank analysiert, um die enthaltenen Entitäten, deren Attribute sowie die zwischen ihnen bestehenden Beziehungen zu identifizieren. Ziel ist es, eine formalisierte Darstellung der Datenstruktur im JSON-Format zu erhalten, die dann im nächsten Schritt als Grundlage für das Mapping in das relationale Schema dient. Die Vorgehensweise zur Schema-Extraktion basiert auf dem theoretischen Ansatz von Angelo A.Frozza und Team (Frozza et al. (2020)). Hierbei wird die Graphdatenbank untersucht und in einem JSON-Schema-Format abgebildet. Besonders berücksichtigt werden hierbei Multilabel-Knoten. Diese müssen nämlich speziell behandelt werden, da Multilabels nicht in relationalen Datenbanken existieren. Beispielsweise kann ein Knoten nicht nur das Label *City* tragen, sondern gleichzeitig auch das Label *EconomicHub*. Bevor das Schema extrahiert werden kann, müssen alle Knoten und Kanten mit ihren Eigenschaften aus der Graphdatenbank extrahiert werden. Dies geschieht mit einer einfachen Anfrage an die Datenbank, die alle Daten der Datenbank zurückgibt und in einem JSON-Format für weitere Schritte speichert. Da rohe Graphdaten oft unstrukturiert sind, ist eine systematische Verarbeitung erforderlich, um daraus ein nutzbares Schema zu generieren. Um sicherzustellen, dass alle relevanten Informationen berücksichtigt werden, wird die Extraktion in mehreren Schritten durchgeführt, die jeweils die Daten aufbereiten, analysieren und für das Mapping vorbereiten.

Die Extraktion erfolgt in drei aufeinander folgenden Schritten:

1. **Grouping:** Zunächst werden alle Knoten der Graphdatenbank untersucht und in Gruppen zusammengefasst, basierend auf den ihnen zugewiesenen Labels. Innerhalb dieser Gruppen werden zusätzlich ihre Eigenschaften und Kanten erfasst.

2. **Factoring**: Anschließend erfolgt eine detaillierte Analyse der Gruppen mit mehr als einem Label. Diese werden in Gruppen mit jeweils nur einem Label zerlegt, wobei eine Vererbungshierarchie aufgebaut wird. Dabei wird ein *Supertype* identifiziert - ein übergeordnetes Label, dass in mehreren Knoten mit unterschiedlichen Labels vorkommt und gemeinsame Attribute und Beziehungen enthält. Ein Supertype ist somit ein abstrakter Knoten, der die Eigenschaften und Beziehungen zusammenführt, die in mehren Knoten mit unterschiedlichen Labels vorkommen und verhindert somit doppelte Tupel im Schema. Falls es verschiedene Kombinationen von Multilabel-Knoten gibt, können auch mehrere Supertype existieren. Dies ermöglicht eine systematische Weitervererbung, was die spätere Abbildung auf das relationale Modell erleichtert.
3. **Parsing**: Schließlich werden die gruppierten Knoten und Supertypes mit ihren Eigenschaften und Beziehungen in einem JSON-Dokument gespeichert.

Nach der Durchführung dieser drei Schritte liegt das extrahierte Schema in einer strukturierten Form vor und kann für das anschließende Mapping benutzt werden. Durch die systematische Gruppierung und Analyse wird sichergestellt, dass keine Informationen verloren gehen oder inkonsistent übertragen werden.

## 6.3. Mapping

Nachdem das Schema aus der Graphdatenbank extrahiert wurde, folgt im nächsten Schritt das Mapping in eine relationale Struktur. Ziel dieses Prozesses ist es, die gruppierten Knoten und Kanten der Graphdatenbank in eine Form zu überführen, die den Anforderungen eines relationalen Datensystems entspricht. Da relationale Datenbanken auf Tabellen mit festen Strukturen und expliziten Beziehungen basieren, müssen die aus der Graphdatenbank extrahierten Attribute und Beziehungen in eine geeignete tabellarische Form transformiert werden. Das Mapping erfolgt nach etablierten Methoden zur Konvertierung von Graphdatenmodellen in relationale Datenbanken. Dabei werden folgende grundlegende Prinzipien angewendet.

- **Knoten als Entitätstabellen**: Jeder Knoten mit einem bestimmten Label wird

in einer eigenen Tabelle gespeichert, wobei die Knotenattribute als Spalten übernommen werden.

- **Beziehungen als separate Tabellen**: Beziehungen zwischen Knoten werden mit ihren Attributen in einer eigenen Beziehungstabelle verwaltet.
- **Rekonstruktion von Multilabel-Knoten**: Multilabel-Knoten werden mithilfe des zuvor identifizierten Supertypen rekonstruiert, indem deren Informationen aus den beteiligten Singlelabel-Knoten sowie dem zugehörigen Supertype zusammengeführt werden. Die resultierenden Tupel werden anschließend in den jeweiligen Entitätstabellen gespeichert, wobei durch die gemeinsame ID die Zugehörigkeit sichergestellt wird.

Das Mapping wird in mehreren Schritten durchgeführt, um sicherzustellen, dass alle relevanten Informationen aus dem extrahierten Schema in das relationale Modell übernommen werden.

1. **Erstellung der Entitätstabellen**: Basierend auf dem extrahierten Graphdatenbank-Schema werden für jede identifizierte Entität separate Tabellen erstellt. Hierfür wird die automatisch generierte eindeutige ID aus der Graphdatenbank als Primärschlüssel verwendet. Außerdem enthält jede Tabelle die Attribute der jeweiligen Entität.
2. **Erstellung der Beziehungstabellen**: Da Graphdatenbanken Beziehungen explizit speichern, müssen diese in eigene Tabellen gespeichert werden. Hierfür wird zu Beginn überprüft, welche Beziehungen es zwischen den jeweiligen Knoten gibt. Jede Beziehungstabelle enthält die IDs der verbundenen Entitäten als Fremdschlüssel sowie zusätzliche Eigenschaften der Beziehungen. Die eindeutige ID der jeweiligen Beziehung wird ebenfalls aus dem Schema genommen, die zuvor in der Graphdatenbank erstellt wurde. Damit Start- und Endknoten in der Tabelle gespeichert werden, müssen hierfür die Spalten *start_id* und *end_id* erstellt werden, die jeweils auf die ID des Labels referieren und diese speichert. Zudem fungieren diese beiden Spalten als Foreign Keys. Falls ein Knoten mehrer Labels besitzt, wird die Beziehung in allen entsprechenden Tabellen gespeichert. Dadurch bleibt sichergestellt, dass die

Verknüpfungen unabhängig davon erhalten bleiben, in welcher Entitätstabelle der Knoten gespeichert wurde.

3. **Datenübertragung und Integration**: Nach der Erstellung der Tabellenstrukturen erfolgt die Befüllung mit den extrahierten Daten. Die Daten aus der Graphdatenbank werden den entsprechenden Tabellen zugeordnet und eingefügt. Sollte eine Entität für eine bestimmtes Attribut keinen Wert besitzen, wird diese Spalte mit *NULL* befüllt.

Nachdem das extrahierte Schema auf eine relationale Datenbank gemappt wurde, kann diese in ProSA eingebunden werden. Um jedoch die Gleichheit beider Datenbanken vergleichen zu können, erfolgt eine Schema-Evolution auf beiden Datenbanken.

## 6.4. Schema-Evolution

Da nun die Graphdatenbank in identischer Form als relationale Datenbank vorliegt, kann eine Schema-Evolution durchgeführt werden. Hierfür werden Evolutions-Operatoren wie *add, delete, remove, rename* oder *copy* benutzt und sowohl auf der Graphdatenbank als auch auf der relationalen Datenbank durchgeführt. Dabei wird eine Cypher-Anfrage in eine SQL-Anfrage konvertiert und auf der jeweiligen Datenbank ausgeführt. Damit die Evolution evaluiert werden kann, wird vor und nach der Evolution eine identische Anfrage gestellt. Im Anschluss können die Ergebnisse verglichen und überprüft werden, ob die Evolution auf beiden Datenbanken identische Auswirkungen hat.

## 6.5. Rücktransformierung

Im letzten Schritt muss die veränderte relationale Datenbank wieder in eine Graphdatenbank rücktransformiert werden. Hierbei wird zunächst das relationale Schema extrahiert und analysiert, um Tabellen und deren Beziehungen zu identifizieren. Anschließend werden die einzelnen Datensätze wieder in Form von Knoten mit ihren jeweiligen Eigenschaften gebracht und Cypher-Anfragen generiert, die die Da-

tenbank befüllen. Hierbei müssen wieder alle Multilabel rekonstruiert werden. Da die Schema-Evolution ebenfalls auf der extrahierten Graphdatenbank durchgeführt wurde, kann diese mit der zurücktransformierten Datenbank verglichen werden. Das Ziel ist es hierbei, dass beide Datenbanken keinerlei Unterschiede aufweisen und denselben Datenbestand haben. Somit liegt nach der Evolution wieder eine Graphdatenbank vor.

Die in diesem Kapitel beschriebenen methodischen Schritte bilden die konzeptionelle Grundlage für die technische Umsetzung. Im folgenden Kapitel wird dargelegt, wie diese Methodik konkret in der Implementierung realisiert wird.

# 7. Konzeption und Implementierung

In diesem Kapitel wird die Konzeption und anschließende Implementierung des entwickelten Konzepts zur Transformation von Graphdatenbanken in relationale Datenbanken detailliert beschrieben. Aufbauend auf der zuvor vorgestellten Methodik werden die einzelnen Komponenten des Systems sowie deren technische Umsetzung erläutert. Ziel ist es, die konzeptionellen Überlegungen in ein funktionierendes Softwaremodul zu überführen, das den gesamten Transformationsprozess inklusive Schema-Extraktion, Schema-Mapping, Schema-Evolution und Rücktransformation unterstützt.

## 7.1. System Design und Ablauf

Die Architektur des entwickelten Systems gliedert sich in vier funktionale Hauptbereiche:

1. **Schema-Extraktion**
2. **Schema-Mapping**
3. **Schema-Evolution**
4. **Rücktransformation**

Jeder Bereich wird durch ein eigenes Python-Modul realisiert, was eine saubere Trennung der Funktionalitäten gewährleistet. Der vollständige Quellcode ist im zugehörigen Repository auf GitHub verfügbar. [1] Abbildung 4 stellt den vollständigen Verarbeitungsprozess dar und zeigt die Wechselwirkungen zwischen den einzelnen Modulen sowie deren Anbindung an die jeweiligen Datenbanken:

Zu Beginn stellt *main.py* über Neo4jDatabase.py eine Verbindung zur Neo4j-Datenbank her und extrahiert mit einem Cypher-Befehl alle Knoten und Kanten (Schritt 1-

[1] `https://github.com/timohanoeffner0508/SchemaMappingNeo4jBachelorarbeit.git`

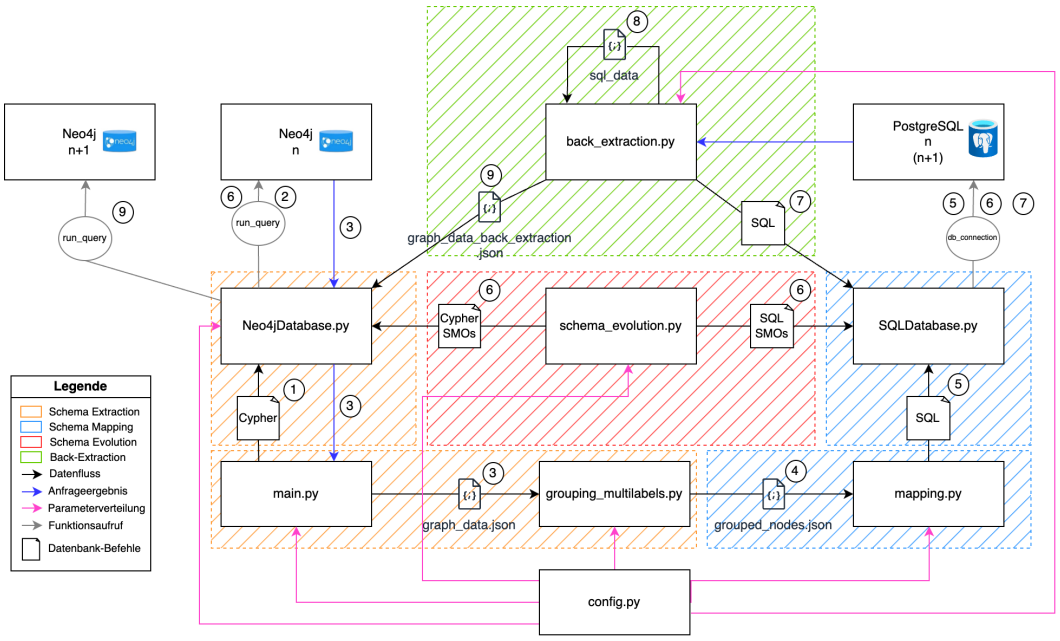



Abbildung 4: Architektur und Datenfluss der Implementierung

2). Das Anfrageergebnis wird aufgefangen und alle Ergebnisse in einer JSON-Datei gespeichert, welche im Anschluss von *grouping_multilabels.py* verarbeitet wird, alle Knoten gruppiert und einen Supertype identifiziert (Schritt 3). Das erstellte Schema dient als Eingabe für *mapping.py*, das die Daten in relationale Tabellen transformiert und SQL-Befehle erstellt (Schritt 4). Über *SQLDatabase.py* werden im nächsten Schritt alle SQL-Befehle auf der relationalen Datenbank ausgeführt (Schritt 5). In *schema_evolution.py* werden die benötigten Schema-Operatoren von Cypher in SQL-Befehlen geparst und diese auf der jeweiligen Datenbank ausgeführt (Schritt 6). Zum Schluss transformiert *back_extraction.py* die geänderte relationale Datenbank in eine Graphdatenbank. Hierfür bringt das Ergebnis einer SQL-Anfrage alle Tabellen der Datenbank (Schritt 7), transformiert es in ein Schema mit Knoten und Kanten (Schritt 8) und mappt es schließlich mit *Neo4jDatabase.py* auf eine neue Graphdatenbank (Schritt 9). Im Folgenden werde die verschiedenen Schritte im Detail vorgestellt.

## 7.2. Anbindung der Datenbanken

Für die Kommunikation mit den beiden Datenbanksystemen kommen zwei separate Klassen zum Einsatz: *Neo4jDatabase* ermöglicht den Zugriff auf die Graphdatenbank Neo4j, während *SqlDatabase* den Zugriff auf die relationale PostgreSQL-Datenbank bereitstellt. Auf diese Weise lassen sich alle Datenbankverbindungen über diese beiden Schnittstellen abwickeln.

Die Klasse *Neo4jDatabase* stellt Methoden zum Aufbau einer Verbindung über den offiziellen Neo4j-Treiber zur Verfügung (*Neo4j Python Driver 5.28* (2025)). Die Methode *init* initialisiert die Verbindung zur angegebenen Neo4j-Datenbank und speichert alle Zugangsdaten wie Benutzername, Passwort, Datenbankname und -url. Mithilfe der Methode *run_query* lassen sich Cypher-Anfragen mit ihren Parametern ausführen. Die Funktion *import_json_to_neo4j* verarbeitet eine strukturierte JSON-Datei, generiert aus dem Schema Cypher-Befehle, die im Anschluss auf der Datenbank ausgeführt werden. Diese Funktion wird benötigt, um das extrahierte relationale Schema wieder zurück in eine Graphdatenbank zu transformieren. Die Methode *close* beendet die Verbindung zur Datenbank ordnungsgemäß.

