Effiziente Darstellung klinischer Patientendaten in einem Knowledge Graphen

Tobias Hübenthal
5640520

Bachelorarbeit
Wirtschaftsmathematik (B.Sc.)

Department Mathematik/Informatik
Mathematisch-Naturwissenschaftliche Fakultät
Universität zu Köln
Juli 2020

Fraunhofer SCAI

Erstgutachterin
Dr. Vera Weil

Zweitgutachter
Dr. Jens Dörpinghaus

Inhaltsverzeichnis

Abkürzungsverzeichnis

CSV comma separated values

RPQ regular path query

CQ conjunctive query

CRPQ conjunctive regular path query

ECRPQ extended conjunctive regular path query

ApoE Apolipoprotein E

HGNC human gene nomenclature

SNP single nucleotide polymorphism

SQL Structured Query Language

IDSN Integrative Datensemantik für die Neurodegenerative Forschung

1 Einleitung

1.1 Ausgangssituation und Motivation

Der britische Informatiker Christopher Strachey schrieb 1954 in einem Brief an den Mathematiker Alan Turing: „I am convinced that the crux of the problem of learning is recognizing relationships and being able to use them.“[1].
Heutzutage gibt es immer mehr und immer größere Datenmengen. Dies gilt insbesondere für den Bereich klinischer Daten. Sie werden in Massen gesammelt und sind ohne digitale Aufarbeitung unüberschaubar und bleiben weit hinter ihrem Potential zurück. Erst mit einer effizienten Datenstruktur lassen sich mehr Informationen und neues Wissen gewinnen. In Zusammenarbeit des Uniklinikums Bonn, des Deutschen Zentrums für Neurodegenerative Erkrankungen e.V. in Bonn und des Fraunhofer Instituts SCAI St. Augustin wurde mit Förderung des Bundesministeriums für Bildung und Forschung das Projekt der Integrativen Daten-Semantik für Neurodegenerationsforschung (IDSN) ins Leben gerufen. Mithilfe der dort gewonnen Daten zur Alzheimererkrankung sollen unter anderem unter den Patienten [1] besondere Risikogruppen und mögliche Untergruppen, die auf bestimmte Behandlungen ansprechen, gefunden werden. Dabei müssen erhobene Primärdaten der untersuchten Personen anwendungsorientiert mit Sekundärdaten aus Publikationen und Datenbanken verknüpft werden. [2]
Aber wie lassen sich klinische Patientendaten aus einer Datenbank mit einem geeigneten Datenschema effizient als Knowledge Graph speichern? Und wie lassen sich danach mithilfe der so verknüpften Daten Abfragen generieren? Welche Laufzeit hat dabei eine anwendungsbezogene Abfrage nach geeigneter Literatur zu einem gegebenen Patienten? Im Rahmen dieser Arbeit soll, aufbauend auf den bereits gespeicherten *PubMed* Daten aus [3], ein Modell vorgestellt werden, das die gesammelten Primärdaten effizient in strukturierte, bereichsspezifische Umgebungen, in diesem Fall Ontologien, einbettet. Über den so generierten Knowledge Graphen sollen dann einzelne Abfragen im Hinblick auf Effizienz untersucht werden.

[1] Aus Gründen der besseren Lesbarkeit wird im Text verallgemeinernd das generische Maskulinum verwendet. Diese Formulierungen umfassen gleichermaßen Personen aller Geschlechter; alle sind damit selbstverständlich gleichberechtigt angesprochen.

1.2 Aufbau der Arbeit

Zu Beginn der Arbeit werden grundlegende Definitionen und Konstruktionen der Graphentheorie gegeben. Sie stellen das Fundament dar, auf das der praktische Teil der Arbeit gesetzt wird. Darüber hinaus wird eine kurze Einführung zu den verschiedenen Datenbanksystemen gegeben und dargelegt, weshalb für diese Arbeit die Graphdatenbank *Neo4j* ausgewählt und verwendet wird. Außerdem wird die in *Neo4j* verwendete Sprache *Cypher* vorgestellt. Da die klinischen Daten über Ontologien mit Kontext verknüpft werden sollen, wird der Aufbau und Konzept der Ontologien ebenfalls in den Grundlagen erläutert. Im Kontext der Forschungsfrage nach der Effizienz des Mappings und der Umsetzung des Graphen schließt das Grundlagenkapitel mit kurzen Ausführungen zur Komplexitätstheorie ab. Diese werden am Ende für die Beurteilung der verwendeten Methode herangezogen.

Kapitel 3 stellt die vorgefundenen Systeme sowie die gegebenen Daten vor. Anschließend wird eine Reihe von biomedizinischen Fragestellungen präsentiert, die als Beispielinstanzen für die spätere Verwendung eines solchen Graphen dienen sollen und als Teil der Grundlage für die zuvor erwähnte und am Ende der Arbeit beschriebene Laufzeitanalyse verwendet werden. Zu diesem Zweck werden im gleichen Kapitel die Fragen nach zugrundeliegenden algorithmischen Problemen klassifiziert, um die spätere Analyse zu unterstützen. Die kategorisierten Fragestellungen können dann in *Cypher*-Querys ausgedrückt werden.

Anschließend wird durch die Betrachtung des gegebenen Datenmodells und die Einteilung der Daten in Klassen und Relationen ein Datenschema erarbeitet, das dann zunächst generalisiert und an das bereits vorhandene Schema aus der vorherigen Masterarbeit [3] angefügt wird.

Schließlich werden, aufbauend auf diesem generischen Schema, mithilfe eines Pythonskripts aus artifiziellen Testdaten CSV-Dateien erstellt, die dann in eine *Neo4j* Graphdatenbank eingelesen werden können. Zusätzlich wird über ein zweites Skript anhand von [4] eine Umgebung des zentralen Apolipoprotein-E-Gens als sehr kleine eigene Ontologie erstelllt und ebenfalls als CSV-Dateien ausgegeben.

Darauf folgend werden die CSV-Dateien und die zugehörigen Header der Ontologien sowie die des Mappings in die Datenbank importiert. Anschließend wird die Güte der Lösung des Problems mithilfe der Überlegungen zur Laufzeitanalyse gemäß Abschnitt 2.4 analysiert. Dabei spielt nicht nur die Effizienz und Laufzeit der Python-Skripte, sondern auch die der Querys eine wichtige Rolle.

Den Abschluss bilden ein Fazit und ein Ausblick auf weitere Möglichkeiten zur Erweiterung des Graphen, zur Verwendung des Konzepts mit Realdaten und zur Verbesserung der Geschwindigkeit der Skripte und der Querys gegeben.

Alle im Rahmen dieser Arbeit verwendeten zusätzlichen Dateien sind genau wie die selbst erstellten auf dem beigefügten Datenträger hinterlegt und unter `https://github.com/TbsHbnthl/bachelorthesis.git` abrufbar.

2 Grundlagen

Die Arbeit beruht auf Grundlagen der Graphentheorie sowie auf darauf aufbauenden technischen und konzeptionellen Weiterentwicklungen. Zunächst werden in diesem Kapitel die elementaren Definitionen und Konzepte erläutert.

2.1 Graphentheorie

Die wichtigste Struktur für diese Arbeit stellt der Graph dar.

Definition 2.1.1 *Ein gerichteter Graph $D = (V, E, \rho)$ ist ein Tripel, bestehend aus einer disjunkten Vereinigung einer endlichen Knotenmenge V, einer endlichen Kantenmenge E sowie einer Funktion $\rho : E \to (V \times V)$. Hierbei weist ρ jedem Element $e \in E$ zwei geordnete Elemente $v, w \in V$ zu. [5][6]*

Neben gerichteten Graphen gibt es auch ungerichtete Graphen mit nicht-orientierten Kanten. Diese spielen im Verlauf der Arbeit jedoch keine Rolle und werden daher nicht weiter betrachtet. Das oben genannte Konzept des Graphen kann zu einem Property Graphen ausgebaut werden. Zu diesem Zweck ermöglicht man es, sowohl Knoten als auch Kanten in Klassen zu unterteilen und ihnen Attribute zuzuordnen.

Definition 2.1.2 *Ein Graph $G' = (V', E') \subseteq G = (V, E)$ ist ein Subgraph von G, wenn die Bedingungen $V' \subseteq V$ und $E' \subseteq E$ erfüllt sind. [7]*

Definition 2.1.3 *Unter einem Property Graphen G versteht man ein Acht-Tupel $G = (V, E, L, P, X, \rho, \lambda, \sigma)$. Dabei wird der Graph aus 2.1.1 um die Mengen L, P und X sowie die Funktionen $\lambda : (V \cup E) \to L$ und $\sigma : (V \cup E) \times P \to X$ erweitert. Die Funktion λ weist sowohl Knoten als auch Kanten Labels $l \in L$ zu. σ ordnet den Knoten und Kanten Properties $p \in P$ zu. Jedes solche p kann dabei Werte $x(p) \in X$ annehmen.[5]*

Die Funktionen λ und σ sind in 2.1.3 als injektiv definiert. Dies kann jedoch durch eine Relaxierung insofern geändert werden, dass Knoten oder Kanten erlaubt wird, mehrere Labels $l_1, l_2 \in L$ oder Werte $x_1(p), x_2(p) \in X$ zu einem Attribut zu speichern. Im Rahmen dieser Arbeit und der später noch vorgestellten Graphdatenbank *Neo4j* wird dies jedoch

nicht weiter vertieft, da es für die gegebene Problematik keinen Mehrwert aufweist oder sogar nicht umsetzbar ist. In [8] wird keine einheitliche Definition eines Knowledge Graphen gegeben. Es werden lediglich vier wichtige Aspekte genannt, die erfüllt sein müssen, um ihn aus der Menge aller Graphen herauszuheben:

- Es werden realitätsnahe Entitäten und deren Beziehungen zueinander beschrieben.
- Klassen und Relationen werden mithilfe eines Schemas definiert.
- Es ist möglich, beliebige Entitäten paarweise mit einer Relation zu versehen.
- Es werden mehrere Themenbereiche abgedeckt.

Dabei versteht man unter einer Entität ein eindeutig identifizierbares Objekt, zu dem Informationen gespeichert werden. [9] Für diese Arbeit wird folgende Definition verwendet, die sich jedoch auch mit den oben genannten vier Punkten verträgt:

Definition 2.1.4 *Ein Knowledge Graph $G = (E,R)$ ist zunächst als ein Graph definiert, dessen Knotenmenge E aus Entitäten und dessen Kantenmenge R aus Relationen zwischen diesen besteht. Dabei ist $E = \{E_1,...,E_n\}$ eine Vereinigung strukturierter Ontologien E_i und $R = \{R_1,...,R_m\}$ analog eine Vereinigung intra- und interontologischer Relationen. Jede Entität $e \in E$ kann darüber hinaus mit zusätzlichen Kontextinformationen $C = \{C_1,...,C_k\}$ verknüpft werden, welche im Allgemeinen selbst wieder Entitäten sind. [10]*

Diese Definition kann man für die spätere Anwendung in *Neo4j* mit den in 2.1.3 definierten Funktionen ρ, λ und σ verknüpfen, um den Knowledge Graphen zu erhalten, der im weiteren Verlauf der Arbeit verwendet wird.

Definition 2.1.5 *Ein Pfad $P = (V,E)$ ist ein nicht-leerer Graph, der folgende Form hat: $V = \{v_0,v_1,...,v_k\}$ $E = \{(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)\}$, wobei alle v_i paarweise verschieden sind $i \in \{0,1,...,k\}$. Die beiden Knoten $v_0,v_k \in V$ sind dabei durch P miteinander verknüpft und werden Start- bzw. Endknoten von P genannt. Oft verwendet man lediglich die simplifizierte Notation $P = (v_0,v_1,...,v_k)$ für einen v_0v_k-Pfad. Die Funktion $\rho : E \to (V \times V)$ aus 2.1.1 ordnet dabei einer Kante $e_i \in E$ die beiden im Pfad aufeinander folgenden Knoten $v_{i-1},v_i \in V$ zu. Die Länge eines Pfades P ist zunächst als Anzahl der Kanten in P definiert. Es ist jedoch auch möglich, die Kanten (v_{i-1},v_i) mit einer Länge zu versehen und die Länge des Pfades P somit als Summe über die Kantenlängen zu definieren. [7]*

Definition 2.1.6 *Wenn $P' = (v_0,v_1,...,v_{k-1})$ ein Pfad in einem Graphen $G = (V,E)$ mit $v_i \in V$ $\forall i = 1,...,k$ mit $k \geq 3$ ist, nennt man $P = P' + v_0$ einen Kreis. [7]*

Definition 2.1.7 *Ein Graph $G = (V,E)$ heißt zusammenhängend, wenn zwei beliebige paarweise verschiedene Knoten $v_i,v_j \in V$ über einen Pfad P in G verbunden sind. [7]*

Definition 2.1.8 *Ein Baum ist ein kreisfreier (azyklischer) und zusammenhängender Graph. [7]*

Definition 2.1.9 *Ein Spannbaum $G' = (V', E')$ ist ein Subgraph eines Graphen $G = (V, E)$, für den gilt, dass er ein Baum ist und dass $V' = V$. [7]*

Definition 2.1.10 *Für ein $k \geq 1$ ist die k-Nachbarschaft eines Knoten $v \in V$ eines Graphen $G = (V, E)$ als eine Knotenmenge $V' \subset V$ definiert, deren Knoten $v' \in V'$ eine Distanz von k oder weniger zu v haben. Dabei wird einer Kante die Länge 1 als Distanz zugewiesen. [11]*

Die letzte Definition kann auch mit Kanten betrachtet werden, die eine eigene Kantenlänge als Attribut besitzen. Dies ist für diese Arbeit jedoch nicht von Belang.

Definition 2.1.11 *Für einen Graphen $G = (V, E)$ und einen Knoten $v \in V$ ist $g(v) := |\{u \in V : \exists e = (u, v)$ oder $e = (v, u) \in E\}|$ der Grad des Knoten v. [12]*

Definition 2.1.12 *Eine stark verbundene Komponente in einem gerichteten Graphen $G = (V, E)$ ist eine maximale Menge von Knoten $U \subset V$, so dass $\forall u, v \in U \exists (u, v) \wedge (v, u) \in E \in U$. [13]*

2.2 Datenbanken

Eine Datenbank ist ein Mittel zur Ordnung und Strukturierung einer Menge von Daten. Die Daten werden gespeichert und können durch eine dankenbankspezifische Skriptsprache abgefragt, modifiziert und gelöscht werden. Es sind vier verschiedene Funktionen erforderlich:

- Modifikation der Struktur der Daten
- Einfügung, Löschung und Änderung der Daten
- Zugriff auf die Daten über externe Anwendungen
- Verwaltung der Datenbank.

Dabei wird anhand ihres Aufbaus, ihres Anwendungsbereichs, ihrer Möglichkeiten sowie der Vor- und Nachteile zwischen verschiedenen Datenbankmodellen differenziert, von denen im Folgenden zwei genauer dargestellt werden. [14]

2.2.1 Relationale Datenbanken

Als relationale oder auch SQL (Structured Query Language) Datenbanken wird eine Datenbank bezeichnet, welche die Daten strukturiert in Tabellen speichert. Jede Tabelle entspricht dabei einer Relation, welche die Art und Anzahl der Spalten vorgibt. Jede Zeile steht für einen unterschiedlichen Datensatz (Tupel) und jede Spalte entspricht einem Attribut des Datensatzes. Jedes Tupel muss durch einen einzigartigen Schlüssel gekennzeichnet werden, welcher dieses eindeutig identifiziert und sich auch nicht ändern darf.
Sehr rechen- und speicheraufwendig sind dabei sogenannte Joins, Verknüpfungen zwischen zwei oder mehr Relationen. Um diesen Aufwand einzusparen, werden häufig verwendete Joins mitunter in eigenen Relationen gespeichert, was bei einer großen Anzahl zu Unübersichtlichkeit und Ineffizienz oder sogar zu Unbrauchbarkeit führt. [14] [15]

2.2.2 NoSQL-Datenbanken

Der Name NoSQL steht für Not (only) SQL. Diese Datenbanken benötigen keine festgelegten tabellarischen Schemata und vermeiden oben genannte Joins. Sie unterscheiden sich von traditionellen Datenbanken außerdem in zwei weiteren Kernpunkten: Skalierbarkeit und Konsistenz. Zur Erreichung effizienter Skalierbarkeit der Datenbank werden mehrere Instanzen von ihr erstellt, die jedoch nach Änderungen in einer der Instanzen nicht notwendigerweise identische Daten enthalten. Dadurch entsteht Inkonsistenz. SQL-Datenbanken sind hingegen so aufgebaut, dass nach Abschluss jeder Interaktion die Daten konsistent sind.
Der Inkonsistenz Abhilfe schaffen kann das Modell der sogenannten Eventual Consistency. Dabei werden die einzelnen Instanzen zu gegebenen Zeitpunkten gespiegelt, das heißt Daten werden abgeglichen und Werte und Änderungen für alle Tabellen übernommen. Dadurch verkürzt sich die Dauer der Inkonsistenz auf ein gegebenes Zeitfenster.
Grundsätzlich wird zwischen vier verschiedenen Arten der NoSQL-Datenbanken unterschieden:

- Key-Value-Datenbanken: Hier werden die Daten in einer 1:n-Zuordnung über Schlüssel gespeichert. Für einfache Anwendungen ist dies ausreichend und durch die Simplizität sehr effizient.
- Spaltendatenbanken: Ähnlich wie bei den erwähnten Key-Value-Datenbanken enthält hier jede Spalte einen Schlüssel, Werte und zusätzlich einen Zeitstempel. Dies wirkt der Inkonsistenz entgegen, da veraltete Daten durch diesen Zeitstempel gekennzeichnet sind.
- Dokumentendatenbanken: Auch diese Datenbanken sind in der einfachsten Form eine Art der Key-Value-Datenbanken, wobei aber die Werte strukturierte Dokumente sind. Dadurch werden komplexere und schnellere Abfragen möglich.