Die Klasse *SqlDatabase* dient als zentrale Schnittstelle zur Anbindung an eine PostgreSQL-Datenbank und ermöglicht zudem die Ausführung von SQL-Anweisungen. Hier wird das Modul *psycopg2* benutzt (Silberschatz et al. (2010)). Analog zur Neo4j-Klasse besitzt diese ebenfalls eine *Init*-Methode, die alle Zugangsdaten wie Benutzername, Passwort, Hostname, Port und Datenbankname der Ziel-Datenbank speichert. Über die Methode *create_database* kann eine neue SQL-Datenbank erstellt werden, indem zunächst eine Verbindung zur Standarddatenbank postgres aufgebaut und anschließend ein entsprechender *CREATE DATABASE*-Befehl ausgeführt wird. Die Methode *db_connection* stellt eine Verbindung zu einer bestehenden Datenbank her und liefert das *Connection*-Objekt und auch den zugehörigen *Cursor* zurück. Ein Datenbank-Cursor ist ein Kontrollobjekt, das innerhalb einer Datenbanksitzung verwendet wird, um SQL-Befehle auszuführen und Ergebnisse zu verwalten. Er fungiert als eine Art Zeiger, der durch die Ergebnismenge einer SQL-Abfrage navigiert. Der Cursor übernimmt die Aufgabe, SQL-Befehle an die Datenbank zu senden, das

Ergebnis zu empfangen und um dieses im Anschluss verarbeiten zu können.

Diese Klassen bilden somit die technische Grundlage für alle weiteren Operationen innerhalb des Systems, wo eine Anbindung zur Datenbank benötigt wird (Abbildung 5).

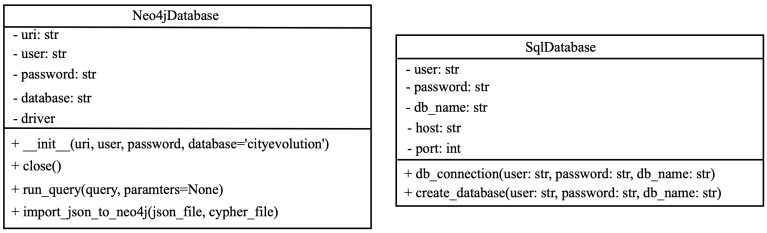



Abbildung 5: UML-Diagramm der Klassen Neo4jDatabase und SqlDatabase mit ihren Attributen und Methoden

## 7.3. Schema Extraktion (Schritt 1+3)

Die Schemaextraktion dient dazu, die Struktur der Graphdatenbank zu analysieren und für eine relationale Speicherung vorzubereiten. Der Prozess gliedert sich in vier Hauptphasen: die Extraktion der Rohdaten, die Gruppierung der Knoten nach Labels, die Verarbeitung von Multilabel-Knoten und die Speicherung der extrahierten Daten. Jede Phase wird in einer spezifischen Funktion innerhalb der Datei *main.py* und *grouping_multilabels.py* abgebildet.

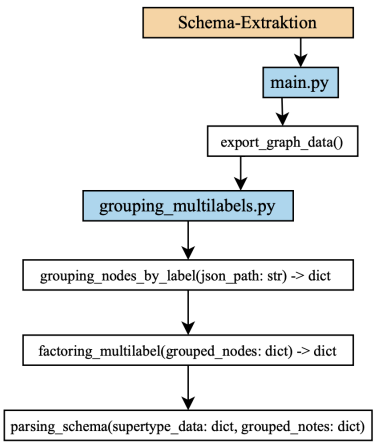



Abbildung 6: Ablauf der Schema-Extraktion

Wie in Abbildung 6 dargestellt, beginnt der Ablauf in *main.py*, wo die Funktion

*export_graph_data* aufgerufen wird. Diese übergibt das exportierte JSON-Schema an die Moduldatei *grouping_multilabels.py*, in der nacheinander die Funktionen *grouping_nodes_by_label*, *factoring_multilabel* und *parsing_schema* ausgeführt werden. Auf diese Weise entsteht ein strukturiertes Schema der Graphdatenbank, das sich für die relationale Weiterverarbeitung eignet.

### 7.3.1. Extraktion der Rohdaten

Um die Daten aus der Graphdatenbank zu extrahieren, wird eine Verbindung zur Datenbank hergestellt. Die Funktion *export_graph_data* greift auf zentrale Methoden der Klasse *Neo4jDatabase* zurück. Zunächst erzeugt der Konstruktor eine neue Instanz der Klasse und übergibt dieser alle notwendigen Verbindungsparameter aus der Config-Datei. Nachdem die Verbindung erfolgreich hergestellt wurde, erfolgt die Extraktion aller Knoten und ihrer Beziehungen. Dies geschieht durch eine Cypher-Anfrage, die sämtliche Knoten sowie deren eingehende und ausgehende Kanten selektiert. Für die Ausführung der Cypher-Anfrage nutzt die Funktion die Methode *run_query*:

Codebeispiel 1: Cypher-Befehl zur Extraktion aller Knoten und Kanten

```
1 MATCH (n)
2 OPTIONAL MATCH (n)-[r]->(m)
3 RETURN n, r, m
```

Diese Abfrage gibt alle Startknoten $n$, Beziehungen $r$ und Zielknoten $m$ zurück. Da es vorkommen kann, dass ein Startknoten mehrere Endknoten besitzt und somit der gleiche Knoten mehrfach im Ergebnis der Cypheranfrage enthalten ist, muss sichergestellt werden, dass jeder Knoten nur einmal gespeichert wird. Um doppelte Einträge zu vermeiden, erfolgt die Speicherung der Knoten in einem Dictionary. Dabei wird geprüft, ob ein Knoten bereits erfasst wurde. Falls nicht, wird er neu hinzugefügt und mit seinen Eigenschaften gespeichert. Dies erfolgt ebenfalls mit allen Beziehungen, welche jedoch separat in einer weiteren Liste gespeichert werden. Um sicherzustellen, dass auch Knoten ohne Label oder ohne eingehende beziehungsweise ausgehende Kanten berücksichtigt werden, nutzt die Cypher-Anfrage

eine Kombination aus *MATCH* und *OPTINAL MATCH*. Auf diese Weise erfolgt zunächst das Matching aller Knoten, unabhängig davon, ob sie ein Label oder Beziehungen besitzen. Anschließend werden alle Knoten identifiziert, die über ein Label und Beziehungen verfügen. Somit entsteht ein vollständiges Abbild der Graphdatenbank, das alle Knoten und Kanten beinhaltet. Dabei besitzt jeder Knoten eine eindeutige ID, ein Label, falls vorhanden, und deren Eigenschaften. Die Beziehungen werden separat gespeichert, wobei der Typ der Beziehung, die Start- und Ziel-ID, das Ziellabel sowie optionale Eigenschaften der Beziehung gespeichert werden. Nachdem alle Knoten und Kanten extrahiert wurden, wird das gesamte Schema in einer JSON-Datei gespeichert, die als Grundlage für die nächste Verarbeitungsstufe dient.

### 7.3.2. Gruppierung der Knoten nach Labels

Nach der Extraktion der Rohdaten erfolgt die Gruppierung der Knoten basierend auf ihren Labels. Da Neo4j Knoten mit mehreren Labels erlaubt, werden alle Knoten mit derselben Label-Kombination zu einer gemeinsamen Entität zusammengefasst. Die Gruppierung erfolgt durch einen iterativen Prozess, bei dem geprüft wird, welche Labels einem Knoten zugewiesen sind. Anschließend werden die Knoten in einem globalen Dictionary gespeichert, wobei die Labels als Schlüssel dienen und die zugehörigen Knoten als Werte hinterlegt werden. Knoten ohne Label sind der Gruppe *unlabeled* zugeordnet. Zusätzlich zur Gruppierung der Knoten werden die Beziehungen entsprechend ihren Startknoten zugeordnet. Hierbei wird überprüft, ob die start_node-Eigenschaft einer Beziehung mit der ID eines bereits erfassten Knotens übereinstimmt. Ist dies der Fall, wird die Beziehung dem Knoten hinzugefügt. Außerdem wird überprüft, ob die Beziehung eine ID in den Eigenschaften enthält. Eine solche ID muss für das Mapping existieren. Ist das nicht der Fall, wird die generierte ID aus der Neo4j-Umgebung als ID eingefügt. Nachdem alle Beziehungen den jeweiligen Knoten zugewiesen wurden, liegen nun alle Knoten mit ihren Kanten gruppiert nach ihren Labels vor (Codebeispiel 2). Im nächsten Schritt wird das Dictionary mit allen gespeicherten Daten benötigt, um alle Multilabel-Knoten

zu verarbeiten und Supertypen zu identifizieren.

Codebeispiel 2: Allgemeine Struktur der gruppierten Knoten mit ihren Kanteninformationen in JSON

```
{ "Label1, Label2": [{
        "id": ID_WERT,
        "properties": {
            "Eigenschaft_1": WERT_1,
            "Eigenschaft_2": WERT_2,
                (...)},
    "edges": [{
        "type": "BEZIEHUNGSTYP",
        "target_node": ZIEL_ID,
        "target_label": [
            "ZIEL_LABEL"],
    "properties": {
        "Beziehungs_Eigenschaft_1": WERT_1,
        "Beziehungs_Eigenschaft_2": WERT_2 }}]}]}
```

### 7.3.3. Verarbeitung von Multilabel-Knoten

Die Verarbeitung der Multilabel-Knoten erfolgt in mehreren aufeinander abgestimmten Schritten. Ziel ist, die Struktur der Daten zu vereinfachen und wiederkehrende Eigenschaften, die sich aus der Kombination mehrerer Labels ergeben, zentral in sogenannten Supertypen zusammenzufassen.

Im ersten Schritt werden alle Multilabel-Knoten identifiziert. Dafür iteriert die Funktion *factoring_multilabels* über alle gruppierten Knoten und überprüft dabei die Länge der Labels. Ist dieser Ausdruck `len(label\_set)> 1` wahr , liegt ein Multilabel-Knoten vor und wird in einem separaten Set gespeichert. Somit liegen alle Knoten mit mehr als einem Label in einem Set vor und können nun nach der Häufigkeit ihrer einzelnen Labels überprüft werden. Hierfür werden alle vorkommenden Labels in einem Multilabel-Knoten identifiziert, um dann zu überprüfen, in welchen Multilabel-Kombinationen die einzelnen Labels noch existieren. Somit erhält man nach diesem Schritt ein Set, das für jedes Singlelabel die Menge aller Multilabel-Kombinationen enthält, in denen dieses Label vorkommt. Mit diesem Set

lassen sich sogenannte *Supertype-Kandidaten* definieren (Frozza et al. (2020)). Dabei handelt es sich um Labels, die in mehreren Multilabel-Kombinationen vorkommen. Diese Labels werden als Supertype-Kandidaten gespeichert. Jedoch ist das alleinige Vorkommen in mehreren Kombinationen noch kein hinreichendes Kriterium, um einen Supertype zu definieren.

Damit ein Label endgültig als Supertype gelten kann, muss es eigene, charakteristische Eigenschaften besitzen, die es klar von den anderen Labels innerhalb der Kombinationen abgrenzen. Diese Eigenschaften dürfen nicht bereits durch die zugehörigen Singellabel-Knoten abgedeckt werden. Um nun final ein Label als Supertype klassifizieren zu können, müssen die Eigenschaften der jeweiligen Kandidaten überprüft werden.

Dazu werden im nächsten Schritt alle Property-Keys (Definition 3.2.6) pro Label-Kombination gesammelt. Dabei wird für alle gruppierten Knoten analysiert, welche Property-Keys in den jeweiligen Knoten vorkommen. Diese Property-Keys werden mit ihren jeweiligen Label-Kombinationen im Dictionary *label_properties* gespeichert. Dabei werden nicht nur Multilabels, sondern auch Singlelabels betrachtet. Da nun im Dictionary *label_properties* sowohl Multilabels als auch Singlelabels gespeichert sind, müssen diese in zwei Gruppen aufgeteilt werden:

- **Single-Label-Sets:** Enthalten genau ein Label und dienen als Referenz für generelle Eigenschaften.
- **Multi-Label-Sets:** Bestehend aus mehreren Labels und beinhalten potenzielle übergeordnete Eigenschaften.

Diese Gruppierung wird erreicht, indem die Länge des jeweiligen Label-Sets überprüft wird. Enthält ein Label-Set genau ein Label, liegt ein Single-Label vor und wird im Dictionary *single_label_properties* gemeinsam mit seinen Property-Keys gespeichert. Label-Sets mit mehr als einem Label werden als Multilabels interpretiert und mit ihren Property-Keys im Dictionary *multilabel_properties* abgelegt.

Im darauffolgenden Schritt erfolgt eine Bereinigung der Multilabel-Eigenschaften. Dies überprüft, welche Eigenschaften der Multilabel-Sets bereits in den zugehörigen Singlelabels enthalten sind. Diese werden aus den Multilabel-Eigenschaften

entfernt, da sie nicht charakteristisch für einen Supertype sind. Somit liegen alle Property-Keys vor, die exklusiv in den Multilabel-Sets vorkommen, in denen ein bestimmter Supertype-Kandidat enthalten ist und nicht in den zugehörigen Single-Labels erscheinen. Im nächsten Schritt wird für jeden Supertype geprüft, ob es Eigenschaften gibt, die in allen Multilabel-Kombinationen, in denen dieser Kandidat existiert, gemeinsam enthalten sind. Dies erfolgt über die Schnittmenge der bereinigten Eigenschaftsmengen aller relevanten Multilabel-Sets. Diese Schnittmenge stellt die finale Eigenschaftsmenge der Supertypen dar (Frozza et al. (2020)).

Diese Logik lässt sich formal in folgender Formel ausdrücken:

$$\text{Properties}(\boldsymbol{S}) = \bigcap_{\boldsymbol{L} \in \boldsymbol{\mathcal{L}_S}} \left( \boldsymbol{P}(\boldsymbol{L}) \setminus \bigcup_{\boldsymbol{l} \in \boldsymbol{L}} \boldsymbol{P}_{\text{single}}(\boldsymbol{l}) \right) \tag{7.1}$$

Dabei bezeichnet:

- $S$ einen Supertype-Kandidaten,
- $\mathcal{L}_S$ die Menge aller Multilabel-Sets, die das Label $S$ enthalten,
- $P(L)$ die Menge der Property-Keys des Multilabel-Sets $L$,
- $P_{\text{single}}(l)$ die Menge der Property-Keys des Singlelabels $l$.

Somit sind Eigenschaften eines Supertypes diejenigen, die in allen Multilabel-Kombinationen vorkommen, in denen ein Supertype-Kandidat $S$ enthalten ist, aber nicht bereits durch die jeweiligen einzelnen Labels dieser Kombinationen abgedeckt werden. Dadurch werden nur jene Eigenschaften dem Supertype zugeordnet, die tatsächlich übergreifend und exklusiv für die Multilabel-Kombinationen mit $S$ charakteristisch sind. Die Eigenschaftsschnittmenge wird anschließend im Dictionary *supertype_properties* gespeichert. Dabei wird der Supertype als Key und die Property-Keys als Value gespeichert.

Im letzten Schritt werden zu diesen Eigenschaften die zugehörigen Daten aus den Originalknoten extrahiert. Dafür wird für jeden Supertype ein Dictionary erstellt und dort für jede Supertype-Eigenschaft eine Liste initialisiert. Anschließend wird über alle Originalknoten iteriert, deren Label-Set den aktuellen Supertype enthält. Für jeden dieser Knoten wird überprüft, ob er die jeweilige Eigenschaft des Super-

typens besitzt. Ist dies der Fall, werden Informationen wie

- die eindeutige ID des Knotens,
- der konkrete Eigenschaftswert (Definition 3.1.3),
- die zugehörigen Kanten

extrahiert und in die entsprechende Liste des Supertyps zugeordnet. Somit liegen am Ende alle Supertypen mit ihren eindeutigen Eigenschaften in einem Dictionary vor und können im nächsten Schritt in das Schema integriert werden.

### 7.3.4. Schema-Zusammenführung und Integration der Supertypes

Im letzten Schritt der Schema-Extraktion werden alle gruppierten Knoten und die jeweiligen Supertypen in eine finale, einheitliche Struktur überführt. Ziel ist es, ein Schema zu erzeugen, in dem alle Multilabels aufgelöst und in die bestehenden Singlelabels überführt, redundante Eigenschaften entfernt und Supertypen korrekt integriert werden. Dies erfolgt in der Funktion *parsing_schema*. Diese Funktion erhält als Parameter die gruppierten Knoten (*grouped_nodes*) sowie die Supertypedaten (*supertype_data*) und gibt das extrahierte Schema als JSON-Datei zurück. Zunächst wird die Struktur der Supertypes erzeugt. Diese Struktur ist sowohl für mehrere Supertypen als auch für einen Supertypen identisch. Dabei wird eine Variable *supertype_json* angelegt, in der alle Supertypen gespeichert werden. Diese Struktur ist in Codebeispiel 3 zu sehen.

Codebeispiel 3: Supertype-Struktur im extrahierten Schema in JSON

```
"Supertypes": [{
    "Supertype": "label_x",
    "data": [{
        "id":0,
        "properties":{
            "prop_1": "",
            "prop_x": ""},
        "edges": [{
            "type": rel_type,
            "target_node": 0,
            ...}}]}]}
```

```
12          {"Supertype": "label_z",
13            "data":[...]}]
```

Da *supertype_data* nicht nur das Label, sondern auch alle Eigenschaften des Supertypens enthält, wird über dieses Dictionary iteriert und alle Supertypen identifiziert. Zudem werden alle Eigenschaften in ein weiteres Dictionary *data_list* gespeichert. Nachdem alle Supertypen und alle Daten gesammelt wurden, müssen diese noch in die Struktur eingefügt werden. Dabei wird dem Wert unter dem Schlüssel *Supertypes* in dem Dictionary *supertype_json* ein neuer Eintrag hinzugefügt. Dieser Eintrag besteht aus den beiden Schlüsseln *Supertype*, dem der Wert der Variable *supertype* zugewiesen wird und *data*, dem alle Eigenschaften aus der Variable *data_list* zugewiesen werden (Codebeispiel 4).

Codebeispiel 4: Befüllen der Supertype-Struktur

```
1  supertype_json["Supertypes"].append({
2      "Supertype": supertype,
3      "data": data_list})
```

Im nächsten Schritt werden alle Multilabel-Knoten in ihre Einzelbestandteile aufgespalten und dabei die Supertype-Eigenschaften aus den Knoten entfernt. Hierfür wird über alle gruppierten Knoten (*grouped_nodes*) iteriert und für jede Gruppierung geprüft, ob eines der enthaltenen Labels als Supertype klassifiziert wurde. Diese Überprüfung erfolgt über die Schnittmenge der Labels mit den zuvor identifizierten *supertype_labels*. Ist dies der Fall, bedeutet das, dass die Eigenschaften des Supertypes aus diesem Knoten entfernt werden müssen. Nachdem alle Supertype-Eigenschaften entfernt wurden, wird ein neues Knotenobjekt erstellt, das die *id* und die bereinigten Eigenschaften enthält. Gleichzeitig wird das Feld *belongs_to* ergänzt, welches angibt, zu welchen Supertypen der Knoten gehört. Dieser Knoten wird dem Dictionary *output_json* übergeben, in dem alle Knoten nach ihren Labels sortiert gespeichert werden. Besitzt eine Gruppe von Knoten keinen Supertype als Label, müssen keine Eigenschaften entfernt werden. Abschließend wird das Dictionary, das alle Supertypes und deren zugehörigen Daten enthält, in das finale Dictionary eingefügt. Dies geschieht durch die Methode *update*(), die das *supertype_dict* in das

*output_dict* integriert und in einer JSON-Datei speichert.

Somit liegt das Schema der Graphdatenbank gruppiert in einer JSON-Datei vor, welche im darauffolgenden Schritt für das Schema-Mapping benötigt wird.

## 7.4. Schema Mapping (Schritt 4+5)

Nachdem das Schema der Graphdatenbank im JSON-Format vorliegt, erfolgt die Umsetzung des Mappings in eine relationale Datenbank.

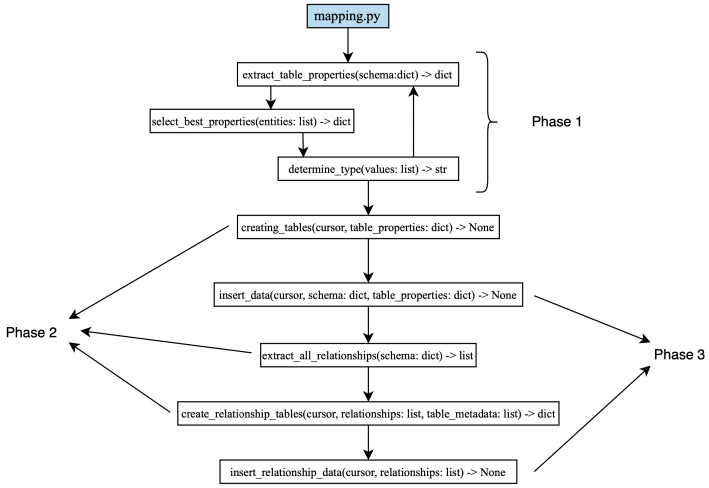



Abbildung 7: Ablauf des Schema-Mappings

Dieser Vorgang gliedert sich in drei zentrale Phasen, wie in Abbildung 7 dargestellt:

- **Extraktion der Entitäts- und Beziehungsdaten:** Dabei werden zunächst alle relevanten Knoten- und Kanteninformationen mit den Funktionen *extract_table_properties* und *extract_all_relationships* erfasst und analysiert (Phase 1).
- **Erstellung der Entitäts- und Beziehungstabellen:** Auf Basis der extrahierten Daten erstellen die Funktionen *creating_tables* und *creating_relationship_tables* alle Entitäts- und Beziehungstabellen (Phase 2).
- **Befüllung der Tabellen:** Abschließend befüllen die Insert-Funktion die Tabellen mit den zuvor analysierten Knoten- und Kanteninstanzen (Phase 3).

### 7.4.1. Erstellung und Verbindung der Ziel-Datenbank

Zunächst wird eine neue relationale Datenbank erstellt, die als Ziel für das Mapping dient. Dies erfolgt über die SQLDatabase-Klasse. Beim Aufruf wird eine neue Instanz der Klasse mit allen Verbindungsparametern erstellt. Mit der Methode *create_database* lässt sich eine neue Datenbank anlegen. Danach ermöglicht die Methode *db_connection* den Aufbau einer aktiven Verbindung zur erstellten Datenbank, die das Cursor-Objekt zurückgibt, um SQL-Anfragen auf der Datenbank ausführen zu können.

### 7.4.2. Extraktion der Tabellenstrukturen

Nach der Erstellung der Zieldatenbank erfolgt eine Analyse der gruppierten Knoten, um eine entsprechende tabellarische Struktur für die relationale Datenbank zu generieren. Dieser Prozess beginnt mit der Bestimmung der Tabellen und ihrer Spalten, basierend auf den extrahierten Knoten und ihren Eigenschaften.

Zunächst werden die Tabellenstrukturen aus dem extrahierten JSON-Schema ermittelt. Dabei bilden die Labels der jeweiligen Knoten Entitätstabellen in der relationalen Datenbank. Die Extraktion dieser Eigenschaften erfolgt über die Funktion *extract_table_properties*, die das JSON-Schema als Parameter erhält und die Tabellen mit ihren zugehörigen Spalten sowie den jeweiligen Datentypen in Form eines Dictionaries zurückgibt. Außerdem gibt die Funktion alle Werte aus dem Schema strukturiert als Dictionary *table_data* zurück. Dafür fängt das Dictionary *table_properties* alle Labels aus dem Schema auf und speichert diese als Key ab.

Neben den regulären Labels fließen auch die Supertypen in das Mapping ein. Diese gelten als eigenständige semantische Einheit und gehen ebenfalls als Entitätstabellen in die relationale Struktur über. Dadurch, dass sich die Struktur des JSON-Abschnitts der Supertypen im Vergleich zu den Singlelabels unterscheidet, werden diese separat analysiert. Die Identifikation der Spalten und deren Datentyp erfolgt jedoch analog zu den regulären Labels.

Anschließend erfolgt die Identifikation der konkreten Spalten und ihrer Datentypen für jede zu erstellende Tabelle. Hierzu analysiert die Funktion *select_best_pro-*

*perties* sämtliche Entitäten eines Labels, aggregiert deren Eigenschaften und ermittelt für jedes Attribut anhand der vorhandenen Werte den geeignetsten Datentyp. Dabei wird sichergestellt, dass sämtliche in den Entitäten vorkommenden Attribute als Spalten in der resultierenden Tabelle abgebildet werden, unabhängig davon, ob sie in jeder Entität vorhanden sind. Enthält beispielsweise ein Knoten die Eigenschaften *name*, *type* und *gdp*, während andere Knoten desselben Labels nur *name* und *type* besitzen, wird dennoch für alle drei Attribute eine eigene Spalte definiert. Auf diese Weise entsteht eine vollständige und konsistente tabellarische Struktur, die alle potenziellen Eigenschaften einer Entitätsklasse berücksichtigt. Die Typisierung erfolgt über die Funktion *determine_type*, die für den jeweiligen Wert den Datentyp zurückgibt. Da es in Neo4j keine strikte Typprüfung gibt, müssen alle Typen (Definition 3.2.9) der Eigenschaften bestimmt und vereinheitlicht werden. Da ein Attribut in verschiedenen Knoten unterschiedliche Datentypen aufweisen kann, wird der allgemeinste Typ gewählt, um alle möglichen Werte abzudecken.

Bei der Typbestimmung werden folgende Regeln angewendet:

- Falls alle Werte eines Attributs *True* oder *False* sind oder nur die Zahlen *0* und *1* enthalten, wird das Attribtut als *boolean* deklariert.
- Falls ein Attribut sowohl *0* und *1* als auch größere Zahlen enthält, wird es als *INT* klassifiziert.
- Falls alle Werte Integer-Zahlen sind, wird überprüft, ob sie den Standard-Integer-Bereich überschreiten. Ist dies bei einem Wert der Fall, wird der Typ des Attibut als *BIGINT* bestimmt.
- Falls sowohl Ganzzahlen als auch Gleitkommazahlen vorhanden sind, wird das Attribut als $DECIMAL(10, 2)$ deklariert.
- Falls ausschließlich Zeichenketten vorliegen, wird $VARCHAR(255)$ als Datentyp gewählt.
- Falls unterschiedliche Datentyp innerhalb eines Attributs vorkommen, wird das Attribut als *TEXT* gespeichert

Durch diese Typisierungsstrategie wird sichergestellt, dass alle Werte korrekt interpretiert werden und keine Konflikte zwischen Datenformaten entstehen. Somit

liegen alle Knoten mit ihren Property-Keys und dem Datentyp in einem Dictionary vor.

Auf diese Weise lassen sich sämtliche Entitätstabellen mit ihrer Spaltenstruktur und den zugehörigen Datentypen ableiten. Dabei fließen sowohl Informationen aus den Singlelabel-Koten als auch aus den als Supertypen klassifizierten Strukturen in die tabellarische Modellierung ein. Die Tabelle 1 zeigt die identifizierten Entitäten mit ihren zugehörigen Attributen sowie den bestimmten Datentypen. Diese Informationen bilden die Grundlage für das Erstellen der Entitätstabellen.

| **Tabelle (Label)** | **Spalte** | **Datentyp** |
|---|---|---|
| City | id | INT |
| | name | VARCHAR(255) |
| | plz | INT |
| State | id | INT |
| | name | VARCHAR(255) |
| Country | id | INT |
| | name | VARCHAR(255) |
| EconomicHub | id | INT |
| | gdp | BIGINT |

Tabelle 1: Tabellenstruktur basierend auf dem extrahierten JSON-Schema

### 7.4.3. Erstellung der Entitätstabellen

Nachdem die Tabellenstrukturen der Entitätstabellen extrahiert wurden, erfolgt die Erstellung der Entitätstabellen in der relationalen Datenbank. Dies geschieht über die Funktion *creating_tables*, die als Parameter das zuvor erstellte table_properties-Dictionary und den Datenbank-Cursor erhält, um für jede Tabelle die entsprechenden Spalten und Datentypen festzulegen. Für jede Spalte wird der zuvor definierte Typ als Spaltentyp übernommen. Somit liegt für jede Spalte der passende SQL-Typ in der Variable *sql_type* vor. Der Spaltenname und der SQL-Typ werden im Anschluss in der Liste *columns* gespeichert. Außerdem identifiziert die Funktion alle Spaltennamen, die mit *id* beginnen, als potenzielle Primärschlüssel (Definition 3.1.10) und speichert diese in einer Liste ab. Liegt mindestens eine ID-Spalte

in der Liste vor, erfolgt das Erstellen eines SQL-Ausdrucks zur Definition des Primärschlüssels. Dieser Ausdruck lautet *PRIMARY KEY* (...) und enthält alle zuvor identifizierten ID-Spalten. Anschließend wird der Ausdruck zur Spaltenliste der Tabellen hinzugefügt.

Um nun die Tabelle auf der Datenbank erstellen zu können, wird eine CREATE-Table-Anweisung generiert. Hierfür erhält diese den Tabellennamen aus dem Dictionary und alle Spalteneigenschaften aus der Liste *columns*. Mit diesem Ausdruck erfolgt das Erstellen einer Entitätstabelle in der Zieldatenbank, wenn diese noch nicht in der Datenbank existiert (Codebeispiel 5).

Codebeispiel 5: Allgemeine SQL-Anfrage zur Erstellung der Entitätstabellen

```
1 CREATE TABLE IF NOT EXISTS {table_name} (
2                 {', '.join(columns)});
```

Somit wurden alle Entitätstabellen erstellt, welche im Anschluss befüllt werden.

### 7.4.4. Befüllung der Entitätstabellen

Nach der Erstellung der Tabellen erfolgt das Übertragen der Daten aus der Graphdatenbank in das relationale Schema. Dieser Prozess erfolgt über die Funktion *insert_data*, die als Parameter die strukturierte Datenbasis *table_data*, die zugehörigen Tabelleneigenschaften *table_properties* sowie den aktiven Datenbank-Cursor entgegennimmt.

Die Funktion iteriert zunächst über alle Einträge in table_data, wobei jeder Eintrag einer zu befüllenden Entitätstabelle entspricht. Für jede enthaltene Entität wird überprüft, ob ein ID-Wert vorliegt, welcher als Primärschlüssel in die Liste der Spalten *columns* und Werte *values* aufgenommen wird. Anschließend erfolgt die Extraktion aller weiteren Attribute aus dem Property-Dictionary der Entität. Dabei werden alle Attribute einer Entität in der Spalten-Liste und deren Werte in der Liste *values* gespeichert. Hierbei findet für den Datentyp boolean eine erneute Typenprüfung statt. Ist der Datentyp einer Spalte *boolean* definiert, werden numerische Werte wie 0 und 1 vor dem Einfügen in den entsprechenden booleschen Wert True bzw. False überführt. Somit wird verhindert, dass diese Werte als Ganzzahlen interpretiert

werden, was andernfalls zu Fehlern bei der Datenvalidierung führen könnte.

Nach der Zusammenstellung der Spaltennamen und Werte wird ein dynamischer SQL-INSERT-Befehl basierend auf dem Tabellennamen sowie den beiden Listen *columns* und *values* generiert. Ein Validierungsschritt stellt zuvor sicher, dass jede Entität über einen gültigen Wert für den jeweiligen Primärschlüssel verfügt. Weist eine Entität im Vergleich zur zugehörigen Tabellendefinition jedoch weniger Eigenschaften auf, so bleiben die fehlenden Attribute im SQL-Befehl unberücksichtigt. In diesem Fall wird für die entsprechenden Spalten beim Einfügen der Wert *NULL* gesetzt. Auf diese Weise lässt sich eine robuste Datenübertragung gewährleisten, auch wenn nicht alle Entitäten vollständig ausgeprägte Attributmengen besitzen.

### 7.4.5. Extraktion der Beziehungstabellen

Im nächsten Schritt erfolgt die Extraktion aller Beziehungen aus der vorbereiteten Datenstruktur. Die Funktion *extract_all_relationships* übernimmt hierfür die zentrale Rolle. Sie erhält als Parameter das Dictionary *table_data*, das sämtliche Entitäten enthält.