- Graphdatenbanken: Das Modell dieser Datenbanken unterscheidet sich grundlegend von den zuvor genannten. Es besteht aus Knoten und Kanten, die jeweils eine beliebige Anzahl von Attributen oder Eigenschaften besitzen können. Darüber hinaus können die einzelnen Knoten willkürliche Relationen erhalten, die nicht zwangsweise vorher definiert werden müssen. Außerdem ist es möglich, schnell neue Relationen hinzuzufügen oder alte zu verändern.

Für diese Arbeit wird die am Fraunhofer SCAI bereits etablierte Graphdatenbank *Neo4j* verwendet. Die zugehörige Query Language ist dabei *Cypher*. [14] [15]

2.3 Ontologien

Unter Ontologien werden computergestützte Strukturen verstanden, die mit eindeutigem und beschränktem Vokabular geordnete Entitäten und Relationen eines Fachbereichs beschreiben. Gegebene Daten können dann um diese Ontologie herum angesiedelt und damit in einen fachlichen Kontext gesetzt werden. Die Entitäten werden dabei als Klassen verstanden, die innerhalb einer oder mehrerer Hierarchien von allgemein bis spezifisch angeordnet werden.
Diese Klassen sind die Grundelemente der Ontologie und stellen den Typus eines Elements dar. Sie besitzen einen eindeutigen Identifier, der jedoch meist keiner speziellen Semantik unterliegt, das heißt keine Referenz auf Klassennamen oder Definitionen aufweist, um Stabilität und Erweiterbarkeit der Ontologie zu ermöglichen. Jede Klasse enthält eine präzise Definition, die sie klar von anderen Klassen abgrenzt.
Die Relationen speichern die domänenspezifischen Informationen in Form von interentitären, gerichteten Verbindungen. Sie bilden die oben genannten Hierarchien, die jedoch nicht zwangsweise eine Baumstruktur aufweisen, sondern im Allgemeinen gerichtete azyklische Graphen sind, um mehrere Väter für eine Klasse zu ermöglichen. Zusätzlich können aber auch weitere Beziehungen nicht-hierarchischer Natur zwischen Klassen über Relationen ausgedrückt werden. Relationen haben, ähnlich wie Klassen, Spezifikationen, wie z. B. einen eindeutigen Identifier, einen Namen und eine Hierarchie. Darüber hinaus können sie reflexiv und transitiv sein sowie eine Inverse besitzen.
Des Weiteren enthält eine Ontologie Metadaten zu Klassen und Relationen. Damit sind unter anderem alternative Identifier, Synonyme und Querverweise auf Klassen anderer Ontologien gemeint. [16]

2.4 Komplexitätstheorie und Laufzeit

Um Algorithmen in der Informatik zu klassifizieren, werden obere Schranken für die Laufzeit im ungünstigsten Fall gesucht. Diese hängt von der Eingabeinstanz $\mathcal{I}$, der Ko-

dierungslänge l, dem Computer $\mathcal{C}$, der Programmiersprache $\mathcal{L}$ sowie der Implementierung $\mathcal{A}$ ab. Zur Vereinfachung wird ein generischer Computer und eine allgemeine Programmiersprache betrachtet, so dass die Laufzeit nur noch von $\mathcal{I}$ und l abhängt. Dabei wird die Anzahl der Elementaroperationen des gegebenen Algorithmus in Abhängigkeit von $\mathcal{I}$ und l gezählt, die der Computer ausführt. Für die oberen Schranken unterscheidet man daher hauptsächlich zwischen konstanten, polynomiellen und exponentiellen Funktionen. Die Laufzeitfunktion beschreibt dabei, in welchem Maße die Laufzeit mit steigender Eingabelänge zunimmt, und stellt dafür eine asymptotische obere Schranke dar. [13] [17]

Definition 2.4.1 *Für eine Funktion $g(n)$ bezeichnet $f \in \mathcal{O}(g)$ alle Funktionen f, für die es positive Konstanten C und n_0 gibt, so dass $0 \leq |f| \leq C \cdot g(n)$ für alle $n \geq n_0$ gilt. [13]*

Theorem 2.4.2 *Seien $T_0(n), T_1(n)$ die Laufzeiten zweier Programmstücke P_0, P_1. Sei weiterhin $T_0(n) = \mathcal{O}(f(n))$ und $T_1(n) = \mathcal{O}(g(n))$ für zwei Funktionen f, g. Dann gilt:*

- $T_0(n) + T_1(n) = \mathcal{O}(max\{f(n), g(n)\})$
- $T_0(n) \cdot T_1(n) = \mathcal{O}(f(n) \cdot g(n))$.

Beweis. Ersteres ist nach 2.4.1 sofort klar: Sei $n > max\{n_0, n_1\}$. Dann gilt:

$$\begin{aligned} & T_0(n) = \mathcal{O}(f(n)) \ \wedge \ T_1(n) = \mathcal{O}(g(n)) \\ \Leftrightarrow \ & \exists C_0, C_1 > 0 \wedge n_0, n_1 \in \mathbb{N} : T_0(n) \leq C_0 \cdot f(n) \wedge T_1(n) \leq C_1 \cdot g(n). \end{aligned} \tag{2.1}$$

Sei $C^+ = max\{C_0, C_1\}$. Dann folgt aus Gleichung 2.1:

$$T_0(n) + T_1(n) \leq C_0 \cdot f(n) + C_1 \cdot g(n) \leq C^+ \cdot (f(n) + g(n)). \tag{2.2}$$

Daraus folgt mit 2.4.1 die erste Behauptung.

$$\begin{aligned} T_0(n) \cdot T_1(n) &\leq C_0 \cdot f(n) \cdot C_1 \cdot g(n) \\ &\leq C_0 \cdot C_1 \cdot f(n) \cdot g(n) \\ &\leq C^* \cdot f(n) \cdot g(n) \end{aligned} \tag{2.3}$$

Für $C^* = C_0 \cdot C_1 > 0$ folgt die zweite Behauptung dann ebenfalls mit 2.4.1. □

Definition 2.4.3 *Ein algorithmisches Problem liegt in der Komplexitätsklasse $\mathcal{P}$, wenn es sich von einem Algorithmus in polynomieller Zeit lösen lässt. [17]*

Definition 2.4.4 *Ein Entscheidungsproblem liegt in der Komplexitätsklasse $\mathcal{NP}$, wenn Ja-Antworten sich von einer nicht-deterministischen Turingmaschine in polynomieller worst-case Laufzeit überprüfen lassen. [17]*

3 | Gegebenheiten

Im Rahmen der Studien zur Integrativen Datensemantik für Neurodegenerationsforschung sind von verschiedenen Instituten klinische Daten zu Patienten, die an Alzheimer erkrankt sind, gesammelt worden. Diese liegen in Form einer NoSQL-Mongo-Datenbank vor. Da das IDSN-Projekt sich noch in der Entwicklung befindet, ist davon auszugehen, dass es noch zu Modifikationen des zugrundeliegenden Datenmodells kommen wird, die aber das Prinzip der in dieser Arbeit präsentierten Lösung nicht beeinflussen sollten.
Das Datenmodell enthält zunächst allgemeine Variablen zu einem Patienten, z. B. Ort der Messung (*site*), Geschlecht (*sex*) und ApoE-Genotypen. Darüber hinaus werden als Variablen eine Vielzahl an neuropsychologischen Testungen (NPT), Störungen (Disturbances), Laborwerten (LAB), Marker aus dem Liquor-Speziallabor (LIQ), Scores, Spinozerebelläre Ataxie Charakterisierungen (SCA) und Diagnosen gegeben.
Aufgrund der Sensitivität der personenbezogenen klinischen Daten wird innerhalb dieser Arbeit mit einem Sample artifizieller Daten gearbeitet. Sie decken nicht das vollständige später vorgestellte Datenschema (vgl. Abbildung 6.2) ab. Das vereinfachte Modell reicht jedoch aus, um das Prinzip der Arbeit darzulegen, und ist anhand des Datenschemas problemlos auf einen vollständigen Datensatz übertragbar bzw. erweiterbar.
Dieser Arbeit liegt außerdem die Masterarbeit [3] von Andreas Stefan zugrunde, in der auf der Basis eines erstellten Datenschemas ein effizienter Polyglot Persistence Entwurf zur Speicherung und Abfrage eines Knowledge Graphen anhand der in der *PubMed* Datenbank enthaltenen Dokumente vorgestellt wird. Dort wird ausgehend von Fragestellungen aus dem biomedizinischen Bereich unter Rückgriff auf das in Abbildung 3.1 gegebene Datenschema ein Knowledge Graph erstellt. [3]
Datenschema und praktische Umsetzung basieren auf klinischen Fragestellungen, die das Ergebnis mehrerer Befragungen von Mitarbeitern aus dem biomedizinischen Bereichs sind. Sie bilden die Grundlage für die Anforderungen an den Knowledge Graphen.

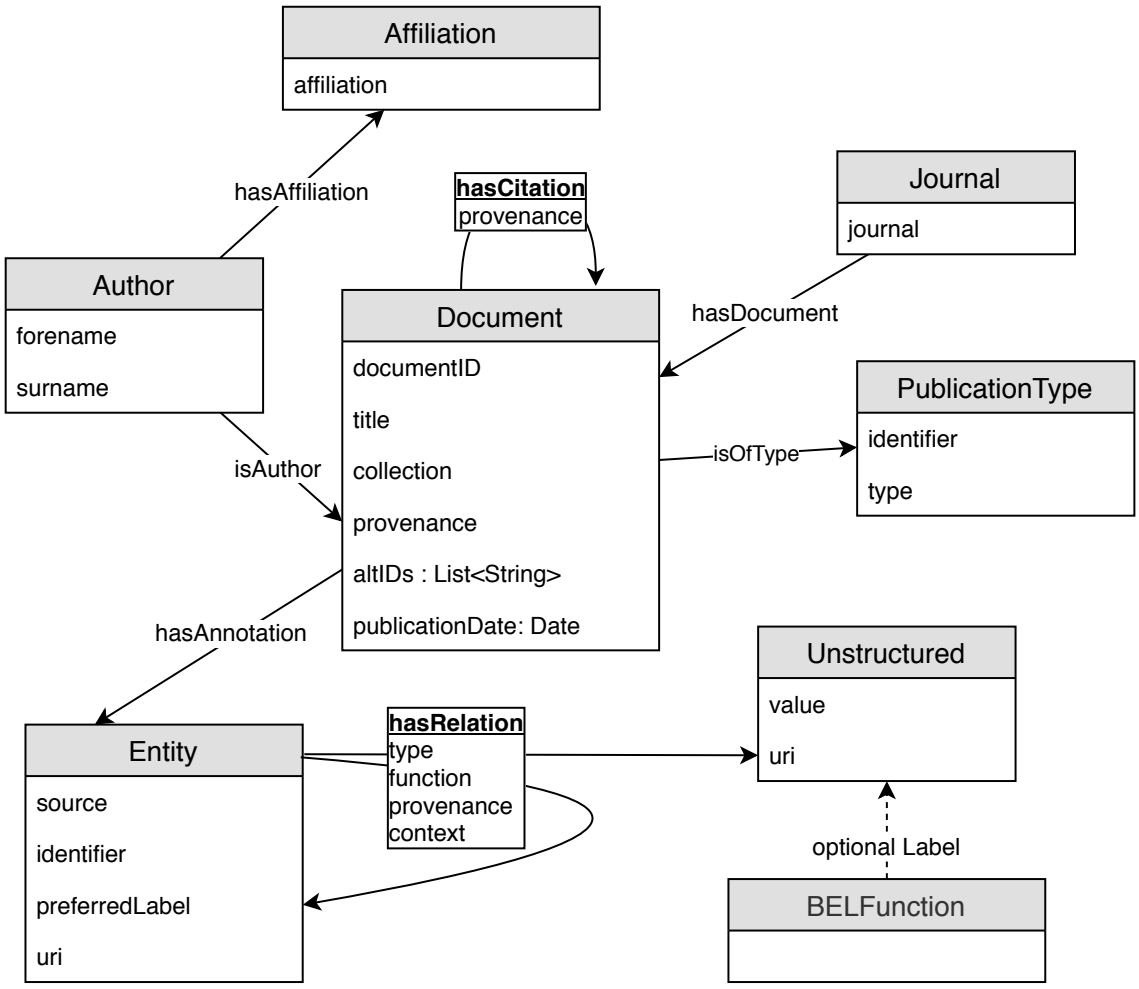

Abbildung 3.1: Formales Datenschema des Knowledge Graphen der *PubMed* Datenbank

4 | Problemstellung

Ausgehend von den gegebenen Daten stellt sich die Frage, wie man sie in einem geeigneten Schema anordnen und damit effizient in einem Knowledge Graphen abspeichern kann, so dass die klinischen Fragestellungen mithilfe von Graph Querys beantwortet werden können. Dabei soll der spätere Anschluss an den Graphen der *PubMed* Datenbank in [3] ermöglicht werden. Zu diesem Zweck soll das dort verwendete Schema, das in Abbildung 3.1 dargestellt wurde, zu einem neuen Schema erweitert werden. Die vorgegebenen Daten aus der IDSN-Studie, bzw. die hier verwendeten artifiziellen, sollen danach mithilfe des neuen Schemas in einen Knowledge Graphen überführt werden. Dieser soll über ein Mapping zwischen den Entitäten und mehreren Ontologien mit Kontext angereichert werden. Hierbei ist mit Mapping eine Abbildung $M : E(D) \rightarrow E(O)$ gemeint, wobei $E(D)$ die Entitäten der gegebenen Daten und $E(O)$ die Entitäten der verwendeten Ontologien beschreiben. Das Mapping besteht also aus Kanten, die im Knowledge Graphen eine logische Relation zwischen biomedizinischen Daten und Lexika schaffen.

5 Klinische Fragestellungen

Um das gegebene Datenschema sinnvoll erweitern zu können, werden zwei Grundpfeiler benötigt. Der erste ist das biomedizinische Datenmodell, das in Form einer Excel-Tabelle zur Verfügung steht. Der zweite sind die klinischen Fragestellungen, die mittels des Graphen beantwortet werden sollen. Diese werden innerhalb dieser Sektion geordnet und kategorisiert, um später in Abschnitt 8.3 ihre Effizienz analysieren und mittels Komplexitätstheorie Vergleichswerte betrachten zu können. Als Grundlage dafür dienen erneut die Quellen [3] und [18]. Dort werden ebenfalls biomedizinische Fragestellungen untersucht und in [18] optimiert, wozu sechs verschiedene Literaturquellen herangezogen werden, die teilweise unterschiedliche und teilweise gemeinsame Kategorien oder Klassen für graphbasierte Querys anführen. In [3] wird mit Rückbezug auf diese Literaturquellen und unter Verwendung eigener Kriterien des Autors eine Hierarchie für die Klassifizierung der Querys erstellt.

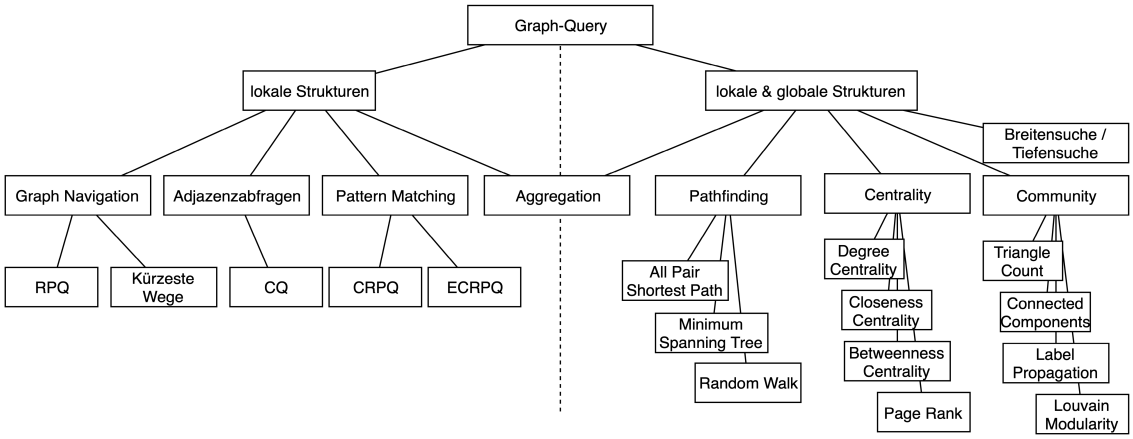

Abbildung 5.1: Kategorisierungshierarchie für Graph Querys

5.1 Kategorien

Das Schema in Abbildung 5.1 unterscheidet zunächst zwei verschiedene Formen der Querys. Die, die sich auf lokale Strukturen des Graphen beziehen, erhalten einen Einstiegspunkt und suchen lokal in einer k-Nachbarschaft dazu (vgl. 2.1.10). Die zweite Kategorie

enthält sowohl globale als auch lokale Querys. Theoretisch können die dort eingeordneten Querys den gesamten Graphen durchsuchen. Allerdings lassen sie sich im Anwendungsfall durch ausschließliche Betrachtung eines Subgraphen oder gegebener Einstiegs- und Austrittspunkte so einschränken, dass nur lokale Strukturen durchsucht werden. Als Beispiel dafür wird in [3] die Unterkategorie Aggregation angeführt. Dort lässt sich sowohl nach dem durchschnittlichen Knotengrad im gesamten Graphen $G = (V, E)$ als auch nach dem Knotengrad eines einzelnen Knoten $v \in V$ suchen. Unter lokale Strukturen fallen dann die drei Bereiche Graph Navigation, Adjazenzabfrage und Pattern Matching. Unter der ersten Gruppierung sind dann folgende Kategorien genannt:

- RPQ: Gesucht wird ein Knotenpaar u, v, so dass ein Pfad zwischen den Knoten existiert, dessen Sequenz von Kantenlabeln einem vorgegebenen Muster gleicht und als ein regulärer Ausdruck gegeben ist. Eine RPQ hat damit die folgende Form: $ans(u, v) \leftarrow (u, r, v)$. r ist hierbei der reguläre Ausdruck aus einem endlichen Alphabet Σ und mit $ans(u, v)$ ist die Rückgabe der Query bezeichnet. [19]

- Kürzester Weg: Es wird ein kürzester Pfad (vgl. 2.1.5) zwischen einem gegebenen Knotenpaar u, v gesucht.

Bei Adjazenzabfragen lassen sich die *conjunctive queries*, abgekürzt CQ, finden. Diese haben über einem endlichen Alphabet Σ die Form $ans(z_1, ...z_n) \leftarrow \bigwedge_{i=1}^{m}(x_i, a_i, y_i)$. Dabei ist jedes x_i und y_i eine Knotenvariable oder eine Konstante. Des Weiteren gilt: $a_i \in \Sigma$ und z_i mit $i \in 1, ..., n$ ist ein Knoten x_j oder y_j mit $j \in 1, ..., m$. Ein Beispiel für eine Anwendung wäre die Suche nach einem Patienten, der zwei verschiedene Diagnosen erhalten hat.
Unter der Gruppierung Pattern Matching befinden sich dann CRPQs und ECRPQs. Erstere sind dabei genauso wie CQs definiert, nur dass statt des Ausdrucks $a_i \in \Sigma$ ein regulärer Ausdruck $r_i \in \Sigma$ verwendet wird. [19]
In der zweiten Kategorie, die lokale und globale Strukturen betrachtet, findet man vier Unterkategorien. Aggregation wurde oben bereits thematisiert. Pathfinding betrachtet folgende drei Gruppen:

- All Pair Shortest Path: Für einen Graphen $G = (V, E)$ wird zwischen jedem Knotenpaar $u, v \in V$ ein kürzester Pfad gesucht. [20]

- Minimum Spanning Tree: Für den Graphen wird ein minimaler Spannbaum (vgl. 2.1.9) gesucht.

- Random Walk: Für einen Graphen $G = (V, E)$ wird von einem Startknoten $v_{k_0} \in V$ aus ein zufälliger adjazenter Knoten v_{k_1} ausgewählt und dem Pfad $P = (v_{k_0})$ hinzugefügt. Von diesem wird erneut ein zufälliger benachbarter Knoten ausgewählt. Dies setzt sich fort, bis eine zuvor festgelegte Pfadlänge für P erreicht ist. [20]

Die nächste Gruppe *Centrality* betrachtet, wie der Name schon sagt, verschiedene Zentralitätsmaße:

- Degree Centrality: Degree Centrality misst die Knotengrade der Knoten $v \in V$ eines Graphen $G = (V, E)$. Je zentraler ein Knoten ist, desto höher ist sein Grad. [20]
- Closeness Centrality: Es werden für einen Graphen $G = (V, E)$ die Knoten $v \in V$ gesucht, die möglichst zentral in G liegen. Dafür werden die kürzesten Pfade von v zu allen anderen Knoten $u \in V, u \neq v$ berechnet, aufsummiert und die Summe invertiert: $C(v) = \frac{1}{\sum_{n=0}^{n-1} d(v,u)}$ mit der Distanz $d(v,u)$ des kürzesten Pfades zwischen v und u und $n = |V|$. Für den Vergleich verschieden großer Graphen kann dies noch normalisiert werden, indem es mit n multipliziert wird. [20]
- Betweenness Centrality: Dies ist ein Maß für die Bedeutung eines Knoten $v \in V$ für den gesamten Graphen $G = (V, E)$. Es wird gemessen, auf wie vielen kürzesten Pfaden zwischen zwei beliebigen anderen Knoten $s, t \in V$ der Knoten v liegt. [20]
- Page Rank: Ursprünglich wurde der Begriff von Google geprägt und gibt für eine Webseite Auskunft darüber, wie stark diese auf anderen Seiten verlinkt ist. Je größer die Zahl und je wichtiger die Seiten, die diese verlinken, desto höher der Page Rank. Übertragen auf die Graphentheorie lässt sich also die Relevanz eines Knoten an den hin- und wegführenden Kanten messen. [20]

Ebenfalls behandelt werden Breiten- und Tiefensuche. Erstere entdeckt systematisch Kanten in einem Graphen $G = (V, E)$ von einem Quellknoten $s \in V$ aus beginnend. Dabei berechnet die Suche sowohl die Entfernung des Quellknotens zu allen entdeckten Knoten als auch einen Baum mit der Wurzel s. Der Algorithmus entdeckt für ein wachsendes $k \in \mathbb{N}$ zunächst alle Knoten mit der Entfernung k von der Wurzel s, bevor er Knoten mit Entfernung $k+1$ entdeckt. Breitensuche ist also eine weitere Möglichkeit, kürzeste Pfade zu finden. [13]
Die Tiefensuche agiert sehr ähnlich. Der Unterschied besteht darin, dass bei einem neu entdeckten Knoten zunächst die noch nicht entdeckten Nachbarn besucht werden. Erst wenn ein solcher Pfad in einem Blatt endet, geht der Algorithmus zurück zum letzten Knoten, der noch unentdeckte Nachbarn hat. [13] Schlussendlich wird die Kategorie Community in folgende Teilbereiche aufgeteilt:

- Triangle Count: Sei $G = (V, E)$ ein Graph. Der Algorithmus zählt für jeden Knoten $v \in V$ die Anzahl der Dreiecke, die durch den Knoten v laufen. Ein Dreieck ist dabei eine Menge von drei Knoten, so dass alle Knoten paarweise Kanten zueinander haben. [20]
- Strongly Connected Components: Nach 2.1.12 werden stark verbundene Komponenten im Graphen gesucht.

- Label Propagation: Sei $G = (V, E)$ ein Graph, in dem jeder Knoten ein Label hat und zwei Knoten je paarweise verschiedene Labels haben. Der Label Propagation Algorithmus weist aufgrund der vorhandenen Kantenstruktur einem Knoten das Label zu, das die meisten seiner Nachbarn haben. Durch viele Iterationen werden Communitys gleichen Labels in Graphen gefunden. [21]

- Louvain Modularity: Mit dieser Methode lassen sich Communitys, also durch Kanten eng verbundene Teilmengen der Knoten, in Graphen finden. Dabei gibt die Modularität mit einem Skalarwert zwischen -1 und 1 die Dichte der Kanten innerhalb der Community an. [22]

Viele der Implementationen der Algorithmen in *Neo4j* weisen bei großen Netzwerken schlechte Laufzeiten auf. Es ist deutlich effizienter, die Algorithmen auszulagern und nur mit einfachen Querys mit der Graphdatenbank zu kommunizieren [18]. Dies ist jedoch nicht mehr Teil dieser Arbeit und das Problem wird später in Kapitel 10 daher nur noch kurz angesprochen. Aus diesem Grund wird im späteren Verlauf nur eine kleine Auswahl dieser Algorithmen tatsächlich angewendet und sich meistens nur auf Querys beschränkt.

5.2 Präsentation und Einordnung der Fragestellungen

Dieser Abschnitt beschäftigt sich mit den einzelnen biomedizinischen Fragestellungen. Sie wurden durch Gespräche mit klinischem und biomedizinischem Fachpersonal gesammelt. Es wird analog zu [3] vorgegangen und zunächst Input und Output für alle Fragestellungen beschrieben. Da der Graph nur eine Teilmenge der eigentlich vorgesehenen Daten enthält, müssen die klinischen Fragestellungen ersetzt werden. Es soll jedoch die Kategorie der Fragestellung aus Abschnitt 5.1 beibehalten werden. Aus einem biomedizinischen Blickwinkel sind diese Fragen unter Umständen nicht sinnvoll, aus der Perspektive der Informatik sind sie jedoch isomorph zu den ursprünglichen. Die Fragestellungen werden bei der Auflistung so nummeriert, dass sie mit den späteren Querys korrelieren. Zusätzlich wurden noch die Querys 13-15 hinzugefügt, um die Algorithmen von *Neo4j* zu testen. Sie haben jedoch keine biomedizinische Frage aus den Gesprächen als Grundlage. Die in *Cypher* formulierten zugehörigen Querys befinden sich im Anhang der Arbeit unter Abschnitt A.1.

Tabelle 5.1: Klinische Fragestellungen mit Ersatz

#	Fragestellung	Ersatz
1	Welche Patienten haben vollständige NPTs?	Welche Patienten haben die meisten verschiedene HGNC-Werte?

Tabelle 5.1: Klinische Fragestellungen mit Ersatz (Fortsetzung)

2	Welche Messwerte werden am häufigsten im Kontext der Diagnose {Diagnose1} gefunden?	Welche Patienten werden am häufigsten im Kontext einer Risikogruppe {RiskGroup1} gefunden?
3	Welche Werte werden gleichzeitig erhoben?	Welche HGNC-Werte werden bei Besuch {Besuch1} am häufigsten erhoben?
4	Wie sieht die zeitliche Reihenfolge der Messwerte der Entität {Entität1} und der Risikogruppe {RG1} für einen Patienten {Patient1} aus?	Wie sieht die zeitliche Reihenfolge der HGNC-Werte {Entität1} und der Risikogruppe {RG1} für einen Patienten {Patient1} aus?
5	Wie viele Patienten haben innerhalb der ersten zwei Tage nach ihrem Besuch eine Diagnose erhalten?	Wie viele Patienten haben während des ersten Besuchs HGNC-Werte erhalten?
6	Welche Patienten mit Diagnose {Diagnose2} haben {Anzahl1} Tage vorher NPTs durchlaufen?	Welche Patienten mit Diagnose {Diagnose2} bei Besuch {Besuch2} hatten beim vorherigen Besuch den HGNC-Wert {HGNC_Wert1}?
7	Kommen Personen des Alters {Alter1} häufiger zur Untersuchung als andere?	Kommen Personen des Geschlechts {sex1} häufiger zur Untersuchung als andere?
8	Welche gemeinsame Entität {Entität2} lässt sich bei Patienten ohne Diagnose finden?	Welche Patienten haben genau {Anzahl2} verschiedene Diagnosen und welchen Risikogruppen gehören sie an?
9	Wie oft taucht ein Allel-Tupel {Allel-Tupel1} unter den Patienten auf?	Welche Risikogruppe kommt unter den Patienten am häufigsten vor?
10	Welche Literatur {Literatur1} findet sich zu Patient {Patient2} mit Diagnose {Diagnose3}?	Welche HGNC-Werte {HGNC_Wert2} finden sich zu Patient {Patient2} mit Diagnose {Diagnose3}?
11	Wie viele Patienten haben NPT {NPT1} durchlaufen und gleichzeitig LAB-Wert {LAB_Wert1}?	Wie viele Patienten haben Diagnose {Diagnose4} und gleichzeitig HGNC-Wert {HGNC_Wert3}?
12	Wie viele Patienten haben Störung {Disturbance1} und welches Geschlecht {sex2} haben sie?	Wie viele Patienten haben Diagnose {Diagnose5} und welchem Geschlecht gehören sie an?

Tabelle 5.1: Klinische Fragestellungen mit Ersatz (Fortsetzung)

13	-	Was ist der kürzeste Weg zwischen Entität {Entität4} und Entität {Entität5} und was liegt auf diesem Weg?
14	-	Welcher Patient verbindet Entitäten am stärksten?

Tabelle 5.2: Kategorisierung der Querys

Query	Struktur	Kategorie	Input	Output	Zugriff auf Attribute	Datentyp der Attribute	berücksichtige Knoten- und Kantentypen	Einstiegspunkt
1	global	RPQ	-	Patient, Anzahl	Entity.source, Patient.patient	2xString	Patients, Entity, hasRelation	mehrdeutig
2	lokal	RPQ	high	Anzahl	Entity.identifier	String	Patients, Entity, hasRelation	eindeutig
3	lokal	RPQ	HGNC, 2	HGNC-Werte, Anzahl	Entity.source, Entity.preferredLabel, unstructured.value	3xString	Entity, Unstructured, Patients, hasRelation, hasValue	eindeutig
4	lokal	CRPQ	22504, HGNC	HGNC-Werte, Risikogruppe, Besuchszahl	Patient.patient, Entity.source, unstructured.value, Entity.preferredLabel, Entity.riskgroup	5xString	Unstructured, Patients, Entity, hasPatient, hasRelation, hasValue	eindeutig
5	lokal	RPQ	0, HGNC	Anzahl Patienten	Entity.source, unstructured.value	2xString	Entity, Patients, Unstructured, hasRelation, hasValue	eindeutig
6	lokal	CRPQ	2, Alzheimer's disease, 1, 35357	Patientenliste	Entity.preferredLabel, unstructured.value, Entity.identifier,	4xString	Entity, Patients, Unstructured, hasValue, hasRelation	eindeutig

Tabelle 5.2: Kategorisierung der Querys (Fortsetzung)

7	lokal	RPQ	6	Geschlecht, Anzahl	unstructured.value, sex.sex	2xString	Patients, sex, Unstructured, hasSex, hasValue	mehrdeutig
8	global	CRPQ	HGNC, rg, 2	Patienten, Risikogruppe, Anzahl HGNC	Entity.source, Entity.category, Entity.identifier, Patient.patient	4xString	Entity, Patients, hasPatient, hasRelation	mehrdeutig
9	lokal	Degree Centrality	-	Risikogruppen, Anzahl Patienten	Entity.identifier	String	Entity, Patients, hasRelation	mehrdeutig
10	lokal	ECRPQ	12864, Alzheimer's disease	HGNC-Werte	Entity.source Entity.preferredLabel, Patient.patient	3xString	Entity, Patients, hasRelation	eindeutig
11	lokal	RPQ	37785, DOID:14332	Patientenliste, HGNC-Werte, Diagnose	Entity.identifier, Patient.patient	3xString	Entity, Patients, hasRelation	eindeutig
12	lokal	RPQ	DOID:004002	Patientenliste, Geschlechter	sex.sex, Entity.identifier, Patient.patient	3xString	Entity, Patients, sex, hasSex, hasRelation	eindeutig
13	lokal	Kürzeste Wege	DOID:0040005, 41022	Pfad	Entity.identifier	2xString	alle Knoten, hasRelation	eindeutig
14	global	Betweenness Centrality	-	Patient	Entity.preferredLabel	String	Patients, Entity, hasRelation	mehrdeutig

6 | Aufbau und Struktur des Graphen

Um das gegebene Problem zu lösen, müssen die klinischen Daten des gegebenen Datenmodells anhand eines geeigneten Schemas angeordnet werden. Dieses Schema wird dann verallgemeinert, um als Datenschema für den Graphen in *Neo4j* zu fungieren.
Es wird im Folgenden angenommen, dass der Datensatz vollständig ist, also zu jedem Patienten alle Daten vorhanden sind. Für die artifiziell generierten Daten ist dies auch insofern zutreffend, dass zwar nur ein Teil der möglichen Gesamtdatenmenge betrachtet wird, aber jeder Patient zu jedem Attribut Werte hat. In den Realdaten wird der Datensatz zu großen Teilen nur lückenhaft gegeben sein, da nicht jeder Patient jeden klinischen Test durchläuft. Dieser Umstand ist aus Sicht der Informatik für diese Arbeit nicht weiter relevant. Wie später anhand der gegebenen HGNC-Werte demonstriert wird, können leere Teildatensätze übersprungen werden.
Zunächst werden also die Variablen aus dem Datenmodell betrachtet. Sie müssen in Klassen und Relationen sowie zugehörige Attribute aufgeteilt werden, um später in dem Graphen unkomplizierte Abfragen innerhalb möglichst vieler Untergruppen der Patienten bzw. Subgraphen zu ermöglichen. Hierzu werden für die meisten Variablen eigene Klassen erstellt, anstatt sie in Attributen zu speichern. So erhält beispielsweise der Patient, der als Zentrum seines personenspezifischen Datensatzes gelten kann, sein Geschlecht oder seinen genetisch vorgegebenen ApoE-Typus nicht als Attribut der Klasse Patient, sondern es werden für beide separate Klassen erstellt. Durch dieses Vorgehen lassen sich später Subgraphen, wie zum Beispiel Patienten eines bestimmten Geschlechts oder eines bestimmten Genotyps, einfacher betrachten. Man kann solche Untergruppen also als eigene Ankerpunkte des Graphen verstehen, die als Quelle einer ganz bestimmten Teilmenge der Gesamtdaten dienen können. Hinzu kommen noch zwei weitere Vorteile: Einerseits ist die Abfrage nach Knoten in *Neo4j* effizienter als die nach Attributen von Knoten, da für die Erreichbarkeit der Attribute eine zusätzliche Datei gelesen werden muss. Dies erfordert weiteren Rechenaufwand und verlangsamt die Verwendung. [18] Andererseits ist es durch die Auslagerung der Attribute in eigene Klassen und damit Knoten möglich, differenziertere und spezifischere Relationen zur Beschreibung der Zusammenhänge zwischen Knoten zu verwenden. Darüber hinaus wäre es sogar möglich, diese Klassen selbst mit weiteren Attributen anzureichern, was im Rahmen dieser Arbeit aber nicht geschieht.