Für jede Entität wird zunächst überprüft, ob sie eingehende oder ausgehende Kanten besitzt. Jede dieser Beziehungen wird analysiert und in einem standardisierten Format erfasst. Die relevanten Informationen umfassen das Start-Label, den Beziehungstyp, die Start- und Ziel-ID, das Ziel-Label sowie die Beziehungsattribute. Um die Beziehungen eindeutig zu identifizieren und Duplikate zu vermeiden, wird aus diesen Informationen ein Schlüssel aus dem jeweiligen Label, Beziehungstyp, Start- und Ziel-ID und Ziel-Label generiert, der in einem Set gespeichert wird. Existiert der Schlüssel bereits, wurde diese Beziehungsart bereits erfasst und überspringt diese Beziehung.

Außerdem prüft die Funktion, ob eine Entität einem Supertype zugeordnet ist. Besitzt eine Entität das Feld *belongs_to*, werden alle Kanten extrahiert, die im zugehörigen Supertype für dieselbe Entität definiert sind. Somit werden die Beziehungen eines Multilabel-Knotens in zwei verschiedene Beziehungstabellen aufgeteilt, behalten jedoch aufgrund der Beziehungstyp-ID ihre semantische Zuordnung bei.

Für jede Beziehung wird außerdem der Name der zugehörigen Beziehungstabelle dynamisch nach folgendem Muster erzeugt:

**beziehungstyp_startlabel_ziellabel** (7.2)

Durch die Kombination aus Beziehungstyp, Startlabel und Ziellabel entsteht für jeden Beziehungstyp eine eindeutige Tabellenbezeichnung, die sowohl die Struktur als auch die Richtung der Verbindung abbildet. Diese Namenskonvention erleichtert nicht nur die automatische Erstellung der Tabellen, sondern auch deren spätere Auswertung und Interpretation innerhalb der relationalen Datenbank. So lässt sich aus jeder Tabelle unmittelbar ablesen, um welche Beziehungsart es sich handelt und welche Entitäten dabei betroffen sind. Jeder Beziehung wird dieser Name hinzugefügt und landet daraufhin mit allen anderen Beziehungen in einer Liste, die als Grundlage für das Erstellen und Befüllen der Beziehungstabellen dient.

### 7.4.6. Erstellung der Beziehungstabellen

Nach der Extraktion aller Beziehungen aus der Graphstruktur erfolgt die Erstellung der entsprechenden Beziehungstabellen in der relationalen Zieldatenbank. Diesen Schritt übernimmt die Funktion *create_relationship_tables*, welche für jede identifizierte Beziehungstabelle die benötigte SQL-Anweisung generiert und ausführt. Als Grundlage dienen die zuvor ermittelten Metadaten jeder Beziehung.

Zunächst überprüft die Funktion, ob für eine Beziehung bereits die entsprechende Tabelle erstellt wurde. Existiert diese Tabelle noch nicht, erfolgt die Ermittlung der notwendigen Spalten und ihrer Datentypen. Dabei wird insbesondere nach einem Property-Attribut gesucht, das mit *id_* beginnt, um dieses als Primärschlüssel der Beziehungstabelle zu verwenden. Alle weiteren Attribute werden anhand ihres Datentyps analysiert und einem passenden SQL-Typ zugewiesen. Neben diesen Feldern besitzen alle Beziehungstabellen zusätzlich die Spalten *start-id* und *end-id*, die als Fremdschlüssel mit den zugehörigen Entitätstabellen verknüpft werden. Für jede Beziehungstabelle wird somit ein vollständiger SQL-Befehl zur Erstellung generiert, der die Definition aller Spalten, Fremdschlüssel und des Primärschlüssels

umfasst.

### 7.4.7. Befüllung der Beziehungstabellen

Nachdem alle nötigen Beziehungstabellen erstellt wurden, erfolgt im letzten Schritt das Befüllen der Beziehungstabellen mit den jeweiligen Beziehungen. Die Funktion *insert_relationship_data* liest die extrahierten Beziehungen ein und konstruiert für jede Kante den benötigten SQL-Befehl.

Zunächst wird überprüft, ob für jede extrahierte Beziehung eine entsprechende Beziehungstabelle erstellt wurde. Hierfür wird überprüft, ob ein Wert für *table* existiert. Ist dies nicht der Fall, wird diese Beziehung übersprungen und die nächste überprüft. Für gültige Beziehungen erfolgt zunächst das Auslesen aller relevanten Informationen wie die Start- und Ziel-ID sowie gegebenenfalls deren Eigenschaften, falls vorhanden. Für alle Eigenschaften einer Beziehung wird ein Dictionary angelegt, das alle Property-Keys als Key und alle Werte als Value speichert. Start- und Ziel-ID werden in der Variable *start_id* und *end_id* gespeichert. Um nun die Beziehungstabellen zu befüllen, erfolgt eine dynamische Generierung von SQL-Befehlen. Hierfür speichern die beiden Listen *columns* und *values* die jeweiligen Informationen, die für das Erstellen der INSERT-Befehle notwendig sind. Die Liste *columns* enthält alle Spaltennamen, beginnend mit Start- und Ziel-ID, gefolgt von den Attributnamen der Beziehungen, die dynamisch aus dem zugehörigen Property-Dictionary extrahiert werden. Entsprechend enthält die Liste *values* die dazugehörigen Werte, in identischer Reihenfolge (Codebeispiel 6). Mit diesen Spalten- und Werteinformationen erfolgt im nächsten Schritt die Erstellung der *INSERT*-Befehle. Zunächst wird über die Länge der Wertliste eine Anzahl an Platzhaltern generiert. Diese bestehen aus dem Zeichen `\%s`, das in SQL-Anfragen als Parameterersatz fungiert. Sie werden jeweils durch ein Komma getrennt und in einer Variable gespeichert. Die Konstruktion des Befehls erfolgt in der Variable *insert_sql*. Dieser enthält zunächst die Ziel-Tabelle sowie sämtliche Spaltennamen, die durch ein Komma voneinander getrennt sind. Die zugehörigen Werte repräsentieren die zuvor definierten Platzhalter.

Die eigentliche Ausführung der Befehle erfolgt anschließend über die Methode *execute* des SQL-Cursors. Diese Methode erhält zum einen den SQL-Befehl über die definierte Variable und die Liste *value*, die den Platz des Platzhalters einnehmen. Auf diese Weise lassen sich alle Beziehungstabellen mit den jeweiligen Beziehungen befüllen.

Codebeispiel 6: Konstruktion der Spalten- und Wertelisten

```
1  columns = ["start_id", "end_id"] + list(properties.keys())
2  values = [start_id, end_id] + list(properties.values())
```

Nach dem Befüllen der Beziehungstabellen wurde die Graphdatenbank erfolgreich auf eine relationale Datenbank transformiert. Im nächsten Schritt wird eine Schema-Evolution mithilfe von ProSA durchgeführt.

## 7.5. Schema-Evolution (Schritt 6)

Bei einer Schema-Evolution wird eine Änderung an der relationalen Datenbank vorgenommen. Um die beiden Datenbanken vergleichen zu können, wird diese Schema-Änderung synchron sowohl auf der relationalen Datenbank als auch auf der Graphdatenbank durchgeführt. Hierfür werden die Schemaoperatoren, die als Cypher-Queries vorliegen, mithilfe von regulären Ausdrücken (*REGEX*) in der Funktion *cypher_parser* automatisch in entsprechende SQL-Queries übersetzt und im nächsten Schritt beide Anfragen über die Funktion *execute_queries_for_schema_evolution* auf der jeweiligen Datenbank ausgeführt (Abbildung 8).

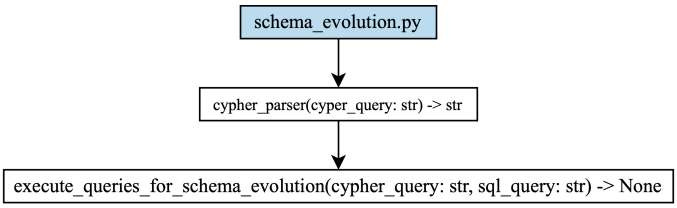



Abbildung 8: Ablauf der Schema-Evolution

### 7.5.1. Parsen der Cypher-Anfragen

Das Parsen der Cypher-Anfragen erfolgt über die Funktion *cypher_parser*. Als Parameter erhält die Funktion eine Cypher-Anfrage, die im Anschluss analysiert wird. Dazu wird der Cypher-String mithilfe von mehreren regulären Ausdrücken geprüft, die jeweils einem bestimmten Evolutions-Operator zugeordnet sind. Die Funktion durchläuft alle definierten Ausdrücke (*patterns*) und prüft, ob die Anfrage dem erwarteten Muster entspricht. Sobald ein passender Ausdruck gefunden wurde, werden die relevanten Parameter (*params*) aus der Cypher-Anfrage extrahiert. Diese Informationen wie Tabellenname, Spaltenname oder bestimmte Werte werden benötigt, um im nächsten Schritt eine analoge SQL-Anfrage generieren zu können, die dem ursprünglichen Evolutionsoperator entspricht (Codebeispiel 7).

Codebeispiel 7: Algorithmus zur Übersetzung von Cypher-Anfragen

```
1  Function cypher_parser(cypher_query)
2      patterns = {
3          "REMOVE": "Regex-Remove",
4          "DELETE": "Regex-Delete",
5          "RENAME": "Regex-Rename",
6          "ADD":    "Regex-Add",
7          "COPY":   "Regex-Copy"}
8      For op, regex_expr in patterns:
9          match = regex(regex_expr, cypher_query)
10         IF match:
11             params = match.fetch()
12             Return "SQL-QUERY + params"
```

Ein konkretes Beispiel ist folgende Cypher-Anweisung:

```
1      MATCH(c:City) REMOVE c.name
```

Diese Anfrage entfernt aus allen Knoten mit dem Label City die Eigenschaft *name*. Um diese Anweisung zu erkennen, wird folgender regulärer Ausdruck verwendet:

$$\textbf{MATCH\textbackslash s*\textbackslash(\textbackslash s*(\textbackslash w+):(\textbackslash w+)\textbackslash s*\textbackslash)\textbackslash s*REMOVE\textbackslash s+\textbackslash 1.(\textbackslash w+)} \tag{7.3}$$

Dieser Ausdruck kann im Detail wie folgt erklärt werden:

- **MATCH\s***: Erkennt das Schlüsselwort MATCH, gefolgt von beliebigen Whitespace (\s*)
- **\( und \)**: Die runden Klammern stehen für die Knotenstruktur in der Cypher-Anfrage. Der Wert in der Klammer beschreibt, welches Label betroffen ist.
- **(\w+)**: Hier wird die Variable erkannt, mit der die Knoten innerhalb der Abfrage bezeichnet werden.
- **":"**: Trennt die Variable vom Knoten.
- **(\w+)**: Erfasst das Label der Knoten.
- **\s* \)**: Schließt die Klammer nach dem Label und erlaubt ein Whitespace.
- **\s*REMOVE \s+**: Erkennt das Schlüsselwort REMOVE, erlaubt beliebige Leezeichen vor dem Schlüsselwort und mindestens eins danach.
- **\1 \.(\w+)**: Nutzt ein Rückverweis (\1) auf die Variable. Danach wird ein Punkt und weiter Teil erkannt. Dieser entspricht den Namen einer Spalte.

Dieser Ausdruck erkennt die Cypher-Anfrage als Remove-Operator. Damit die passende SQL-Anfrage definiert werden kann, müssen Parameter wie Tabellen- und Spaltenname identifiziert werden. Dies erfolgt mithilfe der Methode *group*(*n*), welche den Inhalt der n-ten Gruppe aus dem regulären Ausdruck zurückgibt. Somit wird mit `remove\_part.group(2)` und `remove\_part.group(3)` das Label *City* und die Spalte *name* zurückgegeben. Mit diesen Informationen wird die SQL-Anfrage generiert und als Rückgabewert der Funktion zurückgegeben:

Codebeispiel 8: Generierte SQL-Anfrage zur Änderung des Schemas

```
1    ALTER TABLE City DROP COLUMN name
```

Tabelle 2 veranschaulicht, wie sich die einzelnen Schema-Evolutionen in Cypher und SQL syntaktisch unterscheiden.

| Evolutionsoperator | SMO | Cypher-Anfrage | SQL-Anfrage |
|---|---|---|---|
| **Remove** | DROP COLUMN C | `MATCH (c:City) REMOVE c.name` | `ALTER TABLE City DROP COLUMN name;` |
| **Delete** | DROP TABLE R | `MATCH (c:City) DETACH DELETE c` | `DROP TABLE City CASCADE;` |
| **Rename** | RENAME COLUMN B IN R TO C | `MATCH (c:City) SET c.cityName = c.name REMOVE c.name` | `ALTER TABLE City RENAME COLUMN name TO cityName;` |
| **Add** | ADD COLUMN C AS const INTO R | `MATCH (c:Country {id:0}) SET c.population = 83000000` | `ALTER TABLE Country ADD COLUMN IF NOT EXISTS population INT; UPDATE Country SET population = 83000000 WHERE id = 0;` |
| **Copy** | COPY TABLE R | `MATCH (c:City) CREATE (n2:CopyCity))` | `COPY TABLE CopyCity AS SELECT * FROM City;` |

Tabelle 2: Vergleich von Cypher- und SQL-Anfragen

### 7.5.2. Durchführung der Schema-Evolution

Im nächsten Schritt werden beide Anfragen auf der jeweiligen Datenbank ausgeführt. Dies erfolgt in der Funktion *execute_queries_for_schema_evolution*, die *cypher_query* und *sql_query* als Parameter übergeben bekommt. Mithilfe der beiden Datenbank-Klassen wird eine Verbindung zu den jeweiligen Datenbanken aufgebaut und die Anfrage durchgeführt. Somit findet die Schema-Evolution auf beiden Datenbanken synchron statt.

## 7.6. Rücktransformation (Schritte 7-9)

Nachdem die Schema-Evolution durchgeführt wurde, erfolgt die Rücktransformation der relationalen Datenbank wieder in eine Graphdatenbank. Diese Rücktransformation stellt sicher, dass alle Knoten und Kanten wieder in ein Graphdatenbank-Schema überführt werden. Hierfür extrahiert die Funktion *extract_db_schema* das Schema aus der relationalen Datenbank und rekonstruiert alle Knoten und Kanten. Um alle Multilabel-Knoten zu identifizieren, kommt die Funktion *detect_multilabel* ins Spiel. Bevor das Schema zum Schluss in eine JSON-Datei gespeichert wird und die Methode *import_json_to_neo4j* aus der Neo4jDatabase-Klasse es an die Datenbank übermittelt, bereinigt die Funktion *clean_relationships* alle Beziehungen, die

aufgrund der Multilabels in mehreren Tabellen gespeichert wurden (Abbildung 9).

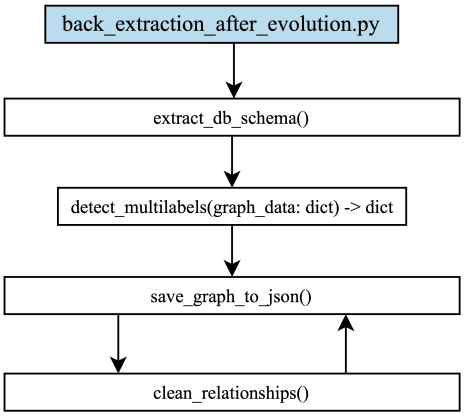



Abbildung 9: Ablauf der Rücktransformation von relationalem Schema in ein Graphschema

### 7.6.1. Extraktion des relationalen Schemas

Zunächst wird mithilfe von SQL-Befehlen auf die Struktur der relationalen Datenbank zugegriffen. Im ersten Schritt werden alle Beziehungstabellen aus der Datenbank extrahiert, die genau zwei Fremdschlüsselbeziehungen besitzen. Da jede Beziehungstabelle eine Start- und End-ID-Spalte besitzt, welche als Fremdschlüssel dienen, gibt die Anfrage alle Namen der Beziehungstabellen als Ergebnis zurück. Um nun alle Entitätstabellen zu erhalten, wird eine zweite SQL-Anfrage gestellt, die alle Tabellen, die in der Datenbank existieren, zurückgibt (Codebeispiel 9). Die Summe aller Tabellen wird mit allen Beziehungstabellen subtrahiert, um nur alle Entitätstabellen zu erhalten.