6.1 Klassen

Zunächst sollen die verwendeten Klassen des in Abbildung 6.2 präsentierten Modells vorgestellt werden. Die erste und zentrale Klasse bildet der Patient selbst. Sie enthält bis auf eine ID keine weiteren Attribute, da diese aus oben genannten Gründen ausgelagert werden.
Die zweite betrachtete Klasse ist das Geschlecht. Sie ordnet die Patienten in männliche und weibliche Untergruppen ein. Grundsätzlich sind hier natürlich beliebig viele verschiedene Geschlechter unterscheidbar.
Die ditte Klasse ist der ApoE-Risikotyp. Er beschreibt einen genetischen Zustand, der in enger Verbindung zu Risiken für eine Alzheimererkrankung steht. Zu jedem Patienten ist bzw. sind dabei entweder ein 2-Tupel $(\varepsilon i, \varepsilon j)$, $i, j \in \{1,2,3,4\}$ oder zwei 2-Tupel $(rs429358, rs7412)$ aus SNPs, die wiederum ein $\varepsilon i, i \in \{1,2,3,4\}$ bilden, gegeben. Die einzelnen Allele bestehen dabei wiederum aus einem Tupel von SNPs, die entweder vom Typ C oder T sein können. Zur Veranschaulichung ist das kleine Modell in Abbildung 6.1 dargestellt.
Aus dieser Konstruktion ergibt sich die Klasse der Risikogruppe, die patientenspezifisch unterschiedliche Ausprägungen haben kann. Diese Ausprägungen sind implizit durch die Risikotypen vorgegeben. Es werden drei verschiedene Kategorien mit folgenden Bezeichnungen unterschieden:

- low-risk: Patienten, deren Risikotyp einem Tupel $(\varepsilon i, \varepsilon j)$, $i, j \in \{1,2,3\}$ entspricht.
- medium-risk: Patienten, deren Risikotyp einem Tupel $(\varepsilon i, \varepsilon j)$ mit $i = 4, j \in \{1,2,3\}$ oder $i \in \{1,2,3\}, j = 4$ entspricht.
- high-risk: Patienten, deren Risikotyp dem Tupel $(\varepsilon 4, \varepsilon 4)$ entspricht.

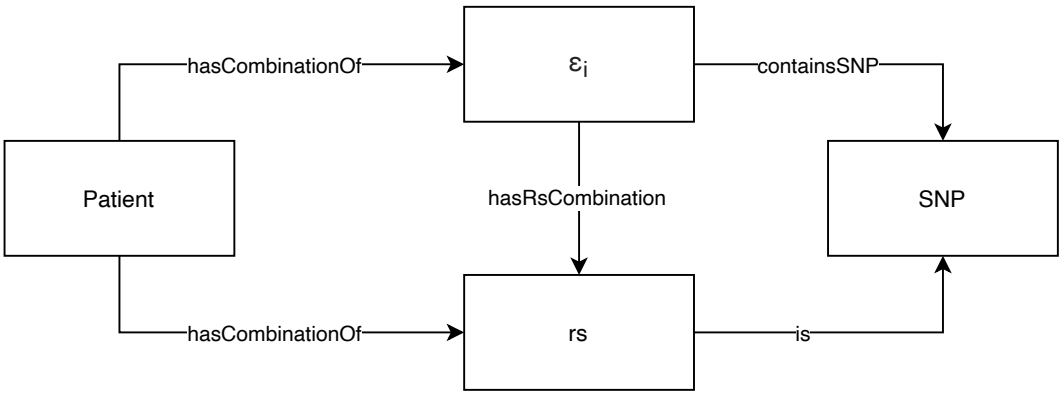

Abbildung 6.1: Epsilon und rs-Code Struktur

Im hier gegebenen Fall wird bei den artifiziellen Daten mit dem ε-Tupel gearbeitet. Grundsätzlich lässt sich das Modell aber natürlich auch auf die Allele übertragen, da eine

eindeutige, bijektive Abbildung zwischen der Menge der für das ApoE relevanten Allele und der ε-Tupel existiert. [4]
Als nächstes werden die Messwerte klinischer Untersuchungen und Studien betrachtet. Darunter fallen LAB, LIQ, NPT, ApoE, SCA und Diagnosen. Sie werden in der Klasse der `attributes` gesammelt und gehören einer übergeordneten Kategorie, dem unter `Unstructured` gespeicherten *topic*, an. Den `attributes` werden im Rahmen der Untersuchungen der Patienten Werte zugewiesen. Diese werden wie die *topics* als separate Objekte der Klasse `Unstructured` angelegt und über Relationen mit der Klasse der `attributes` verknüpft. Die Werte an sich sind zunächst mit keiner Einheit versehen. Es wird eine weitere Klasse `unit` erstellt, die zu jedem gemessenen Wert die zugehörige Einheit speichert. Dies ermöglicht, wie oben bereits erwähnt, erneut eine größere Auswahl der Betrachtungs- und Befragungsmöglichkeiten für den Graphen. Außerdem kann man die Einheiten über geeignete Relationen an eine passende Ontologie, beispielsweise UCUM (Unified Code for Units of Measure, [23]), anschließen.
Weiterhin können einem Patient eine oder mehrere Diagnosen zugeordnet sein. Auch diese werden als eigenständige Klasse festgelegt und sind mit einem *diagnosis code* hinterlegt. Dieser stimmt mit den *diagnosis* codes der *Disease Ontology* in seiner bezeichnenden Verwendung überein, wird also auch in fachbezogener Literatur standardmäßig verwendet. [16]
Darüber hinaus gibt es noch die Klasse `time`. Jeder Messwert und jede Untersuchung ist mit einem Zeitstempel versehen. Dadurch lässt sich zu einer beliebigen Teilmenge der Gesamtdaten eine zeitliche Hierarchie feststellen und unter Umständen lassen sich sogar Trends bezüglich des Verlaufes der Krankheit erkennen. Die Zeitstempel können dabei das Datum einer Untersuchung aber auch rückwirkend das Datum des ersten Auftretens eines Symptoms sein.
Die letzten verwendeten Klassen sind `source` und `sourceAll`. `source` dient dabei als Quelle eines Datensatzes eines bestimmten klinischen Instituts des DZNE. Dabei wird auch der Ort der Erhebung der Daten gespeichert. Dies dient zur lokalen Abgrenzung der Daten untereinander und erlaubt es, Datensätze verschiedener Herkunft in einem Graphen zu vereinen, ohne dass sie ihre Zugehörigkeit verlieren.
Die von `source` erbende Klasse `sourceAll` ist nahezu identisch mit dieser, jedoch darüber hinaus noch mit einem Attribut *provenance* versehen. Dadurch lassen sich innerhalb eines gegebenen Datensatzes Zugehörigkeiten verschiedener Instanzen von Klassen zueinander eindeutig definieren und festhalten. Beispielsweise lässt sich der Bezug von Objekten A,B und C zueinander festhalten, wenn Patient A zum Zeitpunkt B die Diagnose C erhalten hat. Genauer wird darauf in dem folgenden Teilkapitel eingegangen.

6.2 Relationen

Die oben genannten Klassen stehen in verschiedenen Relationen zueinander. Durch die Auslagerung der Attribute und die dadurch gesteigerte Anzahl der Knoten erhöht sich allerdings auch die der Kanten. In einem gerichteten Graphen $G = (V, E)$ mit einer Knotenanzahl $n = |V|$ gilt für die Anzahl der Kanten im ungünstigsten Fall $|E| = n \cdot (n-1)$, da jeder Knoten $v \in V$ eine Kante zu jedem anderen Knoten $u \in V, u \neq v$ haben kann.
Die Benennung der Kanten erfolgte in Anlehnung an Dublin Core (vgl. [24]). Dabei handelt es sich um einfache Standards der Dublin Core Metadata Initiative für Datenformate von Dokumenten oder Objekten. Das dort aufgelistete Vokabular ist teilweise durch bereichspezifischere Ausdrücke ersetzt worden, jedoch unter Beibehaltung der grundsätzlichen Struktur. Als Beispiel hierfür kann der vorgegebene Term *hasFormat* genannt werden, zu dem dann strukturerhaltend die Relation *hasSex* erstellt wird, um die Patient-Geschlecht-Beziehung darzustellen.
Neben der Bezeichnung bzw. des Labels einer Kante werden noch weitere Informationen benötigt. Jede Relation zweier Klassen erhält einen Zeitstempel, der im Attribut *time* gespeichert wird. Dieser kann explizit durch ein konkretes Datum oder auch implizit durch die Beziehung der Klassen und Relationen zueinander gegeben sein. Zusätzlich erhalten die Kanten noch ein Attribut *provenance*. Dieses korreliert mit dem gleichnamigen Attribut des `sourceAll`-Knoten und weist Beziehungen zwischen mehreren Klassen eine interne Zugehörigkeit zu. Das Attribut *provenance*, egal ob innerhalb einer Knotenklasse oder als Attribut einer Relation, verhindert also Datenambiguität.

6.3 Datenschema

Auf Basis der obigen Überlegungen wurde das Schema in Abbildung 6.2 entworfen. Für die Graphdatenbank ist dieses Schema jedoch zu detailliert aufgeschlüsselt. Hier soll eine Erweiterung des bereits vorhandene Datenschemas von A. Stefan erstellt werden, die das bisher Konzipierte (Abbildung 6.2) abstrahiert, indem Entitäten zusammengefasst werden. Anschließend muss eine Schnittmenge zwischen dem so generalisierten Schema und dem Modell A. Stefans (Abbildung 3.1) gefunden werden. Betrachtet man die Klassen in Abbildung 6.2, lassen sich die blau unterlegten unter dem Begriff einer Entität zusammenfassen. Damit ist der Kern der Schnittmenge mit dem schon vorhandenen Graphen gefunden. Diese enthält außerdem noch, orange unterlegt, die Knotenklasse `Unstructured`, die, wie der Name sagt, ungeordnete Daten darstellt, die als ergänzender Kontext zu Entitäten und Patienten betrachtet werden. Die gesammelten Entitäten werden später in Abschnitt 7.1 über ein Mapping an mehrere Ontologien, die, wie in Abschnitt 2.3 erläutert, ebenfalls aus Entitäten und Relationen bestehen, angebunden. Das so entstandene Datenschema wird in Abbildung 6.3 dargestellt.

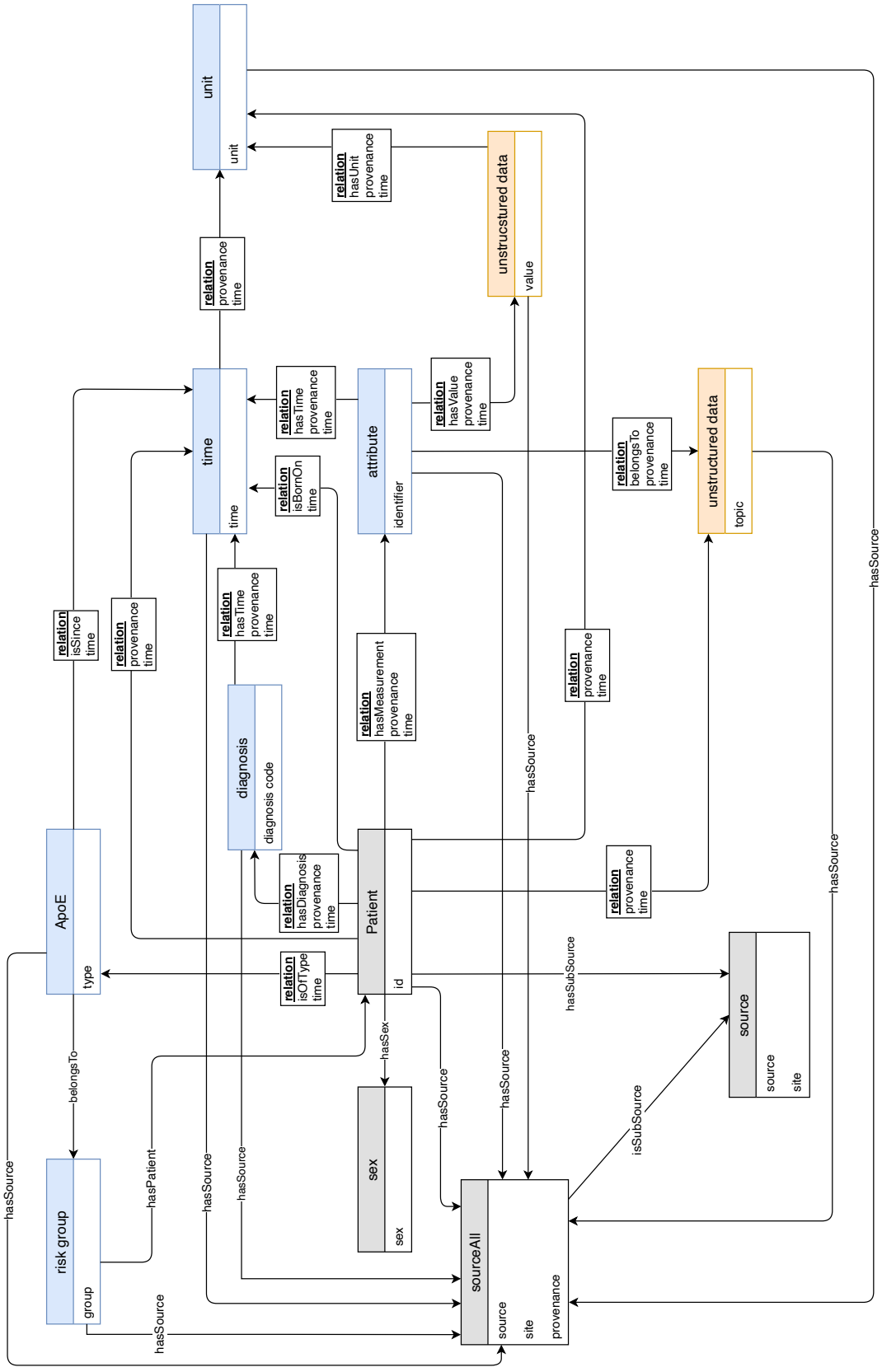

Abbildung 6.2: Detailliertes Datenschema nach Datenmodell

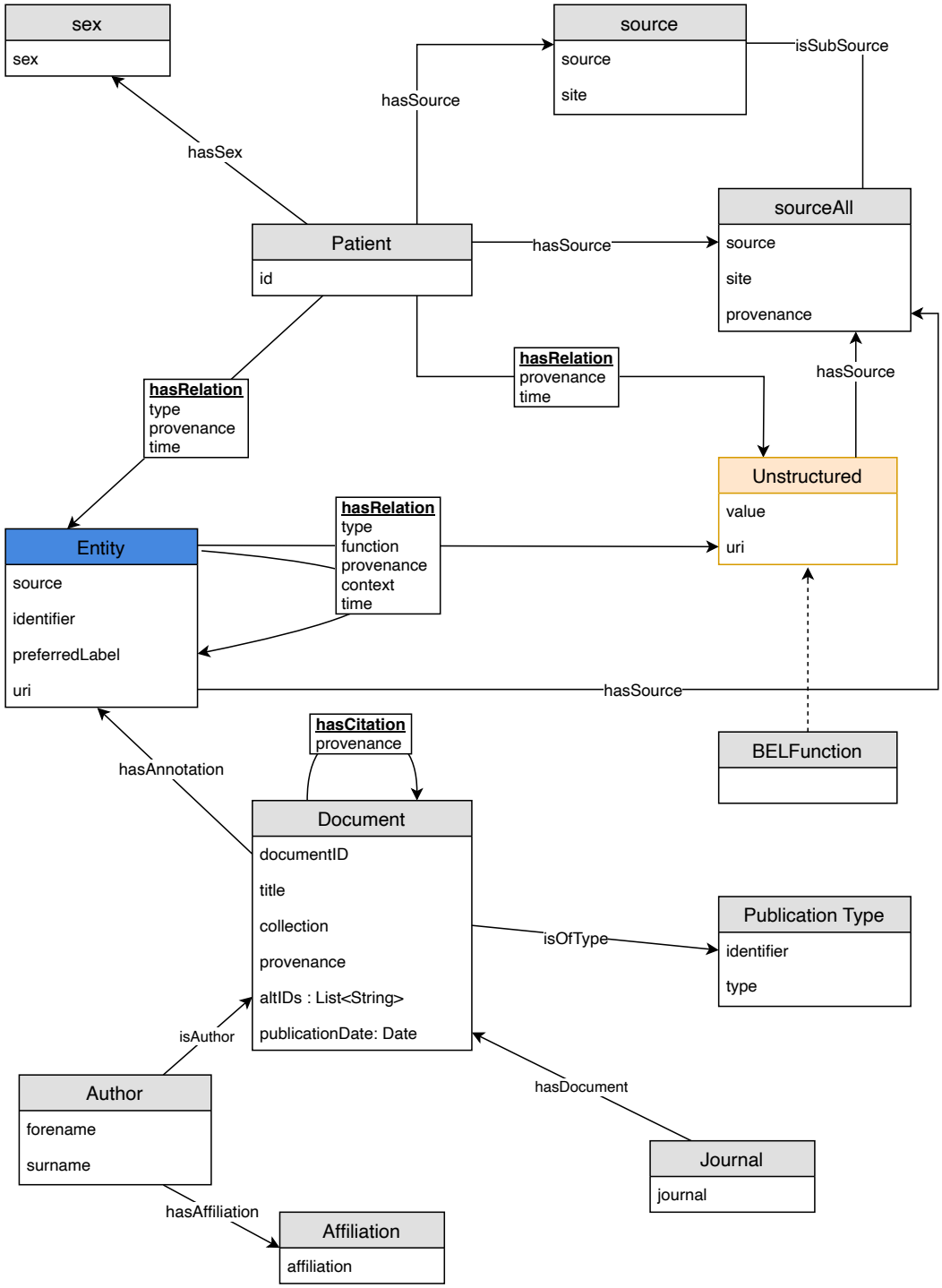

Abbildung 6.3: Formales Datenschema des Knowledge Graphen der klinischen Daten

7 | Programme

Um Daten in die NoSQL-Datenbank zu laden, gibt es verschiedene Möglichkeiten. Für eine geringe Menge an Daten lassen sich Knoten und Kanten inklusive der Attribute manuell in *Neo4j* erstellen. Dies wird jedoch sehr schnell ineffizient oder sogar unmöglich. Eine Alternative ist die *Neo4j* Anweisung *LOAD CSV*. Diese eignet sich nach der Dokumentation von *Neo4j* vor allem für den Import mittelgroßer Datensätze bis zu einer Größe von etwa zehn Millionen Einträgen.
Der Rahmen der in dieser Arbeit behandelten Testdaten übersteigt diese Zahl nicht, doch zur Erzielung eines schnelleren Imports und leichterer Skalierbarkeit auf größere Datensätze wird eine dritte Methode verwendet: *neo4j-admin import*. Diese Variante erlaubt einen pro Datenbank nur einmaligen Massenimport sehr großer Datensätze. Dabei werden sowohl für Relationen als auch für Knoten des Graphen jeweils zwei CSV-Dateien angelegt. Die Erste stellt einen Header dar, der lediglich eine Zeile enthält, die die Spaltenüberschriften speichert. Die zweite CSV-Datei beinhaltet dann die eigentlichen Knoten oder Kanten mit ihren Attributen.[25] Der explizite Import wird in Abschnitt 7.3 behandelt.
Die Entitäten und Relationen der verwendeten Ontologien lagen bereits als passende CSV-Dateien vor. Das Programm, das in Abschnitt 7.1 vorgestellt wird, erstellt also lediglich die außerhalb der Ontologien benötigten CSV-Dateien und Header, während das in Abschnitt 7.2 präsentierte Skript die CSV-Dateien für die inhaltliche Umgebung der ApoE-Gene bereitstellt.

7.1 Mapping

Die oben erwähnten artifiziellen Daten sind in einer CSV-Datei gegeben, die dem Muster entspricht, das in Listing 7.1 dargestellt ist.

```
    "Visit","sex","rg","diagnosis","e1","e2","HG"
    "0_BN2_0",0,"3","DOID:9841",0,3,"['HGNC_39962', 'HGNC_42531', 'HGNC_46189', '↩
        HGNC_19562']"
    "0_BN2_1",0,"4","DOID:5831",2,2,"[]"
    "1_COL_0",0,"2","DOID:3229",1,0,"['HGNC_17062', 'HGNC_8949']"
    "1_COL_1",0,"4","DOID:7207",1,0,"['HGNC_39664', 'HGNC_30311', 'HGNC_7177', '↩
```

```
        HGNC_15305', 'HGNC_47431', 'HGNC_13743', 'HGNC_35254']"
"1_COL_2",0,"2","DOID:2073",1,2,"['HGNC_39378', 'HGNC_17884', 'HGNC_11099', '↩
        HGNC_24460', 'HGNC_3048', 'HGNC_35038', 'HGNC_18669']"
"1_COL_3",0,"2","DOID:2814",2,3,"['HGNC_21343', 'HGNC_28609', 'HGNC_28003', '↩
        HGNC_18359', 'HGNC_49738', 'HGNC_2864']"
"2_BN2_0",0,"4","DOID:9861",1,1,"['HGNC_17147']"
```

Listing 7.1: Muster der CSV-Datei

mit einer Pfad-Spezifizierung. Im Zielverzeichnis werden die anzulegenden CSV-Dateien gespeichert. Python bietet verschiedene Möglichkeiten zur Sammlung von Daten an:

- List: Listen sind Arrays, die eine intrinsische Ordnung aufweisen, Dopplungen erlauben und veränderbar sind.
- Tuple: Ein Tuple ist einer Liste sehr ähnlich, erlaubt jedoch keine Änderungen.
- Dictionary: Ein Dictionary ist ungeordnet, erlaubt keine Duplikate und verwendet Key-Value-Paare zum Speichern von Daten.
- Set: Sets sind Dictionarys sehr ähnlich, sie speichern jedoch nur Keys und keine zugehörigen Values. [26]

Dieses Programm verwendet Listen und Sets sowie ein Dictionary. Zunächst werden leere Sets und Listen erstellt, die unterschiedliche Zwecke erfüllen. Die Gruppe von Sets dient der Verhinderung von Duplikaten: Patienten können in der einzulesenden Datei mehrfach vorkommen, genauso wie Lokalitäten oder Besuchsanzeiger. Das Gleiche gilt zwar auch für Entitäten wie HGNC und Diagnosen. Diese sind jedoch durch die HGNC Ontologie und die *Disease Ontology* bereits vorhanden und müssen daher nicht als eigene Klassen und auszugebende CSV-Dateien erstellt werden. Um Duplikate zu vermeiden, werden die bereits betrachteten Daten in diesen Sets gespeichert. Vor der Erstellung eines neuen Objekts wird dann durch Abfrage des entsprechenden Sets gewährleistet, dass beispielsweise der Patient noch nicht erstellt wurde.
Die leeren Listen dienen der Speicherung der erstellten Kanten und Knoten, was notwendig ist, um die Objekte am Ende in die CSV-Dateien schreiben zu können. Darüber hinaus wird ein Dictionary für die Abbildung der ε-Tupel auf die Risikogruppen erstellt (vgl. Abschnitt 6.1). In der einzulesenden Datei sind nicht nur die ε-Werte, sondern auch Risikogruppen gegeben. Da Letztere sich aus biomedizinischer Sicht jedoch implizit aus den Epsilons ergeben, wird dieses Dictionary verwendet, um die zugehörige Risikogruppe zu ermitteln und die gegebenen Werte dieser Spalte werden ignoriert.
Das eigentliche Programm erstellt als erstes mit der Methode `createPath()` das Verzeichnis, das durch die obigen Pfade angegeben wird, um dort später die Ausgabe zu speichern. Danach werden die Geschlechter-Knoten mit der Methode `createSex()` erstellt. Dies kann vor dem eigentlichen Einlesen passieren, da die Anzahl der Geschlechter

endlich ist. Bei der später beschriebenen Laufzeitanalyse wird erläutert, warum es auch effizienter ist, konstante Mengen derart auszulagern.
Schließlich wird die Methode `impCSVfile(_filename)` aufgerufen. Sie stellt den Kern des Programms dar. `_filename` wird hier als Platzhalter für einen Dateinamen einer einzulesenden CSV-Datei verwendet, da später mehrere Dateigrößen untersucht werden. Listing 7.2 zeigt, dass die Methode zunächst die gegebene Datei einliest und dann einige später verwendete Hilfsvariablen erstellt. *provenance* wird auf -1 gesetzt. Es soll wie oben beschrieben die Zugehörigkeit der einzelnen Daten zueinander bewahren und wird daher für jede gelesene Zeile der CSV-Datei um eins inkrementiert. Selbstverständlich lassen sich unter *provenance* auch konkretere und komplexere Informationen speichern.

```
def impCSVFile(_filename):

  with open(_filename) as csvfile:
      readCSV = csv.reader(csvfile, delimiter=',')


      # provenance dummy for test data
      provenance = -1
      # placeholder uri
      uriPlaceholder = 0
      # check first line
      firstLine = True

      # create source:
      source = _filename

      # iterate through all rows
      for row in readCSV:
```

Listing 7.2: Begin CSV-Import

Ähnliches gilt für den *uri-placeholder*: Die Entitäten innerhalb des Datenschemas nutzen eine *uri*, die an dieser Stelle verwendet werden könnte, aber für die gegebenen artifiziell generierten Daten nicht explizit gegeben ist. Der Name der eingelesenen Datei wird als *source* gespeichert, um als spätere ID der `source`-Knoten und Attribut der `sourceAll`-Knoten dienen zu können. Wären beispielsweise verschiedene Quelldateien gegeben, könnte man über *source* die Zugehörigkeit speichern und später abfragen.
Danach wird jede Reihe der CSV-Datei einzeln aufgerufen. Innerhalb eines solchen Aufrufs wird zunächst abgefragt, ob die erste Zeile aktiv ist, da diese nur die Überschrift der Spalten der CSV-Datei enthält und nicht mit verwendet werden soll. Dann werden die einzelnen Werte des Besuchs aus *row [0]* voneinander getrennt und in einem Array *visit* gespeichert. Außerdem wird der letzte Eintrag der Zeile, der aus einem String besteht, welcher wiederum eine Liste von Strings von HGNC-Werten enthält, aufgeteilt und die Werte werden einzeln gespeichert. Dabei werden die eckigen Klammern entfernt (vgl. Listing 7.3). Für den Fall, dass bei Realdaten oder alternativen artifiziellen Daten anstatt

genau einer Diagnose mehrere oder gar keine Diagnosen zu einem Besuch gegeben sind, lässt sich dieser Teil auf die Diagnosen übertragen.

```
# split last entry into parts (split HGNC markers)
l = len(row)
# "['HGNC_XXXX', 'HGNC_YYYY', 'HGNC_ZZZZ']"
# listHGNCValues = [HGNC_XXXX, HGNC_YYYY,...]
listHGNCValues = row[l-1].strip(']['').split(', ')
```

Listing 7.3: HGNC Splitting

Anschließend soll der Patientenknoten erstellt werden, allerdings nur, wenn er nicht bereits vorhanden ist. Die bisher gelesenen Patienten-IDs werden in Sets gespeichert. In jeder eingelesenen Zeile wird mittels einer *if*-Abfrage geprüft, ob der Patient mit der aktuellen ID schon als Objekt erstellt wurde oder nicht. Die Auswirkung dieser Abfrage auf die Laufzeit wird im folgenden Kapitel untersucht.
Im Anschluss werden `source`-Knoten erstellt. Dabei wird wie oben beschrieben zwischen `sourceAll` und `source` unterschieden. In Ersterem wird zusätzlich die *provenance* gespeichert. Des Weiteren wird die Information, um den wievielten Besuch des Patienten es sich handelt, als `Unstructured` gespeichert. Da in den artifiziellen Daten keine expliziten Zeitpunkte gegeben sind, wird dieser Knoten als implizite Zeitangabe verwendet.
Am Anfang einer jeden Reihe, die von der *for*-Schleife durchlaufen wird, setzt das Programm das Geschlecht auf weiblich. Falls dann beim Lesen der Zeile ein anderes Geschlecht, in diesem Fall männlich, gefunden wird, setzt das Programm die Variable dann auf den entsprechenden Wert. Dies wird später für die Kanten des Graphen, die den Patienten ihr Geschlecht zuordnen, benötigt. Falls das Geschlecht nicht nur eine binäre Angabe beinhaltet, lässt sich diese Variable über ein endliches Dictionary mit allen Geschlechtern setzen. Danach betrachtet man die $\varepsilon-$Werte des Patienten. Die Informatik zählt anders als die Biologie, so dass in der gegebenen Datei die Werte dazu als Ziffern $z \in \{0,1,2,3\}$ angegeben werden. Benötigt sind jedoch die Werte $w \in \{1,2,3,4\}$. Aus diesem Grund werden die ε-Werte noch um eins inkrementiert. Aus biologischer Sicht gilt: $(\varepsilon i, \varepsilon j) = (\varepsilon j, \varepsilon i)\ \forall i, j \in \{1,2,3,4\}$ [4]. Infolgedessen wird geprüft, ob die beiden Epsilons in der richtigen Reihenfolge vorliegen. Andernfalls werden sie vertauscht gespeichert. Aus der Kombination lässt sich dann über ein Dictionary die zugehörige Risikogruppe ermitteln (vgl. Listing 7.4).

```
# dictionary for riskgroups:
epsRgDict = {
    'epsilon1epsilon1': 'low',
    'epsilon1epsilon2': 'low',
    'epsilon1epsilon3': 'low',
    'epsilon2epsilon2': 'low',
```

```
    'epsilon2epsilon3': 'low',
    'epsilon3epsilon3': 'low',
    'epsilon1epsilon4': 'medium',
    'epsilon2epsilon4': 'medium',
    'epsilon3epsilon4': 'medium',
    'epsilon4epsilon4': 'high'
    }

[...]

# store espilons (given as 0-3, needed as 1-4)
e1 = int(row[4]) + 1
e2 = int(row[5]) + 1

# create tempEpsId for edge to eps1/eps2 combination
if e1 > e2:
    tempEpsCombId = 'epsilon' + str(e2) + 'epsilon' + str(e1)
else:
    tempEpsCombId = 'epsilon' + str(e1) + 'epsilon' + str(e2)

# find corresponding riskgroup
tempRg = epsRgDict[tempEpsCombId]
```

Listing 7.4: Zuweisung der Risikogruppen

Schließlich muss noch das Mapping, der eigentliche Kern des Programms, entwickelt werden. Dazu werden alle benötigten Kanten mit ihren Attributen als Objekte erstellt und in separaten Listen gespeichert. Die erstellten Kanten erhalten einen Relations-Typ, die zugehörige *provenance*, einen Start- und einen Zielknoten sowie einen *time*-Parameter. Danach werden diese Objekte in eigenen Listen gespeichert, die später für die Ausgabe als CSV-Datei durchlaufen werden. Für die Kanten zwischen Patienten und HGNC-Werten wird dabei die aktuelle Liste *listHGNCValues* der zu dieser Zeile gehörenden HGNC-Werte Eintrag für Eintrag bearbeitet, allerdings nur, wenn diese mindestens einen Eintrag enthält. Diese Bedingung ist wichtig, weil sonst in der CSV-Datei leere Felder auftauchen und beim Import in die Datenbank Probleme verursachen. Dies würde zu Kanten führen, die einen Start- aber keinen Endknoten enthalten, wodurch der spätere Import in die Graphdatenbank scheitern würde. Wie oben schon erwähnt wurde, lässt sich dieses Prinzip auch übertragen, wenn neben HGNC-Werten noch weitere Attribute in mehrfacher Ausführung vorliegen. Nachdem alle möglichen Kanten als Objekte erstellt und in den separaten Listen gespeichert worden sind, ist die Methode beendet.
Die beiden Methoden `nodeCreation()` und `edgeCreation()` arbeiten sehr ähnlich. Beide rufen mehrere Methoden auf, die dann wiederum jede mit Graph-Objekten, seien es Knoten oder Kanten, befüllte Liste auslesen und die einzelnen Objekte in Zeilen einer objekteigenen CSV-Datei schreiben. Es erfolgt eine Orientierung an dem verwendeten Datenschema. Überflüssige Attribute werden beim Schreiben ignoriert. Schließlich werden noch die letzten beiden Methoden `headerNodes()` und `headerEdges()` aufgerufen. Diese erstellen zu jedem Knoten- und Kantentyp einen geeigneten Header für die

Datenbank. Dieser besteht nur aus einer Zeile, die für *Neo4j* als Überschrift für die tabellarisch angeordneten CSV-Dateien dient, um die Kanten und Knoten richtig einzulesen. Das Programm beschränkt sich durch die schmale Grundlage der artifiziellen Testdaten nur auf einen Teil der möglichen Daten. Trotzdem lässt es sich auf umfangreichere Daten erweitern. Die Knoten werden als eigene Objekte gespeichert, was eine Erweiterung um zusätzliche Attribute und sogar klassenspezifische Funktionen erlaubt. Außerdem behält es durch die Objektorientierung die Flexibilität, Datentypen schnell zu ändern oder entfernen zu können. Das Gleiche gilt für die erstellten Kanten. Es handelt sich auch hier ausschließlich um Objekte, die dadurch jederzeit erweitert, modifiziert oder verkürzt werden können. Für die zur Verfügung gestellten dünn besetzten artifiziellen Testdaten wäre dies nicht notwendig gewesen, aber das Verfahren gewährleistet die generische Anwendbarkeit des Skripts.