Codebeispiel 9: SQL-Anfrage, die alle existierenden Tabellen zurückgibt

```
1       SELECT table_name
2       FROM information_schema.tables
3       WHERE table_schema = 'public'
```

Mithilfe aller Beziehungs- und Entitätstabellen können alle notwendigen Informationen extrahiert werden. Damit alle Entitätstabellen in der Graphdatenbank wieder als Knoten dargestellt werden können, wird zunächst über all diese Tabellen iteriert. Für jede Tabelle wird eine SQL-Anfrage an die Datenbank gestellt, die alle

Tupel und Spaltennamen zurückgibt. Dabei wird für jedes Tupel ein Knotenobjekt erstellt, das alle nötigen Informationen enthält (Codebeispiel 10):

Codebeispiel 10: Struktur eines Knotens im JSON-Format

```
1 "node" ={
2     "id": ...
3     "properties": {...}
4     "edges": []}
```

Der Wert für das Attribut *id* wird dabei aus der entsprechenden Spalte des Tupels extrahiert. Die restlichen Spalten des Tupels werden im Feld *properties* gespeichert. Hierfür wird über alle Spaltennamen iteriert und jede Spalte gemeinsam mit dem zugehörigen Wert aus dem aktuellen Tupel als Key-Value-Paar in das Dictionary eingefügt. Da die ID-Spalte bereits überführt wurde, wird diese vernachlässigt.

Die *edges*-Liste wird in diesem Schritt nur angelegt und im späteren Verlauf mit den jeweiligen Beziehungen befüllt. Abschließend werden alle Knoten im Dictionary *graph_data* gespeichert.

Im Anschluss erfolgt die Verarbeitung der Beziehungstabellen. Da der Tabellenname bereits Informationen wie Beziehungstyp, Start- und End-Label besitzt, muss dieser in seine Bestandteile aufgeteilt werden. Diese Informationen werden benötigt, um die Beziehung dem zugehörigen Knoten zuweisen zu können. Analog zu den Entitätstabellen werden alle Einträge gelesen und die Spalten *start_id* und *end_id* als Referenzen für die verbundenen Knoten verwendet. Weitere Spalten werden als Eigenschaften der Beziehung übernommen. Um die Beziehung einem Knoten zuordnen zu können, wird geprüft, ob eine ID eines Knotens aus dem *graph_data*-Dictionary mit der *start_id* übereinstimmt. Ist dies der Fall, wird eine neue Kante mit dem extrahierten Beziehungstyp, dem Zielknoten und den zugehörigen Eigenschaften in die *edges*-Liste des Knotens eingefügt. Somit liegen alle Knoten mit ihren jeweiligen Kanten im Dictionary *graph_data* vor.

### 7.6.2. Erkennung und Rekonstruktion der Multilabels

Da Multilabels bei der Transformation auf das relationale Schema in einzelne Labels aufgeteilt wurden, müssen diese wieder zusammengeführt werden, um den ursprünglichen Zustand der Graphstruktur zu rekonstruieren. Hierfür dient die Funktion *detect_multilabels*. Diese Funktion analysiert die bereits extrahierten Knoten, die möglicherweise mehrfach unter verschiedenen Labels vorkommen.

Zu Beginn werden zwei Hilfsstrukturen initialisiert:

- **id_to_labels**: Dient der Zuordnung aller vorkommenden Labels zu einer bestimmten Knoten-ID.
- **id_to_nodes**: Speicherung aller Knoteneigenschaften und Beziehungen zu einer bestimmten Knoten-ID.

Anschließend iteriert der Code über alle Knoten in *graph_data* und überprüft, ob die ID des Knotens bereits in *id_to_labels* gespeichert wurde. Ist dies nicht der Fall, erfolgt eine erstmalige Eintragung der ID mit dem zugehörigen Label. Folgt nun ein Knoten mit einer bereits erfassten ID, handelt es sich um einen Multilabel-Knoten. In diesem Fall wird das neue Label zum bestehenden Label-Set ergänzt. Analog erfolgt die Aggregation der Eigenschaften und Beziehungen. Diese werden dem bereits gespeicherten Knoten mit derselben ID hinzugefügt, um alle Eigenschaften und Beziehungen beider Knoten zu vereinen. Somit liegen alle Label-Kombinationen mit ihrer Id in *id_to_labels* und alle Eigenschaften und Beziehungen in *id_to_nodes*. Im letzten Schritt entsteht das neue Datenmodell *merged_graph_data*. Für jede eindeutige Knoten-ID entsteht ein zusammengesetzter Label-String, indem alle zugehörigen Labels alphabetisch sortiert und durch Kommata getrennt sind. Dieser String gilt als Schlüssel in *merged_graph_data*. Das zugehörige Knotenobjekt wird der jeweiligen Label-Kombination mithilfe der ID zugeordnet.

### 7.6.3. Bereinigung der Beziehungen

Um Redundanzen innerhalb der Beziehungen eines Knotens zu vermeiden, erfolgt eine Bereinigung der Beziehungen. Dabei wird für jeden Knoten überprüft,

ob bereits eine Beziehung mit identischem Zielknoten, gleichem Beziehungstyp sowie identischer Beziehungs-ID existiert. Diese Prüfung ist notwendig, da durch die Aufteilung der Multilabel-Knoten beim Schema-Mapping Beziehungen mehrfach in verschiedenen Relationen gespeichert wurden.

Die Bereinigung erfolgt durch das Anlegen eines Sets, in dem jede eindeutige Beziehung als Tupel aus Ziel-ID, Beziehungstyp und Beziehungs-ID gespeichert wird. Nur wenn dieses Tupel noch nicht vorhanden ist, wird die Beziehung der finalen Kantenliste hinzugefügt. Dadurch bleiben unterschiedliche Kanten erhalten, während Duplikate entfernt werden.

### 7.6.4. Finalisierung der Rücktransformation

Nach der erfolgreichen Extraktion und Bereinigung der relationalen Daten wird die Graphstruktur in ein JSON-Format eingebettet. Dies erfolgt in der Funktion *save_graph_to_json*, die das Datenmodell in eine strukturierte Datei überführt.

Anschließend erfolgt die Rückführung in die Graphdatenbank. Dies geschieht durch die Methode *import_json_to_neo4j* aus der Neo4jDatabase-Klasse, welche die zuvor erzeugte JSON-Datei entgegennimmt und die enthaltenen Knoten sowie Beziehungen mit generierten Cypher-Anfragen in die Datenbank importiert. Zunächst erfolgt die Extraktion und Sortierung aller Knoten anhand ihrer ID. Nachdem werden für jeden Knoten alle Labels extrahiert und in der Liste *label_list* gespeichert. Enthält ein Knoten kein Label, wird eine leere Liste zurückgegeben (Codebeispiel 12, Zeile 1). Für jeden Knoten entsteht eine MERGE-Anweisung, die prüft, ob ein Knoten mit dieser ID bereits in der Graphdatenbank existiert. Dafür wird überprüft, ob in *label_list* Labels existieren. Ist dies der Fall, entsteht eine MERGE-Anweisung *merge_part*, in der alle vorhandenen Labels mithilfe von Doppelpunkten aneinandergereiht und dem Knoten zugewiesen werden. Dadurch kann ein mehrlabeliger Knoten in Cypher korrekt beschrieben werden (Codebeispiel 12, Zeile 7). Ist die Liste hingegen leer, wird eine MERGE-Anweisung ohne Label erzeugt, bei der lediglich die Knoten-ID eingefügt wird (Codebeispiel 12, Zeile 4-5). Zusätzlich werden alle Knoten-Eigenschaften extrahiert. Dafür wird dynamisch eine SET-Anweisung er-

zeugt, welche alle Attribute in der Form *n.key = $schlüssel* auflistet. Mithilfe einer Iteration über das properties-Dictionary (Codebeispiel 12, Zeile 3), dass alle Eigenschaften eines Knotens besitzt, wird für jede Eigenschaft ein entsprechender Zuweisungsausdruck generiert und anschließend zu einem vollständigen Ausdruck zusammengefasst und mit einem Komma separiert (Codebeispiel 12, Zeile 8-9). Somit liegen beide Anweisungen in folgender Form vor:

Codebeispiel 11: Generierte Cypher-Anweisung zur Erstellung eines Knotens und dessen Eigenschaften

```
1 MERGE (n:Label1:Label2:Label3 {id: $id})
2 SET n.key = $key, n.key1 = $key2, ...
```

Sowohl bei der Erstellung der Knoten als auch der Eigenschaften kommen Platzhalter zum Einsatz. Diese Platzhalter, etwa in Form von *$id* oder *$key*, dienen dazu, konkrete Werte erst zur Laufzeit über ein separates Parameter-Objekt zu übergeben. Die tatsächlichen Werte werden über das Dictionary *params* an die Methode *session.run(query, params)* übergeben. Dieses Dictionary enthält die Knoten-ID und die jeweiligen Eigenschaften, die zuvor aus den sortierten Knoten extrahiert wurden. Die Keys des Dictionaries entsprechen hierbei exakt den Platzhaltern in der Abfrage. Somit werden alle Cypher-Anfragen final zur Laufzeit generiert und ausgeführt (Codebeispiel 12).

Codebeispiel 12: Generierung von MERGE- und SET-Anweisungen zur Erstellung von Knoten mit Labels und Eigenschaften in Cypher

```
1  label_list = [label.capitalize() for label in labels.split(",")]
       if labels and labels != "Unlabeled" else []
2  node_id = node["id"]
3  properties = node.get("properties", {})
4      if labels == "unlabeled" or not label_list:
5          merge_part = "MERGE (n{id: $id})"
6      else:
7          merge_part = f"MERGE (n:{':'.join(label_list)} {{id: $id}})"
8      if properties:
9          properties_str = ", ".join(f"n.{key} = ${key}" for key in
               properties.keys())
```

Im Anschluss erfolgt die Verarbeitung der Kanteninformationen. Damit eine Beziehung dem jeweiligen Start- und Zielknoten zugeordnet werden kann, müssen beide Knoten mit einer MATCH-Anweisung über die eindeutige ID identifiziert werden. Zudem wird das Start- und Ziellabel benötigt. Das Start-Label ergibt sich dabei aus den sortierten Beziehungen, während das Ziel-Label direkt aus dem jeweiligen Knoten extrahiert wird. Ist dieses Label ein Multilabel, muss es zu einem String zusammengefasst und mit einem Doppelpunkt voneinander getrennt werden.

Nach erfolgreicher Referenzierung beider Knoten erstellt eine MERGE-Anweisung eine gerichtete Beziehung vom Start- zum Zielknoten. Besitzen Beziehungen Eigenschaften, ergänzt eine SET-Anweisung diese. Analog zu den Knoteneigenschaften muss aus allen Kanteneigenschaften ein String erstellt werden. Dieser Properties-String wird der SET-Anweisung hinzugefügt. Abschließend erfolgt die Extraktion aller erforderlichen Parameter, welche anschließend in die generierten Cypher- Anfragen eingebettet und an die Graphdatenbank übermittelt werden.

Somit ist die Rücktransformation abgeschlossen und die zuvor aus der relationalen Struktur extrahierten Informationen liegen nun wieder in einer Graphstruktur innerhalb der Neo4j-Datenbank vor. Auf Basis dieser vollständigen Transformation erfolgt nun eine Evaluation, um die Korrektheit, Konsistenz und Leistungsfähigkeit des entwickelten Verfahrens zu überprüfen.

# 8. Evaluation und Ergebnis

In diesem Kapitel wurde die Qualität und Korrektheit des entwickelten Transformations- und Evolutionsverfahrens evaluiert. Anhand konkreter Anfragen und systematischer Vergleiche wurde geprüft, ob die modellierten Schritte wie Schema-Extraktion, Mapping, Schema-Evolution (Kapitel 8.3) und Rücktransformation (Kapitel 8.2) erwartungsgemäß funktionieren. Zusätzlich erfolgte eine Analyse der Performance, um die Effizienz, Robustheit und Skalierbarkeit des gesamten Transformationsprozesses unter verschiedenen Datenbankgrößen und -strukturen zu bewerten.

## 8.1. Vergleich der Datenbanken nach dem Mapping

Nach der Transformation der Graphdatenbank in ein relationales Datenbankschema wurde überprüft, ob alle Informationen vollständig und korrekt übernommen wurden. Hierfür wurden identische Anfragen sowohl an die relationale Datenbank in SQL als auch an die ursprüngliche Graphdatenbank in Cypher gestellt. Ziel ist es, sicherzustellen, dass beide Datenbanken inhaltlich äquivalent sind, das heißt, dieselben Informationen in strukturierter Form enthalten. Dabei handelt es sich um Abfragen, die Verbundoperationen und Aggregationen beinhalten:

- **Erkennung von Multilabel-Knoten (Aggregation):** Diese Abfrage dient dazu, in der Graphdatenbank alle Knoten zu identifizieren, denen mehr als ein Label zugewiesen wurde. In der relationalen Datenbank entspricht dies dem Auftreten einer ID in mehreren Entitätstabellen. Somit kann überprüft werden, ob alle Multilabels beim Mapping-Prozess korrekt eingefügt wurden. Das Ergebnis besteht aus allen IDs, die mehrfach in mehreren Tabellen auftreten und den jeweiligen Tabellen. Die SQL- und Cypher-Anfragen aus Code-

beispiel 13 und Codebeispiel 14 liefern angewandt auf die durch das Mapping erzeugte relationale Datenbank sowie die Graphdatenbank jeweils:

| id | allLabels |
|---|---|
| 0 | [Country, Economichub] |
| 9 | [City, Economichub] |
| 28 | [City, Economichub] |

(a) Ergebnis der Neo4j-Abfrage

| id | tables |
|---|---|
| 9 | City,EconomicHub |
| 28 | City,EconomicHub |
| 0 | Country,EconomicHub |

(b) Ergebnis der PostgreSQL-Abfrage

Abbildung 10: Vergleich der Ergebnisse zur Multilabel-Erkennung zwischen Neo4j und PostgreSQL

Vergleicht man beide Anfrageergebnisse fällt auf, dass sie jeweils die gleiche Anzahl an Tupeln liefern. Trotz unterschiedlicher Darstellung -als Label-Liste in Neo4j und als kommagetrennter String in PostgreSQL - enthalten beide exakt dieselben Informationen zu den betroffenen Entitäten.

Codebeispiel 13: Multilabel-Erkennung (SQL)

```
1  SELECT id, string_agg(s_t, ','
       ) AS tables
2  FROM (
3       SELECT id, 'City' AS s_t
             FROM City
4       UNION ALL
5       SELECT id, 'State' AS s_t
             FROM State
6       UNION ALL
7       SELECT id, 'Country' AS
             s_t FROM Country
8       UNION ALL
9       SELECT id, 'EconomicHub'
             AS s_t FROM
             EconomicHub)
10      AS all_labels
11 GROUP BY id
12 HAVING COUNT(*) > 1;
```

Codebeispiel 14: Multilabel-Erkennung (Cypher)

```
1  MATCH (n)
2  WITH n.id AS nodeId, labels(n)
       AS allLabels
3  WHERE size(allLabels) > 1
4  RETURN nodeId, allLabels
```

- **Verbundanfrage auf Beziehungen:** Diese Anfrage prüft, welche Staaten (*State*) eine Hauptstadtbeziehung (*has*) mit einer Stadt (*City*) besitzen, bei der das Attribut *capital* den Wert *true* hat. Dabei wir getestet, ob relationale Fremdschlüssel- und Attributbeziehungen korrekt in Tabellen wie *has_State_to_City* abgebildet wurden. Als Ergebnis der jeweiligen Abfrage (Codebeispiel 15, Codebeispiel 16) erhält man jeweils die ID und den Staat der eine Hauptstadt besitzt:

| id | state |
|---|---|
| 18 | Bavaria |
| 24 | England |
| 21 | Jura |
| 19 | New York |

(a) Ergebnis der Verbundabfrage in Neo4j

| id | State |
|---|---|
| 18 | Bavaria |
| 24 | England |
| 21 | Jura |
| 19 | New York |

(b) Ergebnis der Verbundabfrage in PostgreSQL

Abbildung 11: Vergleich der Ergebnisse einer Verbundabfrage zwischen Neo4j und PostgreSQL

Beide Anfrageergebnisse liefern identische Tupeln hinsichtlich der Bundesstaaten, die über eine Hauptstadtbeziehung verfügen.

Codebeispiel 15: Ermittlung der Hauptstädte (SQL)

```
1 SELECT DISTINCT s.*
2 FROM State s
3 JOIN has_State_to_City h ON s.
      id = h.start_id
4 WHERE h.capital = TRUE;
```

Codebeispiel 16: Ermittlung der Hauptstädte (Cypher)

```
1 MATCH (s:State)-[h:has]->(c:
      City)
2 WHERE h.capital = true
3 RETURN DISTINCT s;
```

- **Komplexe Anfrage mit mehreren Joins und Bedingungen:** Die letzte Anfrage ermittelt alle Staaten, deren Hauptstadt eine Temperatur im Januar aufweist, die niedriger ist als die Temperatur von mindestens einer anderen Stadt im selben Staat, sofern diese andere Temperatur im Januar mehr als 2°C aufweist. Die Anfrage in Codebeispiel 17 und Codebeispiel 18 liefern als Ergebnis den

Namen des Bundesstaates, die zugehörige Hauptstadt sowie den Namen einer weiteren Stadt im Staat die im Januar wärmer als 2°C ist:

(a) Ergebnis in Neo4j

| state_name | capital_city | warm_city |
|---|---|---|
| Bavaria | Munich | Regensburg |
| Bavaria | Munich | Landshut |

(b) Ergebnis in PostgreSQL

| state_name | capital_city | warm_city |
|---|---|---|
| Bavaria | Munich | Landshut |
| Bavaria | Munich | Regensburg |

Abbildung 12: Vergleich der Join-Ergebnisse beider Datenbanken

Wie Abbildung 12 zeigt, liefern beide Abfragen inhaltlich identische Ergebnisse. Zwar unterscheidet sich die Reihenfolge der Tupel, jedoch sind die enthaltenen Informationen äquivalent.

Codebeispiel 17: Ermittlung aller Hauptstädte mit einer Temperaturmessung (SQL)

```sql
SELECT s.name AS state_name,
c1.name AS capital_city,
c2.name AS warm_city
FROM State s
JOIN has_State_to_City h1 ON s.id = h1.start_id
JOIN City c1 ON h1.end_id = c1.id AND h1.capital = TRUE
JOIN has_State_to_City h2 ON s.id = h2.start_id
JOIN City c2 ON h2.end_id = c2.id
JOIN data_City_to_City d ON d.start_id = c2.id AND d.month = 'January'
WHERE d.temp > 2.0 AND c1.id != c2.id;
```

Codebeispiel 18: Ermittlung aller Hauptstädte mit einer Temperaturmessung (Cypher)

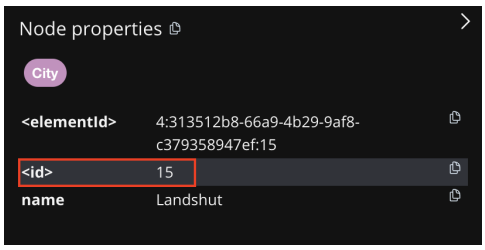
```cypher
MATCH (s:State)-[r1:has]->(c1:City)
WHERE r1.capital = true
MATCH (s)-[r2:has]->(c2:City)
WHERE c1.id <> c2.id
MATCH (c2)-[d:data]->(c2)
WHERE d.month = 'January' AND d.temp > 2.0
RETURN  s.name AS state_name,
c1.name AS capital_city,
c2.name AS warm_city
```

Insgesamt zeigen alle drei durchgeführten Abfragen, dass die inhaltliche Übereinstimmung zwischen der ursprünglich modellierten Graphdatenbank und der

gemappten relationalen Datenbank vollständig gegeben ist.

## 8.2. Vergleich der Graphdatenbanken nach der Rücktransformation

Anschließend wurde die gemappte relationale Datenbank wieder in eine Graphdatenbank zurücktransformiert und mit der ursprünglichen Graphdatenbank verglichen. Dabei wurden alle Knoten und Kanten beider Datenbanken extrahiert und miteinander verglichen. Die Analyse zeigte, dass alle semantischen Informationen wie Labels, Properties und Kanten inklusive Eigenschaften identisch sind. Beim Vergleich der Knotendaten fällt jedoch auf, dass die zurücktransformierte Graphdatenbank zusätzlich zur <*id*> eine Eigenschaft *ID* besitzt, welche in der ursprünglichen Datenbank nicht vorhanden ist (Abbildung 13).




(a) Knoten *Landshut* aus ursprünglicher Datenbank


(b) Knoten *Landshut* aus rücktransformierter Datenbank

Abbildung 13: Vergleich der Eigenschaften aus der ursprünglichen und rücktransformierten Datenbank

## 8.3. Vergleich der Datenbanken nach der Schema-Evolution

Um die Schema-Evolution evaluieren zu können, erfolgt ein Vergleich der relationalen Datenbank und der Graphdatenbank nach der Anwendung der jeweiligen Evolutionsoperatoren. Hierzu dient eine analoge Abfrage auf eine betroffene Tabelle bzw. einen betroffenen Knoten in beiden Datenbanken, also genau dort, wo eine strukturelle Änderung vorgenommen wird: Mit den Rückgabewerten dieser Abfragen kann bestimmt werden, ob die Änderungen konsistent umgesetzt werden und Strukturen sowie Inhalt der beiden Datenbanken übereinstimmen. In diesem Beispiel ändert der Evolutionsoperator *RENAME* den Property-Key *name* aller Städte zu *cityname*:

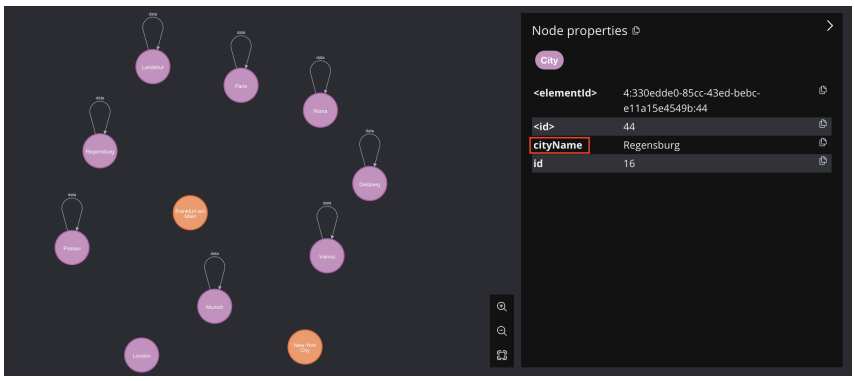



Abbildung 14: Anfrageergebnis nach der Schema-Evolution in Neo4j

| | cityname character varying (255) | id [PK] integer |
|---|---|---|
| 1 | Munich | 8 |
| 2 | Vienna | 10 |
| 3 | Delsberg | 11 |
| 4 | Roma | 12 |
| 5 | Paris | 13 |
| 6 | Landshut | 15 |
| 7 | Regensburg | 16 |
| 8 | Passau | 17 |
| 9 | London | 29 |
| 10 | New York City | 9 |
| 11 | Frankfurt am Main | 28 |

Abbildung 15: Anfrageergebnis nach der Schema-Evolution in PostgreSQL

Das Ergebnis beider Datenbankanfragen besteht jeweils aus 11 Städten, die die Attribute *id* und *cityname* vorweisen . Dabei wurde in der Graphdatenbank der Property-Key von *name* zu *cityname* umbenannt, während in der relationalen Datenbank die Spalte entsprechend angepasst wurde (Abbildung 14,Abbildung 15).

## 8.4. Vergleich der Datenbanken mit ProSA

Da nun die Graphdatenbank in einer relationalen Datenbank vorliegt, ist die Verwendung von ProSA gewährleistet. ProSA ermöglicht die Ausführung von Schema-Evolutionen basierend auf einer formalen Beschreibung der Evolutionsoperatoren und bietet darüber hinaus Funktionen wie die Analyse der Datenherkunft.

Im Rahmen der Evolution erfolgt zunächst eine Verifikation der Datenübernahme. Dazu stellte ProSA eine Verbindung zur gemappten PostgreSQL-Datenbank

her und führt eine einfache vordefinierte SQL-Anfrage aus, um zu überprüfen, ob der Datenimport korrekt und vollständig durchgeführt wurde.Analog erfolgte eine identische Anfrage auf Seiten der Graphdatenbank, um sicherzustellen, dass der semantische Inhalt beider Datenbanken vor der Schema-Evolution übereinstimmt und somit eine konsistente Ausgangsbasis für den anschließenden Schritt gegeben ist.

Die eigentliche Schema-Evolution erfolgte anschließend durch Übergabe eines Evolutionsoperators in Form einer XML-Datei (Codebeispiel 19). Diese Datei enthält alle relevanten Informationen zur Art und zum Umfang der geplanten Änderungen und wird von ProSA automatisiert verarbeitet. Als Beispiel wurde für die Evaluation der RENAME-COLUMN-Operator benutzt. Hierbei wurde bei allen City-Entitäten die Spalte *name* zu *cityName* unbenannt. Die zugrunde liegende SMO *Rename* wird ProSA-intern als logische Formel $X \rightarrow Y$ interpretiert, wobei $X$ eine Relation mit den ursprünglichen Attributen darstellt und $Y$ die neue Relation mit dem umbenannten Attribut. Aufgrund der Implementierung in ProSA wird dabei eine neue Tabelle angelegt, die alle ursprünglichen Werte aus *city* enthält, jedoch mit dem geänderten Spaltennamen.

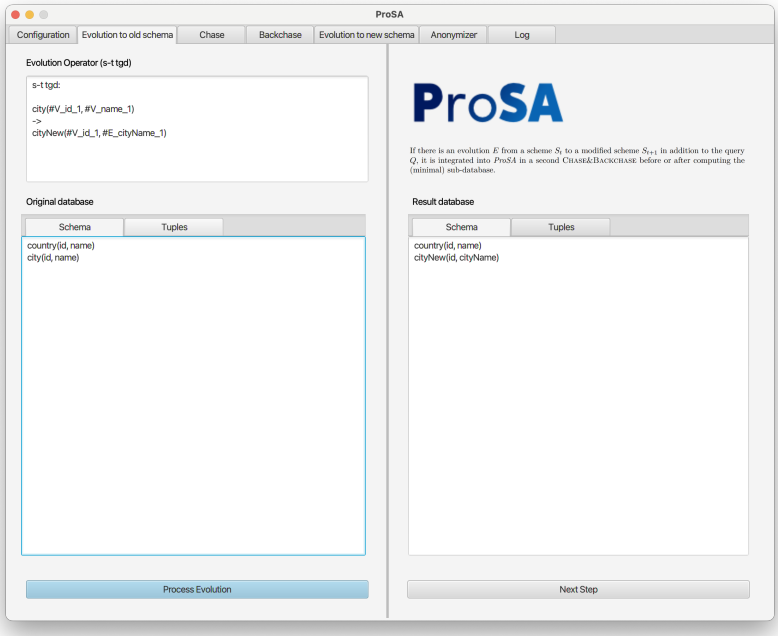
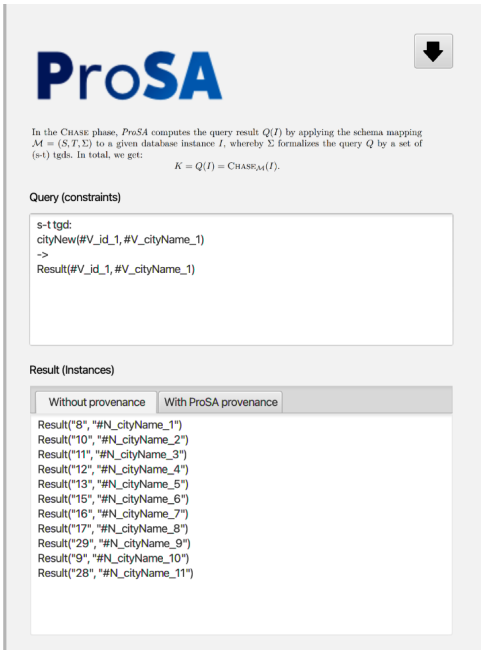
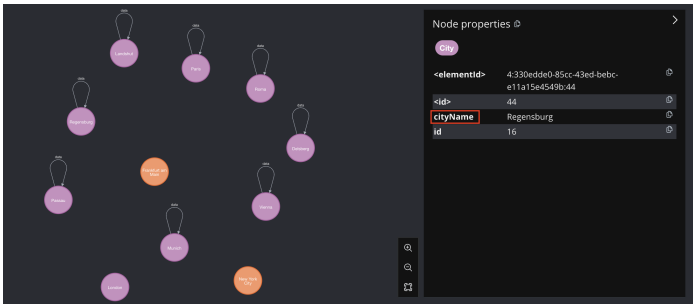



Abbildung 16: Durchführung der Schema-Evolution in ProSA

Da diese Evolution bereits auf der Graphdatenbank durchgeführt wurde, kann das Ergebnis von ProSA mit dem Zustand der Graphdatenbank verglichen werden. Dafür wird erneut eine Anfrage auf der Datenbank in ProSA und auf der Graphdatenbank ausgeführt. Die Anfrage `SELECT * FROM cityNew;` und `MATCH (n :City) RETURN n;` fragt alle Tupel aus der Tabelle *city* ab und vergleicht diese mit allen City-Knoten aus der Graphdatenbank:




(a) Anfrageergebnis nach der Schema-Evolution in ProSA




(b) Anfrageergebnis nach der Schema-Evolution auf der Graphdatenbank

Abbildung 17: Vergleich der Anfrageergebnisse nach der Schema-Evolution in ProSA mit der Graphdatenbank

In beiden Fällen liefern die Abfragen exakt 11 Tupel zurück und zeigen, dass der Spaltenname bzw. das Attribut von *name* zu *cityName* unbenannt wurde. Somit konnte überprüft werden, ob durch fehlerhafte Abbildung oder unvollständige

Transformation semantische Informationen verloren gegangen sind. Der Vergleich dient als Validierungsschritt, um sicherzustellen, dass sowohl Struktur als auch Inhalt der relationalen Datenbank nach der Schema-Evolution mit der ursprünglichen Graphdatenbank übereinstimmen.