7.2 Kontextumgebung des ApoE

Einer der zentralsten Aspekte der klinischen Daten ist das Apolipoprotein-E-Gen (kurz: ApoE). Es trägt nach [27] den bisher größten bekannten genetische Risikofaktor für die Erkrankung an Alzheimer-Demenz. Aus diesem Grund wird das ApoE in einen kleinen biomedizinischen Kontext gesetzt.
Für diesen wird [4] verwendet. Die Erstellung des Subgraphen erfolgt auf der Basis des Abbildung 7.1 dargestellten Schemas. Er wird ebenso wie die verwendeten Ontologien an die klinischen Daten angefügt und stellt, wenn auch nicht in dem Umfang einer ganzen Ontologie, eine Kontextumgebung für das untersuchte Protein dar. Der genauere Zusammenhang zwischen den dabei gegebenen einzelnen Klassen wurde bereits in Abschnitt 6.1 erläutert.
Das Skript ruft hintereinander mehrere Methoden auf. Die Erste erstellt, wie schon beim Mapping, ein Verzeichnis, in dem die später ausgegebenen CSV-Dateien gespeichert werden sollen. `createEpsNodes()` erstellt dann für jedes Epsilon ein eigenes Objekt, das als Knoten des Subgraphen dient. `createRiskGroups()`, `createRsCombinations()` und `createEpsilonCombinations()` werden danach aufgerufen und erstellen Knoten für die Risikogruppen, die SNP-Kombinationen der rs-Werte und die ε-Tupel. Anschließend wird die Methode `createEdgesOnts()` aufgerufen. Diese erstellt alle benötigten Kanten zwischen den zuvor generierten Knoten. Aufgrund des geringen Umfangs der verwendeten Informationen können die Kanten einzeln erstellt werden. Einerseits ist davon auszugehen, dass sich der wissenschaftliche Hintergrund nicht ändern wird, und andererseits entsteht so ein leicht nachvollziehbares Programm. Die Knoten und Kanten werden, analog zum Mapping, in eigenen Listen gespeichert und schließlich über die Methode `writeToCSV()` in die entsprechenden CSV-Dateien geschrieben. Außerdem werden in der Methode auch die zugehörigen Header-CSV-Dateien erstellt.

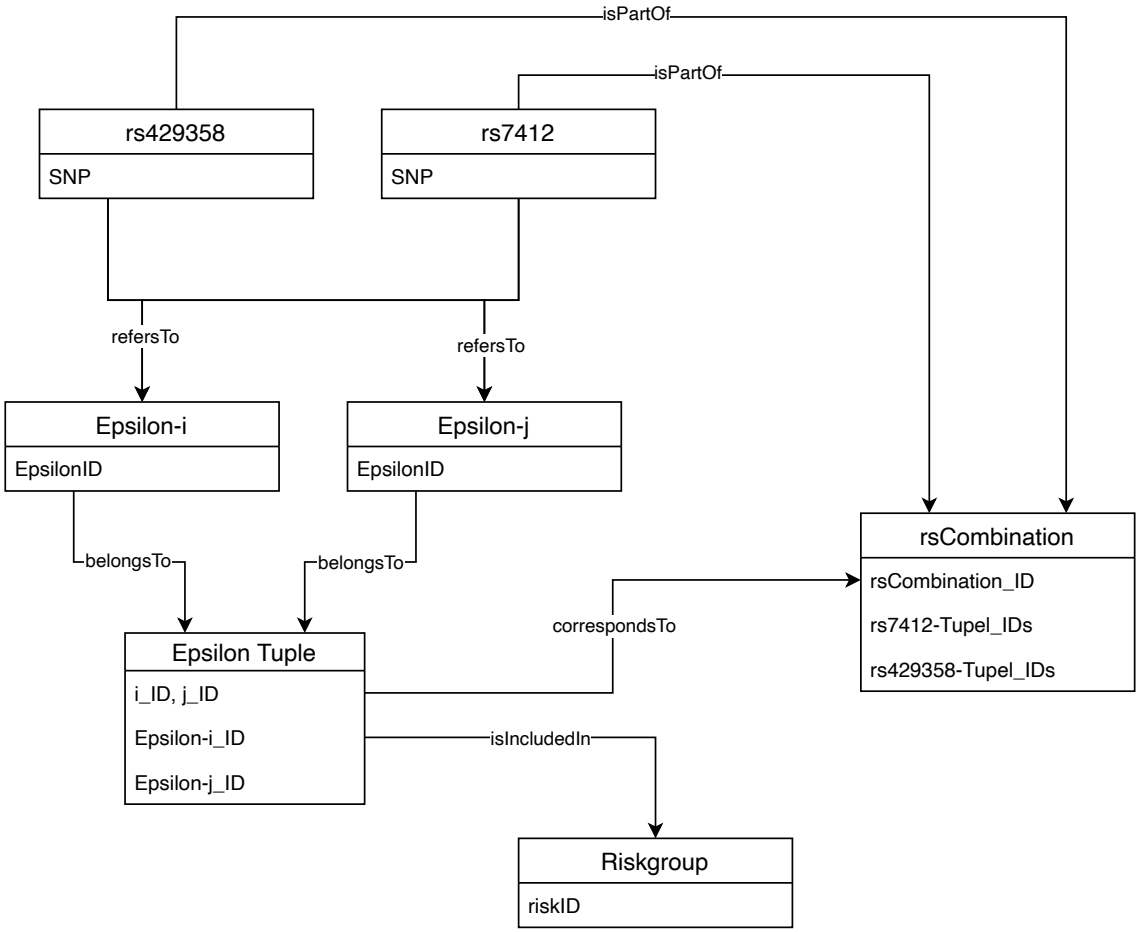

Abbildung 7.1: ApoE Kontext Schema

7.3 Import in *Neo4j*

Der Import der Daten in die Graphdatenbank *Neo4j* erfolgt mithilfe des *import tools* von *Neo4j*. Dieses eignet sich für einen effizienten Import besonders großer Anzahlen von Knoten und Relationen, kann aber ausschließlich für eine zuvor leere Datenbank erfolgen. Zunächst wird von der *Neo4j* Webseite [28] der *Neo4j* Community Server, *Neo4j* Community Edition 3.5.20 heruntergeladen. Für jede Knotenklasse und Relation werden dann zwei CSV-Dateien benötigt. Die eine enthält zeilenweise die einzelnen Knoten oder Kanten mit ihren Attributen, die andere stellt einen einzeiligen Header dar, der die Überschriften für die einzelnen Spalten der zugehörigen ersten Datei enthält. Diese Dateien werden dann im Verzeichnis *NEO4J_HOME/import* unter einer in [25] vorgegebenen Ordnerstruktur hinterlegt. *NEO4J_HOME* bezeichnet dabei den Ordner, in dem sich der *Neo4j* Server befindet. Für die spätere Verwendung von Algorithmen, wird die *Neo4j* Graph Data Science Library Version 1.1.3 als Plug-in installiert. (vgl. [29]) Anschließend können die Daten in die Datenbank geladen werden. Dies geschieht mit einem Command für die Konsole. Es wird in den *NEO4J_HOME*-Ordner navigiert und nach dem Muster von [25] der Befehl erstellt. Dieser ist unter Abschnitt A.4 zu finden.

Dabei können für den entstehenden Knowledge Graphen die in 2.1.3 definierten Funktionen λ und σ konkretisiert werden. Als Beispiel hierfür wird die Klasse `sourceAll` mit dem in Listing 7.5 dargestellten Header und einer Beispielinstanz in der Zeile darunter herangezogen. Die Knoten der Klasse `sourceAll` haben als Propertys die Attribute *sourceAll-ID*, *source*, *site* und *provenance* erhalten. Für die Anwendung der Funktion $\sigma : (V \cup E) \times P \to X$ gilt, dass diesen Propertys durch das Mapping in Abschnitt 7.1 Werte aus X zugewiesen wurden. Diese sind in der zweiten Zeile von Listing 7.5 zu sehen. Die Menge X besteht also für die `sourceAll`-Knoten aus den verschiedenen durch das Mapping generierten *provenanceIDs*, *sites*, *provenance*-Werten und dem als *source* gespeicherten exportGross.

```
sourceAll:ID(sourceAll-ID),source,site,provenance
provenanceID_0,exportGross.CSV,BN2,0
```

Listing 7.5: CSV-Header mit Beispielinstanz

Die Funktion $\lambda : (V \cup E) \to L$ aus 2.1.3 wird durch den Import selbst gegeben. Wieder wird als Beispiel der `sourceAll`-Knoten betrachtet. Der Import-Befehl in Abschnitt A.4 enthält für diesen Knoten den Teilbefehl, der in Listing 7.6 gegeben ist. Nach [25] weist die Bezeichnung vor dem Gleichheitszeichen den importierten Knoten ein Label zu. Analog können auch die anderen Knoten und Kanten mit 2.1.3 verknüpft werden.

```
--nodes:sourceAll="import/CSVHeaderFiles/nodes/sourceAll-header.csv,import/nodes/↩
    sourceAll.csv"
```

Listing 7.6: Import `sourceAll`

Der Import für die kleine CSV-Quelldatei mit etwa 1300 Zeilen hat bei mehreren Versuchen eine Zeit zwischen 4999 und 5679 Millisekunden benötigt, der große Import mit der 100-fachen Datenmenge zwischen 28195 und 32962 Millisekunden. Die genauen Werte sind im Anhang unter Abschnitt A.2 zu finden. Nach dem erfolgreichen Import kann die Untersuchung der aus den klinischen Fragetstellungen generierten Querys erfolgen (vgl. Abschnitt 8.3).

Der durch den Import entstandene Graph in *Neo4j* kann aufgrund der Millionen Knoten und Kanten nicht vollständig angezeigt werden. Als Beispiel eines dort gefundenen Knoten und seiner Nachbarn ist in Abbildung 7.2 ein Knoten der Klasse `sourceAll` abgebildet. Im oberen Bereich der Abbildung befinden sich Erklärungen zu den verschiedenen Farbcodierungen der Knoten.

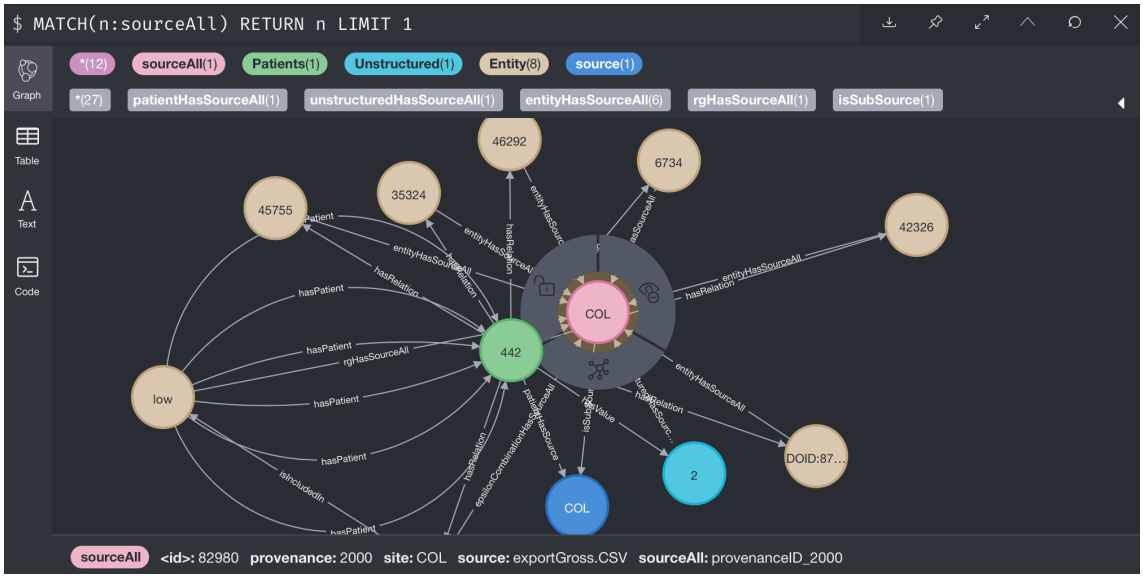

Abbildung 7.2: Beispielknoten `sourceAll` aus dem Knowledge Graphen

8 Komplexität und Laufzeitanalyse

Dieses Kapitel der Arbeit ist in mehrere Abschnitte unterteilt, um die einzelnen Skripte und Querys separat zu betrachten. Es werden dabei zwei verschiedene Untersuchungen angestellt. Als Erstes wird wie in Abschnitt 2.4 dargestellt die Zeitkomplexität der Python-Programme untersucht. Dann erfolgt unter Zuhilfenahme mehrfacher Durchläufe der Programme eine explizite Zeitmessung und die Präsentation der Ergebnisse. Ähnlich wird mit den Querys verfahren. Für die zugrundeliegenden Algorithmen wird die Zeitkomplexität gegeben und danach die Bearbeitungszeit der Query in *Neo4j* gemessen und mit der Komplexität verglichen. Für die folgenden Abschnitte ist 2.4.2 von besonderer Relevanz, wenn Teile des Codes einzeln betrachtet und schließlich aufsummiert werden.

8.1 Zeitkomplexität des ApoE-Kontextes

Die Methoden des Programms werden einzeln betrachtet. Die erste Methode stellt lediglich einen Pfad für die Speicherung der Dateien bereit. Dies geschieht offensichtlich in konstanter Zeit, liegt also in der Komplexitätsklasse $\mathcal{O}(1)$. Die nächsten vier Methoden erstellen alle eine endliche und konstante Anzahl an Objekten. Auch dies liegt in $\mathcal{O}(1)$. Die Methode `createEdgesOnt()` erzeugt die Kanten des kleinen Subgraphen. Auch diese sind vordefiniert und bestehen daher aus einer invariablen Anzahl von Objekten und damit liegt diese Methode ebenfalls in $\mathcal{O}(1)$. Zu guter Letzt wird die Methode `writeToCSV()` betrachtet. Diese arbeitet zwar mehrfach Listen ab, da dort die Objekte der zuvor ausgeführten Methoden gespeichert werden, doch sind diese Listen auch mit einer konstanten Anzahl von Objekten gefüllt, wodurch auch diese Methode in $\mathcal{O}(1)$ liegt. Mithilfe des Moduls *time* lassen sich die Realzeiten der Programmdurchläufe messen. Dabei wurden bei zehn verschiedenen Durchläufen stets ähnliche Werte im Bereich 0,01 bis 0,03 Sekunden gemessen. Insgesamt ist dieses Ergebnis weder positiv noch negativ zu bewerten, da aufgrund der Ausgangssituation einer endlichen, invariablen Menge an Daten eine konstante Komplexität und eine sehr geringe Laufzeit zu erwarten waren. Dies stellt jedoch nur einen kleinen Teilaspekt der eigentlichen Arbeit dar.