## 8.5. Performance-Evaluation des Transformationsprozesses

Zur Bewertung der Effizienz und Robustheit des entwickelten Transformationsprozesses wurde ein Benchmarking anhand mehrerer realitätsnaher Beispieldatenbanken auf Neo4j und PGAdmin durchgeführt. Um ein aussagekräftiges Ergebnis zu erzielen, umfasst der Vergleich eine kleine Datenbank, eine mittelgroße Datenbank, eine Variante mit deutlich erhöhter Beziehungskomplexität sowie eine großskalierte Datenbank (Tabelle 3). Bei den verwendeten Datenbanken handelt es sich um offizielle Beispieldatenbanken, die von Neo4j bereitgestellt werden (Neo4j (2024)).

| Datenbank | Anzahl Knoten | Anzahl Kanten |
|---|---|---|
| Dr. Who | 1.060 | 2.286 |
| Movie Komprimiert | 28.863 | 66.261 |
| Movie | 28.863 | 166.261 |
| fraud-detection | 332.973 | 980.098 |

Tabelle 3: Vergleich der Datenbankgrößen

Die Durchführung der Benchmarks erfolgte auf einem MacBook Pro mit Apple M4 Pro-Chip. Um jedoch ein vergleichbares Ergebnis zu gewährleisten, wurde eine virtuelle Maschine unter Arch Linux eingerichtet. Diese wurde mit 4 CPU-Kernen und 16 GB Arbeitsspeicher konfiguriert. Um belastbare und vergleichbare Ergebnisse zu erhalten, wurden die Teilprozesse des Transformationsverfahrens auf dem beschriebenen System jeweils 50-fach ausgeführt. Diese Anzahl an Wiederholungen orientiert sich an Empfehlungen aus der Literatur, die eine ausreichende statistische Aussagekraft bei moderatem Ressourcenaufwand gewährleisten soll (Noori et al. (2025)). Die dabei gemessenen Laufzeiten wurden aufgezeichnet und visualisiert, um Schwankungen zu erkennen und die Robustheit sowie Effizienz der ein-

zelnen Schritte zu beurteilen. Zur besseren Analyse des Transformationsprozesses wurden die Laufzeiten des Extraktionsprozesses und der nachfolgenden Mapping-Operation, bei der das extrahierte Graphschema auf eine relationale Datenbank gemappt wird, separat analysiert.

Wie in Tabelle 4 dargestellt, variiert sowohl die Anzahl der Knoten als auch die Anzahl der Kanten sowie die sogenannte Beziehungsdichte deutlich zwischen den untersuchten Datenbanken. Zusätzlich unterscheiden sich die jeweiligen Laufzeiten der Schema-Extraktion und des Mappings in erheblichem Maße. Dabei wurde bei der Dr. Who-Datenbank die kürzeste Laufzeit sowohl bei der Extraktion (0,17s) als auch beim anschließenden Mapping (0,6s) gemessen. Bei der komprimierten Movie-Datenbank steigt die Extraktionszeit auf 9,3 Sekunden, während das Mapping rund 11 Sekunden in Anspruch nimmt. Für die unkomprimierte Movie-Datenbank liegt die Extraktionszeit bei 17 Sekunden und übertrifft damit die Mappingzeit von 13,5 Sekunden. Die größte Datenbank Fraud-detection benötigt wiederum bei der Extraktion rund 73 Sekunden, während das Mapping über 146 Sekunden benötigt, um die extrahierten Daten in ein relationales Schema zu überführen. Es ist zu beobachten, dass mit zunehmender Knoten- und Kantenanzahl auch die absolute Laufzeit ansteigt. Da sich die Beziehungsdichte, die das durchschnittliche Verhältnis zwischen Kanten und Knoten beschreibt , mit Werten zwischen 2,09 und 5,76 unterscheidet, lassen sich die absoluten Laufzeiten jedoch nur eingeschränkt miteinander vergleichen.

| **Datenbank** | **Beziehungsdichte** | **Extraktionszeit** | **Mappingzeit** |
|---|---|---|---|
| Dr. Who | 2,1566 | 0,1718s | 0,6s |
| Movie Komprimiert | 2,0878 | 9,3435s | 11,13s |
| Movie | 5,7603 | 17,03706s | 13,54s |
| Fraud-detection | 2,9434 | 73,1773s | 146,45s |

Tabelle 4: Vergleich der Beziehungsdichte sowie der durchschnittlichen Laufzeiten für Schema-Extraktion und Mapping über alle Datenbanken

Um den Einfluss der strukturellen Komplexität auf die Effizienz des Transformati-

onsprozesses differenzierter bewerten zu können, wurde neben der reinen Laufzeit eine Kennzahl zur Abschätzung der durchschnittlichen Traversalkosten eingeführt. Dabei handelt es sich um ein Maß für den durchschnittlichen zeitlichen Aufwand, den das System benötigt, um ein Objekt unter Berücksichtigung seiner strukturellen Verknüpfung zu extrahieren. Dies berücksichtigt nicht nur die absolute Datenmenge, sondern gewichtet die durchschnittliche Verarbeitungszeit pro Objekt zusätzlich mit der jeweiligen Beziehungsdichte. Somit konnte untersucht werden, ob mit zunehmender Beziehungsdichte auch die Traversalkosten steigen – sprich, ob die einzelnen Prozesse des Transformationsprozesses unabhängig von der strukturellen Komplexität agieren. Die Berechnung erfolgt nach folgender Formel:

$$\textbf{Durchschnittliche Traversalkosten} = \frac{\textbf{Laufzeit (s)}}{\textbf{Knoten + Kanten}} \times \textbf{Beziehungsdichte} \tag{8.1}$$

Somit lässt sich die Systemeffizienz im Kontext von stark verknüpften Datenmodellen differenziert bewerten. Dabei zeigen sich deutlich Unterschiede zwischen den untersuchten Datenbanken.

Abbildung 18 stellt die durchschnittlichen Traversalkosten pro Objekt sowohl für die Schemaextraktion (links) als auch für das Mapping (rechts) dar. In der linken Darstellung wird deutlich, dass die Traversalkosten bei der Extraktion je nach Datenbank deutlich variieren. Während bei der Movie-Datenbank ein erhöhter Wert von 0,0005 zu erkennen ist, weist Dr. Who den geringsten Wert mit 0,0001 auf. Die komprimierte Movie-Datenbank und Fraud-detection bewegen sich jeweils mit 0,0002 und 0,00017 im unteren Bereich.

In der rechten Darstellung liegen die Traversalkosten aller vier Datenbanken auf einem insgesamt ähnlichen Niveau zwischen 0,0002 und 0,0004. Es sind jedoch Unterschiede zu erkennen, diese fallen weniger stark aus als bei der Extraktion.

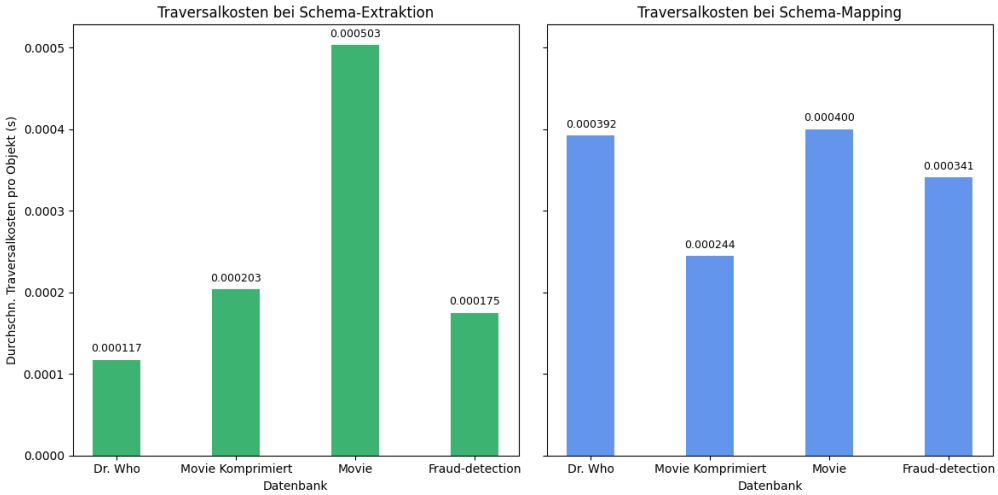



Abbildung 18: Vergleich der durchschnittlichen Traversalkosten pro Objekt bei Schemaextraktion (links) und Schema-Mapping (rechts)

Im Anschluss wurde die Robustheit des Systems untersucht. Hierfür wurde die Standardabweichung der jeweiligen Durchführungen analysiert. Da die absolute Streuung jedoch stark von der Höhe der Laufzeit abhängt und somit kein Vergleich zwischen unterschiedlich großen Datenbanken gezogen werden kann, wurde zusätzlich die relative Abweichung berechnet. Diese setzt die Standardabweichung ins Verhältnis zum jeweiligen Mittelwert der Laufzeit.

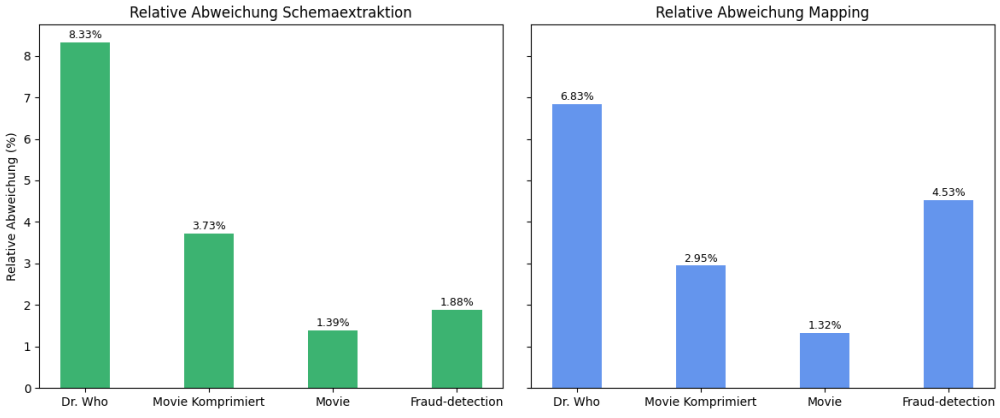



Abbildung 19: Vergleich der relativen Abweichung bei Schemaextraktion und Schem-Mapping

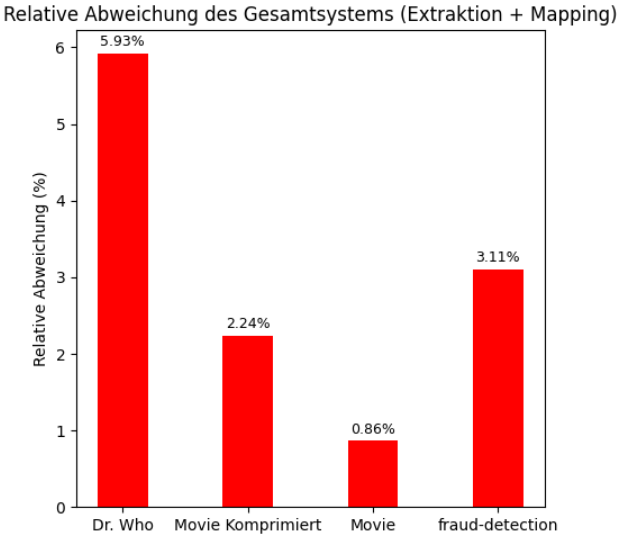



Abbildung 20: Relative Abweichung des Gesamtsystems

In der linken Abbildung, dass die relative Abweichung des Extraktionsprozesses visualisiert ist, fällt auf, dass die Dr. Who mit 8,3% die größte Abweichung aufzeigt, gefolgt von der komprimierten Movie-Datenbank mit 3,7%. Die geringste Abweichung wurde bei der Movie- und Fraud-Detection gemessen (Abbildung 19).

Auch bei relativen Abweichungen des Mappingprozesses, die auf der rechten Abbildung zu sehen sind, besitzt Dr. Who ebenfalls die höchste Abweichung mit 6,8% gefolgt von *Fraud-Detection* mit 4,5% und *Movie-Komprimiert* mit 2,9%. Bei der Movie-Datenbank wurde auch hier die geringste Abweichung von 1,3% gemessen (Abbildung 19).

Analysiert man die Robustheit des gesamten Transformationsprozesses, liegen die relativen Abweichungen in einem Bereich zwischen 0,9% (Movie) und 5,9% (Dr. Who). Die Datenbank *Movie Komprimiert* und *Fraud-detection* liegen mit 2,2% und 3,1% im mittleren Bereich und zeigen eine moderate Schwankung (Abbildung 20).

Die Evaluation der Robustheit und Effizienz ist essenziell, um die Praxistauglichkeit und Verlässlichkeit des entwickelten Systems beurteilen zu können. Gerade bei der Verarbeitung großer und komplexer Graphstrukturen spielt die Performanz eine zentrale Rolle, da eine lange Laufzeit und inkonsistente Ergebnisse die Anwendung einschränken würden. Deshalb liefern Messwerte wie Laufzeiten, Traversalkosten und relative Abweichungen wertvolle Hinweise auf mögliche Engpässe im

System und zeigen zudem, wie stabil das System bei unterschiedlich großen Datenmodellen arbeitet.

# 9. Diskussion

Das System ist in der Lage, eine Graphdatenbank in ein relationales Datenbankschema zu transformieren und erfüllt dabei die strukturellen Anforderungen, die für eine Weiterverarbeitung innerhalb von ProSA notwendig sind. Dies zeigten die Ergebnisse aus der zuvor durchgeführten Evaluierung. Durch die drei Prüf-Abfragen zur Validierung der Äquivalenz zwischen der ursprünglichen Graphdatenbank und der gemappten relationalen Datenbank lieferten in beiden Fällen identische Ergebnisse und zeigen somit, dass die Abbildung der Multilabel-Knoten sowie der Beziehungsinformationen in der relationalen Datenbank korrekt und verlustfrei erfolgt ist.

Ebenfalls erfolgte ein Vergleich zwischen der ursprünglichen Graphdatenbank und der zurücktransformierten Graphdatenbank, der den Rücktransformationsprozess auf seine Richtigkeit testet. Dabei konnte ebenfalls eine vollständige und korrekte Wiederherstellung der ursprünglichen Datenstruktur bestätigt werden. Einziger Unterschied besteht jedoch darin, dass bei der ursprünglichen Extraktion des Schemas - sofern keine explizite ID-Eigenschaft im Knoten angelegt wurde - die automatisch generierte ID aus Neo4j verwendet wurde. Diese wurde bei der Rücktransformation als reguläre Eigenschaft übernommen, wodurch Neo4j zusätzlich eine neue automatische interne ID generiert. Dieser Datenunterschied stellt jedoch kein Problem dar, da bei einer erneuten Schema-Extraktion nun alle Knoten eine ID besitzen und die generierte ID nicht aus der Datenbank extrahiert wird.

Ein weiterer zentraler Aspekt der Evaluation war der Vergleich der beiden Datenbanken nach der Anwendung eines Evolutionsoperators. Dabei wurde einmal die Evolution direkt über die Datenbank durchgeführt und einmal in Verbindung mit ProSA. Durch die Durchführung des RENAME-Operators auf beiden Systemen konnte gezeigt werden, dass sowohl über den direkten Weg als auch über ProSA

strukturelle Änderungen konsistent umgesetzt wurden. Die anschließende Anfrage zeigte, dass es zwischen beiden Datenbanken keinerlei Unterschiede in den Ergebnissen gab und somit ein identischer Datenstand vorliegt. Dieser Abgleich bestätigt, dass nicht nur die Daten korrekt transformiert, sondern auch spätere Schemaänderungen zuverlässig synchron auf beiden Datenbankmodellen abgebildet werden können.

Neben der korrekten funktionalen Umsetzung wurde ebenfalls die Performance des Systems analysiert. Ziel dabei war es, die Effizienz und die Robustheit des Transformationsprozesses zu bewerten. Hierfür wurden vier unterschiedlich große Graphdatenbanken hinsichtlich ihrer Struktur sowie ihrer Verarbeitungskosten untersucht.

Bei der Analyse der durchschnittlichen Laufzeiten konnte festgestellt werden, dass die Laufzeit sowohl bei der Schema-Extraktion als auch beim Mapping mit zunehmender Datenmenge ansteigt. Da sich jedoch die Beziehungsdichte in allen vier Datenbanken unterscheidet, reicht die alleinige Betrachtung der reinen Laufzeit nicht aus. Die Beziehungsdichte, die das durchschnittliche Verhältnis von Kanten pro Knoten beschreibt, beeinflusst den strukturellen Aufwand, den das System bei der Traversierung und der Verarbeitung leisten muss. Deshalb wurden die durchschnittlichen Traversalkosten eingeführt, die die Laufzeit in Relation zur Datenmenge und zur Komplexität setzen. Somit kann bewertet werden, ob eine lange Laufzeit auf die große Menge an Daten oder auf die Struktur der Graphdatenbank zurückzuführen ist. Beim Vergleich der Traversalkosten für Schema-Extraktion und Mapping wurde deutlich, dass die Beziehungsdichte insbesondere im Extraktionsprozess einen Einfluss auf die Laufzeit hat. Datenbanken mit höherer Beziehungsdichte weisen hier erhöhte Traversalkosten auf. Dies weist darauf hin, dass die Komplexität und die Struktur des Graphen bei der Extraktion eine zentrale Rolle spielen. Im Gegensatz dazu zeigen sich im Mappingprozess vergleichsweise geringere Unterschiede in den Traversalkosten zwischen den jeweiligen Datenbanken. Dies zeigt, dass die absolute Datenmenge das Mapping mehr beeinflusst als die Beziehungsdichte.