8.2 Zeitkomplexität des Mappings

Noch vor den eigentlichen Methoden werden mehrere Pfade sowie Sets, Listen und ein Dictionary angelegt. Diese sind alle leer und die Erstellung liegt daher in $\mathcal{O}(1)$. Da das Mapping mit verschiedenen Methoden arbeitet, werden diese für die Laufzeitanalyse nacheinander betrachtet. Die beiden Methoden `createPath()` und `createSex()` haben jeweils nur vier Befehle. Sie laufen in konstanter Zeit, da der Aufwand offensichtlich nicht von der Größe der einzulesenden Datei abhängt. Die Methode `impCSVFile()` liest die CSV-Datei über einen Reader ein und erstellt ein Objekt, über das iteriert werden kann. Die Erstellung der Hilfsvariablen liegt wie oben schon in konstanter Zeit. Dann wird die *for*-Schleife betrachtet. Diese liest jede Zeile genau einmal, was eine lineare Zeitkomplexität bedeutet, also $for - loop = \mathcal{O}(n)$ für die Eingabegröße n. Wie schon oben liegt auch die Deklaration der Hilfsvariablen danach in konstanter Zeit. Gleiches gilt für die verwendeten *if*-Abfragen bezüglich der Überprüfung des Boolean-Werts für die erste Zeile und der Abfrage des Geschlechts der aktuell untersuchten Person.
Schließlich wird die HGNC-Liste mit `split()` aufgeteilt. Für die gegebenen Testdaten liegt dies auch in konstanter Zeit, da keine Zeile mehr als acht verschiedene Elemente in der HGNC Liste enhält. Theoretisch könnte man die Komplexität zusätzlich zur Eingabelänge auch bezüglich der Anzahl aller möglichen HGNC-Werte betrachten. Damit wäre für das Aufteilen der Liste ein Worst-Case von $\mathcal{O}(k)$ möglich, wobei k der Anzahl aller HGNC-Werte entspricht. Da diese Zahl aber aus biologischer Sicht nach oben durch eine Konstante beschränkt werden kann und nicht im Bezug zur Eingabegröße der Zeilen der CSV-Datei steht, kann auch hier $\mathcal{O}(1)$ angenommen werden.
Danach wird über ein Set geprüft, ob ein Element bereits gelesen wurde. Ursprünglich wurde dies nicht mit Sets, sondern mit Listen gemacht. Die Zeitkomplexität der Suche in einer Liste in Python ist jedoch von der Länge der Liste abhängig, welche wiederum durch die Größe der einzulesenden Datei bedingt ist. Damit würde man innerhalb der *for*-Schleife erneut $\mathcal{O}(n)$ benötigen. Dies würde insgesamt schon zu $\mathcal{O}(n^2)$ führen. An dieser Stelle zeigt sich die Stärke der Sets: Sie arbeiten ähnlich wie ein Dictionary, nur dass sie statt Key-Value-Paaren nur keys speichern. Dies erlaubt anders als bei Listen keine Duplikate. Von Vorteil ist außerdem die Verwendung einer Hash-Funktion, die einen solchen Key auf einen Index abbildet. Dadurch kann der Schlüssel wie ein Index verwendet werden und die Suche verkürzt sich bei geeigneter Hash-Funktion von $\mathcal{O}(n)$ auf $\mathcal{O}(1)$.
Der explizite zeitliche Unterschied wird in Abbildung 8.1 deutlich. Es fällt auf, dass die Zeiten, die bei der Variante mit Listen gemessen wurden, ziemlich genau das Quadrat der Zeiten aus der Variante mit Sets ergeben. Die explizite Zeitmessung korreliert also mit der theoretischen Analyse. Alle Messwerte wurden auf einem MacBook Air 2018 mit einem Intel Core i5-8210Y Dual Core mit 1,6 GHz (3,6 TurboBoost) durchgeführt.
Abgesehen von der Suche nach Duplikaten in Sets besteht die Methode zu einem großen

Teil aus Zuweisungs-, Erstellungs- und Speicherungsoperationen sowie Anhängen an Sets und Listen. Diese Operationen laufen alle in konstanter Zeit. [26] Noch zu erwähnen ist die Zuweiseung des Patienten zu einer Risikogruppe anhand seiner Epsilon-Kombination. Dies wird über ein Dictionary abgefragt. Zum einen gilt für die *lookup*-Operation in Dictionarys das Gleiche wie für die in Sets, zum anderen wird hier ohnehin nur eine endliche und feste Anzahl an Möglichkeiten betrachtet, daher liegt auch dies in $\mathcal{O}(1)$. [26] Insgesamt liegt die betrachtete Methode `impCSVFile()` also in $\mathcal{O}(n)$.
Die nächste Methode `nodeCreation()` erstellt aus den in Listen gespeicherten Objekten Einträge in CSV-Dateien. Dafür werden die Listen durchlaufen und jedes Objekt wird in eine eigene Zeile geschrieben. Da die gesamte Liste verwendet werden muss, kann die schnellere *lookup*-Funktion für Sets hier nicht ausgenutzt werden und die Zeitkomplexität liegt in $\mathcal{O}(n)$, allerdings nur additiv zur bisherigen Laufzeit, wodurch diese bei $\mathcal{O}(n)$ bleibt (vgl. 2.4.2). Gleiches wie für die gerade beschriebene Methode gilt auch für `edgeCreation()`.
Es bleiben noch die beiden Methoden `headerNodes()` und `headerEdges()`. Diese erstellen beide nur eine feste Anzahl an CSV-Dateien, die für die Datenbank als Header verwendet werden und folglich nur eine Zeile enthalten. Die Anzahl der dort verwendeten Operationen ist damit konstant und somit liegen beide Methoden in $\mathcal{O}(1)$. Nach 2.4.2 wird die Gesamtkomplexität des Programms damit zu $\mathcal{O}(1)+\mathcal{O}(1)+\mathcal{O}(1)+\mathcal{O}(n)+\mathcal{O}(n)+\mathcal{O}(n)+\mathcal{O}(1)+\mathcal{O}(1)=\mathcal{O}(n)$ und das Programm läuft in linearer Zeit, wobei n der Anzahl der Zeilen der CSV-Datei entspricht. Daraus folgt, dass das Programm in der Komplexitätsklasse $\mathcal{P}$ liegt (vgl. Abschnitt 2.4). Die expliziten Zeitwerte wurden sowohl für die Duplikatsvermeidung mithilfe von Listen als auch für die Duplikatsvermeidung mithilfe von Sets in zehn Durchläufen gemessen und in Abbildung 8.1 abgebildet.
In Anbetracht der Art der gegebenen Daten innerhalb der CSV-Datei ist die Laufzeit von $\mathcal{O}(n)$ als gut zu bewerten, da jede Zeile betrachtet werden muss und dies, wie oben beschrieben wurde, in konstanter Zeit $\mathcal{O}(1)$ erfolgt. Aus diesem Grund ist es kaum möglich, die Zeitkomplexität des Programms weiter zu verbessern.

8.3 Laufzeiten der Querys

Die in Abschnitt 5.2 erstellten Querys sollen jetzt auf den Graphen angewendet und bezüglich ihrer Laufzeit beurteilt werden. Dafür wird die tatsächliche Laufzeit in *Neo4j* bei mehrfacher Durchführung gemessen und in Bezug zur Zeitkomplexität der Algorithmen gesetzt. Die Querys werden nach ihren Kategorien geordnet betrachtet. Dabei erfolgt die Nummerierung in Anlehnung an Abschnitt 5.2.
Die meisten Querys sind RPQs. Nach [30] haben diese durch Überführung in einen nichtdeterministischen endlichen Automaten eine polynomielle Laufzeit. Für den praktischen Test wurden alle RPQs Q1, Q2, Q3, Q5, Q7, Q11, Q12 zehnmal aufgerufen und die benö-

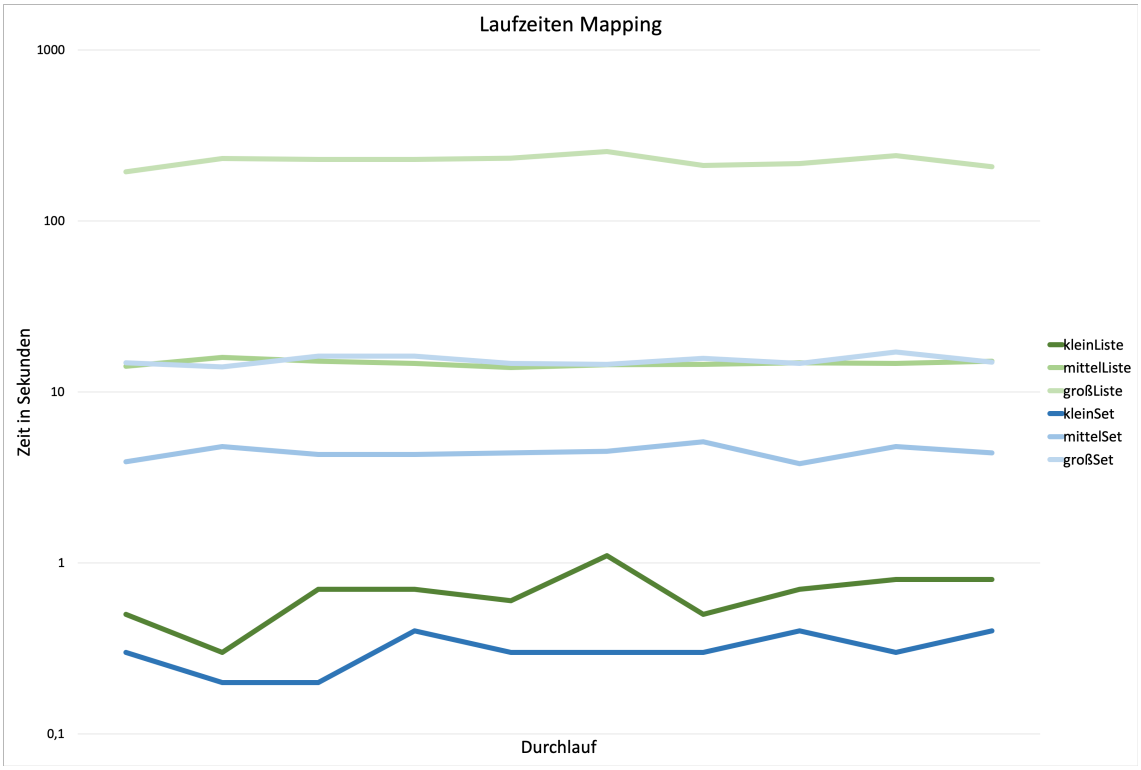

Abbildung 8.1: Laufzeit des Mappings mit Listen und Sets

tigte Zeit festgehalten. Die jeweiligen durchschnittlichen Laufzeiten der einzelnen Querys sind in Abbildung 8.2 zu sehen.

Für Darstellungszwecke wurde eine logarithmische Skalierung der Zeit-Achse ausgewählt. Die exakten Zeiten aller Durchläufe sind in Unterabschnitt A.3.1 zu finden. Bei der Betrachtung der Zeit fällt auf, dass die Abfragen Q1, Q3 und Q5 deutlich mehr Zeit benötigen als die anderen (Q2, Q7, Q11, Q12). Durch einen Vergleich der Abfragen nach Tabelle 5.2 lassen sich dafür nicht direkt Gründe ausmachen. Q1 unterscheidet sich von den anderen Querys darin, dass es global statt nur lokal sucht. Dadurch müssen wesentlich mehr Knoten untersucht werden, was zu einer solchen Laufzeitdiskrepanz führen kann. Außerdem ist Q1 zusammen mit Q7 die einzige RPQ mit mehrdeutigem Einstieg. Dies könnte eine schwankende Laufzeit je nach Einstiegspunkt nach sich ziehen. Da in Abbildung 8.2 jedoch gemittelte Werte mehrfacher Abfragen untersucht werden, sollte dies hier keine besondere Relevanz haben.

Nach [18] werden Querys mit steigender Anzahl abgefragter Attribute und verwendeter Relationen langsamer. Hier werden zunächst die abgefragten Attribute betrachtet: Diese unterscheiden sich bei den zwei Gruppen der schnellen und langsamen RPQs jedoch nicht grundlegend. Sie variieren zwischen einem und dreien, jedoch werden auch bei den sehr gut laufenden Abfragen mehrere Attribute benötigt. Der Vergleich der Relationen führt zu mehr Auffälligkeiten: Die langsamen Querys Q3 und Q4 verwenden jeweils zwei verschiedene Relationen. Bei den gut laufenden Querys machen Q7 und Q12 das zwar auch, aber wenn man die verwendeten Knoten betrachtet, fällt ein Unterschied auf: Q7 und Q12

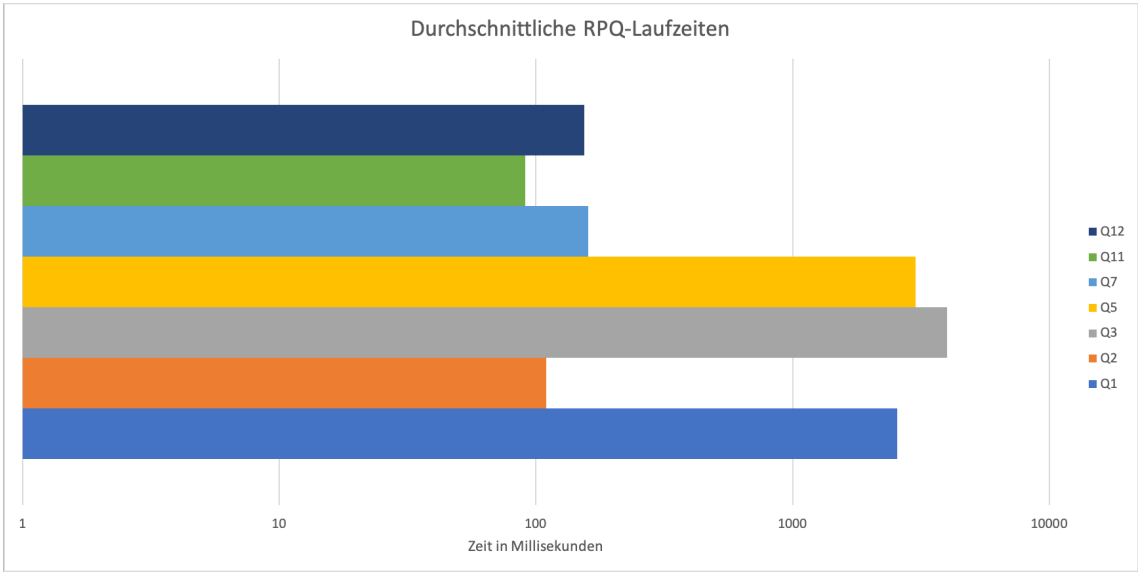

Abbildung 8.2: Durchschnittliche Laufzeiten der RPQs

verwenden neben der höheren Anzahl an Relationen *sex* als einen der Knoten in der Abfrage. Dieser kennt jedoch nur zwei Ausprägungen: männlich und weiblich. Die Querys Q3 und Q4 weisen eine ähnliche Struktur auf, greifen aber statt auf *sex* auf *Entity* zu. Die erstgenannte Klasse enthält nur zwei verschiedene Knoten (männlich und weiblich), während die zweite Klasse über 50.000 Knoten enthält. Es müssen also bei Entitäten wesentlich mehr Knoten überprüft werden, was den Geschwindigkeitsunterschied erklären kann.

Als nächstes werden die CRPQs (Q4, Q6, Q8) sowie die ECRPQ (Q10) zusammen betrachtet. Auch diese wurden zehnmal in der Datenbank angewendet, um eine durchschnittliche Zeit ermitteln zu können. Die exakten gemessenen Zeiten sind in Unterabschnitt A.3.2 zu finden, während Abbildung 8.3 einen Vergleich der durchschnittlichen Laufzeiten bietet. Die Skalierung ist auch hier logartihmisch gehalten. Dies verzerrt die Verhältnisse zwar, lässt aber unter regulärer Skalierung verschwindende Messungen erkennbar werden.

Nach [19] ist die Klasse der CRPQ $\mathcal{NP}$-vollständig, lässt sich also vermutlich nicht effizient lösen. Die oben erwähnte Abbildung zeigt diese Tatsache deutlich, denn neben sehr guten Ergebnissen für Q4 und Q10 findet man auch sehr schlechte Laufzeiten für Q6 und Q8. Die zuletzt genannte Query fällt sofort ins Auge, da sie nach Tabelle 5.2 gleich mehrere der oben genannten Aspekte belegt. Sie arbeitet als einzige der hier vorgestellten (E)CRPQs global, greift auf vier verschiedene Attribute zu und verwendet zwei verschiedene Relationen. Q6 sucht zwar nicht global, sondern nur lokal, hat aber auch zwei verschiedene Kantentypen, die jeweils in beiden konjugierten Teilquerys vorkommen und fragt sogar vier verschiedene Attribute ab.