Zusätzlich wurde die Robustheit des Systems betrachtet, welche angibt, wie stabil die Laufzeit bei mehrfacher Ausführung bleibt. Dabei wurde die relative Abweichung in Betracht gezogen, da diese unabhängig von der absoluten Höhe der Laufzeit ist und somit eine vergleichbare Bewertung der Schwankungen über unterschiedlich große Datenbanken gewährleistet. Die Analyse ergab, dass sowohl bei der Schema-Extraktion als auch bei dem Mapping-Prozess bei mittelgroßen und großen Datenbanken die relative Abweichung in einem niedrigen Bereich liegt. Jedoch bei kleineren Datenbanken wie zum Beispiel Dr. Who war die relative Abweichung mit 8% und 6,8% deutlich höher. Dies kann darauf zurückzuführen sein, dass bei geringen Datenmengen Systemprozesse oder Initialisierungseffekte, wie die Verbindungsherstellung zur Datenbank, stärkere Einflüsse auf die gemessenen Laufzeiten haben. Betrachtet man jedoch die Robustheit des gesamten Systems, zeigt sich ein ähnliches Bild: Die relative Abweichung des gesamten Prozesses bleibt über alle Datenbanken hinweg auf einem niedrigen Niveau. Lediglich bei kleinen Datenbanken ist eine erhöhte Streuung erkennbar. Dies bestätigt die Robustheit des Systems und zeigt, dass es bei wachsender Datenmenge stabiler und zuverlässiger arbeitet.

Es ist jedoch zu beachten, dass die durchgeführte Evolution in einem isolierten System stattfand. Unterschiede in Hardware, Systemlast oder parallelen Prozessen können die Streuung und Laufzeit einzelner Vorgänge zusätzlich beeinflussen. Außerdem stellen die getesteten Datenbanken nur einen Ausschnitt möglicher Graphstrukturen dar.

Für zukünftige Arbeiten könnten gezielt besonders stark verknüpfte Datenbanken, die eine hohe Beziehungsdichte aufweisen, einbezogen werden, um mögliche Schwachstellen weiter zu identifizieren. Somit könnten weitere Optimierungsansätze entwickelt werden, die eine noch effizientere Verarbeitung von komplexeren Strukturen ermöglichen. Ein derzeitiger Limitierungsfaktor besteht darin, dass sich der Benutzer an einer Reihe vordefinierter Modellierungskonventionen halten muss. So wird beispielsweise vorausgesetzt

# 10. Fazit

Die vorgelegte Bachelorarbeit stellte ein System zur reversiblen Transformierung von Graphdatenbanken in relationale Datenbanken vor. Ein besonderer Fokus lag dabei auf der Behandlung von Multilabel-Knoten, welche in Graphdatenbanken eine zentrale Rolle bei der Modellierung komplexer Entitäten spielen, jedoch in einem relationalen Schema nicht direkt abgebildet werden können.

Durch die Kombination etablierter Methoden zur Extraktion und Mapping des Schemas mit einer strukturierten Behandlung von Multilabel-Knoten mittels Supertype-Modellierung konnte ein robustes Transformationssystem realisiert werden. Die anschließende Rücktransformation und deren Evaluation zeigten, dass sowohl die Struktur als auch der semantische Gehalt der Daten erhalten bleiben und konsistent überführt werden. Die erfolgreiche Integration in ProSA bestätigt darüber hinaus die praktische Anwendbarkeit des entwickelten Ansatzes.

Die durchgeführten Performance-Analysen auf unterschiedlichen Datenbanken mit unterschiedlichen Größen und Komplexität belegten die Effizienz und Skalierbarkeit des Verfahrens. Mithilfe der Berechnung der Traversalkosten konnte eine differenzierte Bewertung der Systemleistung unter der Berücksichtigung der strukturellen Komplexität der Graphdaten vorgenommen werden.

Für zukünftige Arbeiten bietet sich insbesondere eine vertiefte Betrachtung der Traversalkosten als Ansatzpunkt für Optimierungen an. Die in dieser Arbeit identifizierten Unterschiede in der Verarbeitungszeit in Abhängigkeit von der strukturellen Komplexität legen nahe, dass ein weiterer Ansatz ausarbeitet werden könnte, der den Transformationsprozess auf Basis der Traversalkosten dynamisch anpasst. Somit könnten Knoten und Kanten mit hoher Beziehungsdichte gezielt vorverarbeitet oder in spezifische Teilprozesse behandelt werden, um die Gesamtperformance zu steigern. Außerdem bietet sich die Möglichkeit, den Transformationsprozess auf

weitere Graphdatenbankmodelle auszudehnen, um somit die Kompatibilität mit ProSA zu erhöhen und dessen Einsatzspektrum zu erweitern.

Insgesamt leistet das entwickelte System einen grundlegenden Beitrag zur Überbrückung der konzeptionellen Kluft zwischen Graph- und relationalen Datenbanksystemen und schafft damit eine fundierte Grundlage für weiterführende Forschungs- und Entwicklungsarbeiten.

# Literaturverzeichnis

Auge, T. (2023). Prosa: A provenance system for reproducing query results. In *Companion proceedings of the acm web conference 2023* (S. 1555–1558). New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/3543873.3587563 doi: 10.1145/3543873.3587563

Auge, T., Hanzig, M. & Heuer, A. (2022). Prosa pipeline: Provenance conquers the chase. In *Symposium on advances in databases and information systems*. Zugriff auf https://api.semanticscholar.org/CorpusID:251980673

Auge, T. & Heuer, A. (2019). Prosa—using the chase for provenance management. In *Advances in databases and information systems: 23rd european conference, adbis 2019, bled, slovenia, september 8–11, 2019, proceedings* (S. 357–372). Berlin, Heidelberg: Springer-Verlag. Zugriff auf https://doi.org/10.1007/978-3-030-28730-6_22 doi: 10.1007/978-3-030-28730-6_22

Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D. & Tsamoura, E. (2017). Benchmarking the chase. In *Proceedings of the 36th acm sigmod-sigact-sigai symposium on principles of database systems* (S. 37–52). New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/3034786.3034796 doi: 10.1145/3034786.3034796

Bonifati, A., Dumbrava, S., Martinez, E., Ghasemi, F., Jaffré, M., Luton, P. & Pickles, T. (2022, August). Discopg: property graph schema discovery and exploration. *Proc. VLDB Endow.*, *15* (12), 3654–3657. Zugriff auf https://doi.org/10.14778/3554821.3554867 doi: 10.14778/3554821.3554867

Chillón, A. H., Klettke, M., Ruiz, D. S. & Molina, J. G. (2024). A generic schema evolution approach for nosql and relational databases. *IEEE Transactions on Knowledge and Data Engineering*, *36* (7), 2774-2789. doi: 10.1109/TKDE.2024.3362273

Comyn-Wattiau, I. & Akoka, J. (2017). Model driven reverse engineering of nosql property graph databases: The case of neo4j. In *2017 ieee international conference on big data (big data)* (S. 453-458). doi: 10.1109/BigData.2017.8257957

Curino, C. A., Moon, H. J. & Zaniolo, C. (2008, August). Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, *1* (1), 761–772. Zugriff auf https://doi.org/10.14778/1453856.1453939 doi: 10.14778/1453856.1453939

De Virgilio, R., Maccioni, A. & Torlone, R. (2013). Converting relational to graph databases. In *First international workshop on graph data management experiences and systems*. New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/2484425.2484426 doi: 10.1145/2484425.2484426

Frozza, A. A., Jacinto, S. & Mello, R. (2020, 08). An approach for schema extraction of nosql graph databases. In (S. 271-278). doi: 10.1109/IRI49571.2020.00046

Hausler, D. (2024). Estimation, impact and visualization of schema evolution in graph databases. In *Proceedings of the acm/ieee 27th international conference on model driven engineering languages and systems* (S. 123–129). New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/3652620.3688196 doi: 10.1145/3652620.3688196

Herschel, M., Diestelkämper, R. & Ben Lahmar, H. (2017, 12). A survey on provenance: What for? what form? what from? *The VLDB Journal*, *26*. doi: 10.1007/s00778-017-0486-1

Heuer, A., Saake, G. & Sattler, K. U. (2018). *Datenbanken - konzepte und sprachen, 6. auflage*. MITP. Zugriff auf https://mitp.de/IT-WEB/Datenbanken/Datenbanken-Konzepte-und-Sprachen-oxid.html

Lourenço, B., Vaz, C., Coimbra, M. E. & Francisco, A. P. (2023). phylodb: A framework for large-scale phylogenetic analysis. *arXiv preprint arXiv:2310.12658*. Zugriff auf https://arxiv.org/abs/2310.12658

Neo4j. (2024). *Example data - neo4j documentation.* Zugriff auf https://neo4j.com/docs/getting-started/appendix/example-data/ (Accessed: 14.04.2025)

Neo4j Inc. (2025). *Neo4j – The World's Leading Graph Database Platform.* Zugriff auf https://neo4j.com/ (Zugriff am 15. April 2025)

*Neo4j python driver 5.28.* (2025). https://neo4j.com/docs/api/python-driver/current/. (Zugriff am 14. April 2025)

Noori, M., Valiante, E., Vaerenbergh, T. V., Mohseni, M. & Rozada, I. (2025). *A statistical analysis for per-instance evaluation of stochastic optimizers: How many repeats are enough?* Zugriff auf https://arxiv.org/abs/2503.16589

Palagashvili, A. M. & Stupnikov, S. A. (2023). Reversible mapping of relational and graph databases. , *33* (2), 285-295. doi: 10.1134/S1054661823020098

Preusse, M. (2017). Graphdatenbanken helfen bei der biomedizinischen forschung. *BigData Insider*. Zugriff auf https://www.bigdata-insider.de/

graphdatenbanken-helfen-bei-der-biomedizinischen-forschung-a-777926/ (Accessed: 2023-01-01)

Sanderson, L.-A., Caron, C. T., Tan, R. & Bett, K. E. (2021). A postgresql tripal solution for large-scale genotypic and phenotypic data. *Database: The Journal of Biological Databases and Curation, 2021*. Zugriff auf https://api.semanticscholar.org/CorpusID:237008577

Silberschatz, A., Korth, H. F. & Sudarshan, S. (2010). *Database system concepts* (7. Aufl.). New York: McGraw-Hill. Zugriff auf http://www.db-book.com/

Stonebraker, M. & Rowe, L. A. (1991). The design of postgres. *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, 340–355.

Team, T. (2018). *Timescaledb: An open-source time-series database optimized for fast ingest and complex queries.* Zugriff auf https://www.timescale.com (Accessed: 2023-01-01)

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y. & Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual acm southeast conference.* New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/1900008.1900067 doi: 10.1145/1900008.1900067

Wardani, D. W. & Kiing, J. (2014). Semantic mapping relational to graph model. In *2014 international conference on computer, control, informatics and its applications, IC3INA 2014, bandung, indonesia, october 21-23, 2014* (S. 160–165). IEEE. Zugriff auf https://doi.org/10.1109/IC3INA.2014.7042620 doi: 10.1109/IC3INA.2014.7042620

Yousaf, M. & Wolter, D. (2019). How to identify appropriate key-value pairs for querying osm. In *Proceedings of the 13th workshop on geographic information retrieval.* New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/3371140.3371147 doi: 10.1145/3371140.3371147

## Erklärung zur Urheberschaft

Ich habe die Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle Zitate und Übernahmen von fremden Aussagen kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Die vorgelegten Druckexemplare und die vorgelegte digitale Version sind identisch.

Regensburg, 24. April 2025 _____

Unterschrift

## Erklärung zur Lizenzierung und Publikation dieser Arbeit

**Name:** Timo Hanöffner

**Titel der Arbeit:** *Schema-Mapping von Neo4j nach ProSA*

Hiermit gestatte ich die Verwendung der schriftlichen Ausarbeitung zeitlich unbegrenzt und nicht-exklusiv unter folgenden Bedingungen:

- ☐ Nur zur Bewertung dieser Arbeit
- ☐ Nur innerhalb des Lehrstuhls im Rahmen von Forschung und Lehre
- ☒ Unter einer Creative-Commons-Lizenz mit den folgenden Einschränkungen:
  - ☒ BY – Namensnennung des Autors
  - ☐ NC – Nichtkommerziell
  - ☐ SA – Share-Alike, d.h. alle Änderungen müssen unter die gleiche Lizenz gestellt werden.

(An Zitaten und Abbildungen aus fremden Quellen werden keine weiteren Rechte eingeräumt.)

Außerdem gestatte ich die Verwendung des im Rahmen dieser Arbeit erstellten Quellcodes unter folgender Lizenz:

- ☐ Nur zur Bewertung dieser Arbeit
- ☐ Nur innerhalb des Lehrstuhls im Rahmen von Forschung und Lehre
- ☐ Unter der CC-0-Lizenz (= beliebige Nutzung)
- ☒ Unter der MIT-Lizenz (= Namensnennung)
- ☐ Unter der GPLv3-Lizenz (oder neuere Versionen)

(An explizit mit einer anderen Lizenz gekennzeichneten Bibliotheken und Daten werden keine weiteren Rechte eingeräumt.)

Ich willige ein, dass der Lehrstuhl für Medieninformatik diese Arbeit – falls sie besonders gut ausfällt - auf dem Publikationsserver der Universität Regensburg veröffentlichen lässt.

Ich übertrage deshalb der Universität Regensburg das Recht, die Arbeit elektronisch zu speichern und in Datennetzen öffentlich zugänglich zu machen. Ich übertrage der Universität Regensburg ferner das Recht zur Konvertierung zum Zwecke der Langzeitarchivierung unter Beachtung der Bewahrung des Inhalts (die Originalarchivierung bleibt erhalten).
Ich erkläre außerdem, dass von mir die urheber- und lizenzrechtliche Seite (Copyright) geklärt wurde und Rechte Dritter der Publikation nicht entgegenstehen.

- ☒ Ja, für die komplette Arbeit inklusive Anhang
- ☐ Ja, für eine um vertrauliche Informationen gekürzte Variante (auf dem Datenträger beigefügt)
- ☐ Nein
- ☐ Sperrvermerk bis (Datum):

Regensburg, 24. April 2025

Unterschrift

# A. Anhang 1

Codebeispiel 19: XML-File des Schema-Operators RENAME

```
<input>
  <schema>
    <relations>
      <relation name="city" tag="T">
        <attribute name="id" type="string" />
        <attribute name="name" type="string" />
      </relation>
      <relation name="cityNew" tag="S">
        <attribute name="id" type="string" />
        <attribute name="cityName" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="cityNew" >
            <variable name="id" type="V" index="1" />
            <variable name="cityName" type="V" index="1" />
          </atom>
        </body>
        <head>
          <atom name="city" >
            <variable name="id" type="V" index="1" />
            <variable name="name" type="V" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance/>
</input>
```

# B. Inhalt des beigefügten Datenträgers

Beispiel (Ordner + Beschreibung):

| | |
|---|---|
| `/1_Ausarbeitung` | Die schriftliche Ausarbeitung als PDF und DOC |
| `/2_Code` | Python Quellcode |
| `/4_Quellen` | Alle in der Arbeit zitierten Quellen im PDF-Format |
| `/5_Bilder` | Alle selbst erstellten und aus anderen Quellen übernommenen Bilder |
| `/6_Vorträge` | Folien von Antritts- und Abschlussvortrag im PDF-Format |