Besonders interessant ist Q4. Die Abfrage sucht nur lokal, fragt aber fünf verschiedene Attribute ab und verwendet drei verschiedene Relationstypen. Eine mögliche Erklärung

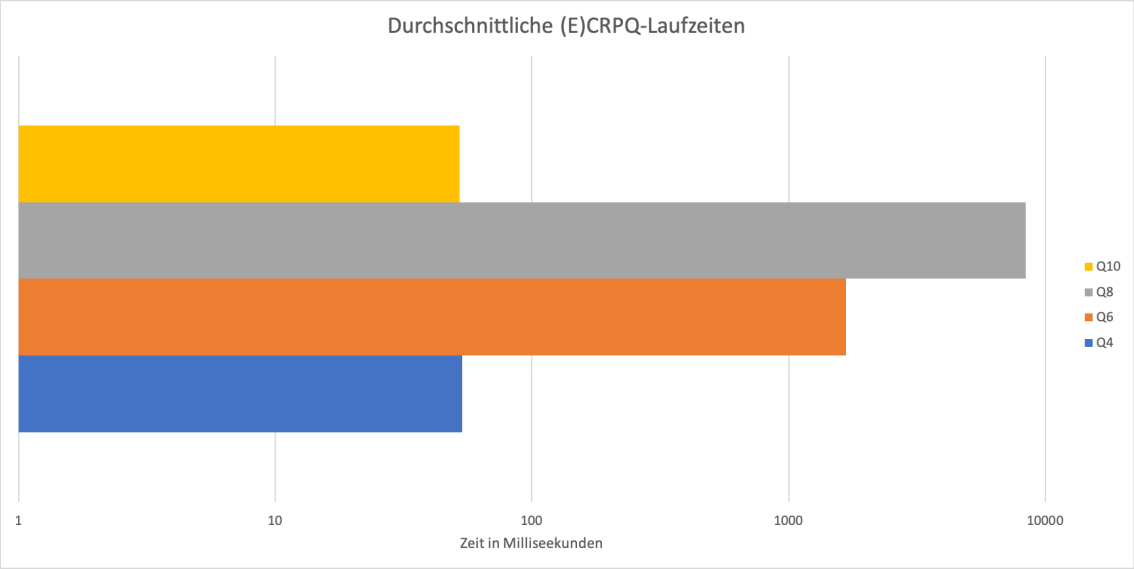

Abbildung 8.3: Durchschnittliche Laufzeiten der (E)CRPQs

für die trotzdem so geringe Laufzeit könnte die Struktur des Graphen liefern. Ein gerichteter Graph $G = (V,E)$ mit $n = |V|$ Knoten kann unter der Voraussetzung, dass es keine doppelten Kanten gibt, bis zu $n \cdot (n-1)$ viele Kanten besitzen, denn jeder Knoten v kann eine Kante (v,u) zu jedem anderen Knoten $u \in V, v \neq u$ haben. Betrachtet man die hier verwendeten CSV-Dateien, fällt die geringe Dichte des Graphen sofort auf. Ein Beispiel dafür bietet die eingelesene CSV-Datei, die zwar etwa 30.000 Patienten beinhaltet, aber pro Patient maximal sechs Besuche, die wiederum weniger als 20 Relationen enthalten. Also entstehen pro Patient nicht mehr als 120 Kanten. Bei Betrachtung des ganzen Graphen sieht man dieses Verhältnis auch beim Import in die Datenbank: Bei einer halben Million Knoten entstehen lediglich geringfügig mehr als 2,6 Millionen Kanten. So fällt bei Q4 auf, dass der kleine Teil der Daten, den die Query betrachtet, sehr dünn besetzt ist. Zu einem Patienten gibt es nur sehr wenige Daten, daher muss nur eine sehr begrenzte Anzahl an Möglichkeiten in Betracht gezogen werden. Dies kann die schnelle Antwort der Datenbank erklären.

Zuletzt wird die ECRPQ Q10 betrachtet. Sie verwendet drei verschiedene Attribute, aber nur einen Kantentyp und darüber hinaus wird hier das Gleiche wie bei der zuvor diskutierten Query Q4 deutlich: Für den Patienten liegt nur eine sehr begrenzte Menge an Daten vor, was die Geschwindigkeit der Abfrage erheblich beeinflussen dürfte.

Zum Schluss werden die verwendeten Algorithmen genauer untersucht. Alle drei sind in *Neo4j* integrierte und über das Plug-in Graph Data Science 1.1.3, das schon in Abschnitt 7.3 erwähnt wurde, bereitgestellte Algorithmen. Zunächst wird Q9 mit Degree Centrality betrachtet. Da die inzidenten Kanten eines jedes Knoten dabei gezählt werden, erhält man für einen dichten Graphen $G = (V,E)$ im ungünstigsten Fall eine Komplexität von $\mathcal{O}(|V|^2)$, wenn nämlich jeder Knoten $v \in V$ zu jedem anderen $u \in V, u \neq v$ inzident ist. In [18] wird bereits gezeigt, dass die in *Neo4j* implementierten Algorithmen auf großen

Netzwerken eine so hohe Laufzeit haben, dass sie nahezu unbrauchbar werden. Die in Unterabschnitt A.3.3 genau aufgelisteten Laufzeiten und der in Abbildung 8.4 dargestellte Durchschnitt der Abfrage Q9 zeigen für Degre Centrality jedoch zunächst etwas anderes. Hierbei kommt erneut zum Tragen, dass der Graph sehr dünn besetzt ist. Die Knoten haben nur wenige direkte Nachbarn und das wirkt sich hier stark auf den Algorithmus aus. Da es nur drei verschiedene Risikogruppen gibt, ließen sich im Falle einer hohen Laufzeit des Algorithmus aber auch die Knotengrade mit einzelnen Querys abfragen und dann vergleichen.

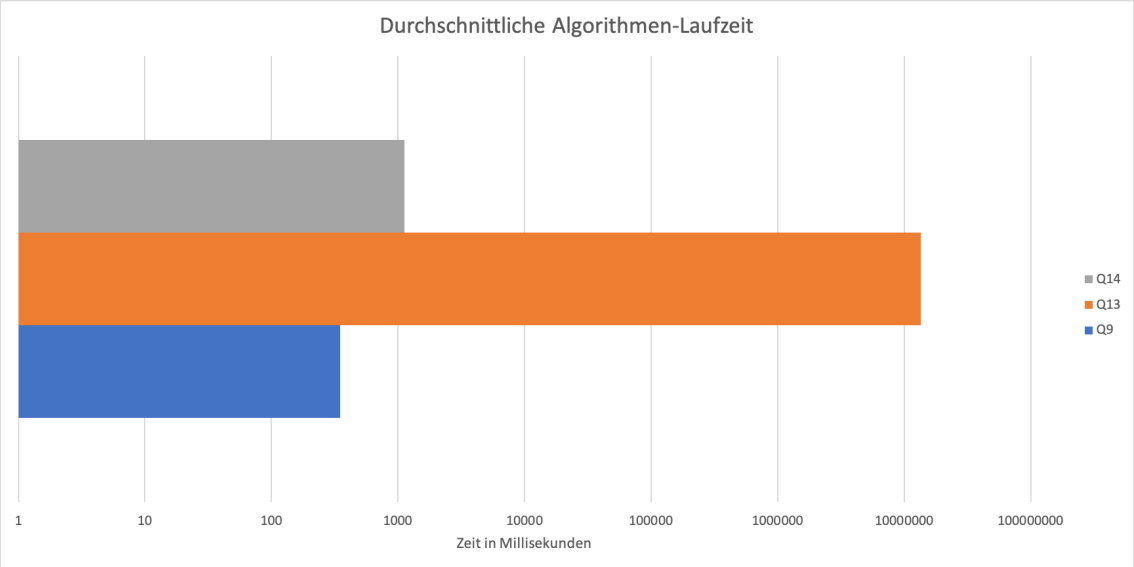

Abbildung 8.4: Durchschnittliche Laufzeiten der Algorithmen

Ähnlich sieht es bei Q14 mit der Berechnung eines kürzesten Pfades aus. Die durchschnittliche Laufzeit ist ebenfalls in Abbildung 8.4 dargestellt. Die exakten, gemessenen Werte sind in Unterabschnitt A.3.3 zu sehen. Als Beispiel für die Ausgabe einer Query ist das Resultat der Query Q14 in Abbildung 8.5 dargestellt.

Es gibt verschiedene Algorithmen, um kürzeste Wege in Graphen zu berechnen. Der in *Neo4j* verwendete Algorithmus soll eine Variante des Dijkstra-Algorithmus sein, der eine Zeitkomplexität von $\mathcal{O}(|V| \cdot log|V| + |E|)$ hat [18]. Für die gegebene Query läuft der Algorithmus auf dem vorhandenen Graphen sehr schnell. Wie aber in der oben genannten Quelle gezeigt wurde, wächst die Laufzeit mit für den Algorithmus weniger günstigen Graphen enorm, daher ist das hier erzielte Ergebnis mit Vorsicht zu betrachten. Die Problematik wird schließlich mit dem dritten Algorithmus deutlich. Nach [12] hat Betweenness Centrality eine Laufzeit von $\mathcal{O}(|V|^3)$, die jedoch mit dem Algorithmus von Brandes zu einer Zeitkomplexität von $\mathcal{O}(|V| \cdot |E|)$ verbessert werden kann. Dies ist nach Angaben in der Dokumentation von *Neo4j* auch der dort verwendete Algorithmus (vgl. [31]). Die hier ausgeführte Query lief, wie der Durchschnitt in Abbildung Abbildung 8.4 und die genauen Werte in Unterabschnitt A.3.3 zeigen, mehrere Stunden, bevor sie zu einem

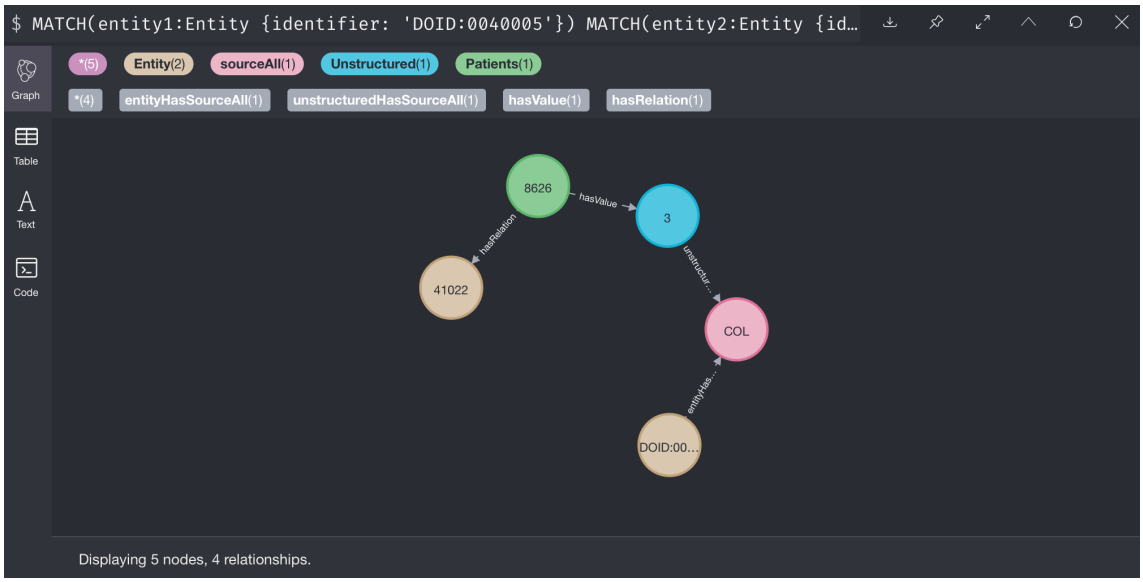

Abbildung 8.5: Resultat der Query Q14

Ergebnis kam. Dies steht nicht nur im starken Kontrast zu den anderen beiden Algorithmen, vor allem zu dem für kürzeste Wege, sondern macht sie auch nahezu unbrauchbar für praktische Anwendungen.

9 | Fazit

Es wurde anhand des zugrundeliegenden Datenmodells des biomedizinischen Bereichs ein Schema für den Graphen erstellt. Dieses gegebene Modell befindet sich zum Zeitpunkt der Arbeit noch nicht in seiner finalen Fassung. Daher ist noch mit weiteren Änderungen zu rechnen, an die das hier präsentierte Modell noch angepasst werden muss. Dies betrifft vor allem das Schema in Abbildung 6.2. Hinzukommende Variablen müssen dann ebenfalls in Klassen unterteilt und mit Relationen verknüpft werden. Das hier vorgestellte Prinzip kann jedoch darauf übertragen und das in Abbildung 6.3 gegebene Schema erweitert werden.
Als nächstes ist das in Python geschriebene Mapping zu betrachten. Je nach Art der Quelle der Daten wird eine andere Einlesemöglichkeit erforderlich und bei umfangreicheren Daten muss das eigentliche Mapping um die hinzugefügten Daten erweitert werden. Da die klassifizierten Variablen des Datenmodells jedoch als Objekte von Klassen in Python vorliegen, lässt sich das Programm schnell anpassen, indem neue Klassen definiert oder bereits vorhandene um neue Attribute erweitert werden. Zudem wurden für verschieden große Datenmengen zu einem bestimmten Patienten aus der Quelldatei passende Operationen verwendet, um das Mapping effizient zu erstellen. Als Beispiel kann hier angeführt werden, dass jeder Patient bei einem Besuch genau eine Diagnose erhalten hat, gleichzeitig aber eine Menge verschiedener oder auch gar keiner HGNC-Werte. Das bei den HGNC-Werten verwendete Muster zur Aufspaltung und Ordnung der verschiedenen Werte lässt sich innerhalb des Programms, also beispielsweise auf Patienten mit mehreren Diagnosen, übertragen. Zudem erlaubt die Struktur des Programms ebenfalls die zusätzliche Nutzung weiterer Ontologien. Diese können durch geringfügige Modifikation des Programms zusätzlich zur *Disease Ontology* und den HGNC-Werten eingepflegt werden. Da die Komplexität des Mappings linear gehalten werden konnte, stellen auch größere Datenmengen im Hinblick auf ihre Laufzeit kein Problem dar. Insgesamt wurde ein Programm erstellt, das also leicht zu erweitern ist und trotzdem eine, gemessen an der Datenmenge der Quelldatei, hohe Geschwindkeit bietet.
Die Untersuchung der gegeben Querys ist differenziert erfolgt. Mit einem dichteren Graphen fallen vor allem die Algorithmen in ihrer Laufzeit gegenüber den gewöhnlichen RPQs zurück. Da jedoch keine der zu Beginn gesammelten biomedizinischen Fragestellungen einen Algorithmus erfordert, werden die Laufzeiten der RPQs als wichtiger be-

trachtet. Die meisten Querys liefen trotzdem mit geringer Zeit, obwohl sie auf dem in Abschnitt 8.2 beschriebenen Prozessor getestet wurden. Auf einem leistungsfähigeren Rechner werden sie vermutlich deutlich bessere explizite Zeiten erreichen.

10 | Ausblick

Wie bereits erwähnt wurde, kann der im Rahmen dieser Arbeit erstellte Graph an den schon vorhandenen aus [3] angefügt werden, um Querys zuzulassen, die Knoten beider Graphen verwenden. Ein Beispiel dafür wurde in Abschnitt 1.1 bereits gegeben: Welche Literatur lässt sich zu einem gegebenen Patienten mit bestimmten Messwerten oder Diagnosen finden? Dabei gilt es zu klären, wie sich die Laufzeit einer solchen Anfrage auf dem zusammengesetzen Graphen verhält. Eine andere Möglichkeit zur Erweiterung des Graphen ist der Anschluss zusätzlicher Ontologien. Für die hier gegebenen Testdaten bietet sich dies nicht unbedingt an, für Realdaten kann aber beispielsweise die *Sequence Ontology* oder die *Gene Ontology* herangezogen werden. Was die Realdaten betrifft, so liegen diese zum Zeitpunkt der Fertigstellung der Arbeit noch nicht vor und auch das gegebene Datenmodell befindet sich noch im Entwicklungsprozess. Trotzdem lässt sich davon ausgehen, dass bei nicht grundlegenden Änderungen im Datenmodell dieses Mapping auf breitere Realdaten erweitert werden kann. Eine praktische Anwendung des Graphen in Kombination mit dem bereits gegebenen Graphen der *PubMed* Datenbank ist auch erst dann wirklich sinnvoll, wenn diese Daten vorliegen.

Im Hinblick auf Effizienz lassen sich Graph und Abfragen ebenfalls weiter optimieren. Es kann überlegt werden, wie das in [3] vorgestellte System für Polyglot Persistenz auf diesen Graphen angewendet werden kann. Der dortige Graph enthält jedoch deutlich mehr Knoten und Kanten. Es muss daher untersucht werden, ob der tatsächliche Laufzeitgewinn durch Polyglot Persistenz dessen Einsatz rechtfertigt. Darüber hinaus wurden in [18] die Querys zu großen Teilen ausgelagert. Anstatt Graph-Algorithmen der *Neo4j* Datenbank direkt aufzurufen, wird über elementare Querys mit dem Graphen kommuniziert und der Algorithmus dann extern über ein eigenes Skript ausgeführt. Dadurch lassen sich, wie anhand zweier Query-Kategorien in [18] gezeigt wurde, enorme Geschwindigkeitsgewinne beim Abfragen des Graphen erzielen.

Für das Mapping selbst können auch noch Verbesserungen der Geschwindigkeit erörtert werden. Wenn statt Python eine Sprache wie C oder C++ genutzt wird, verbessert sich zwar die Zeitkomplexität des Skripts nicht, aber vermutlich seine tatsächliche Laufzeit. Als Hybrid-Variante könnte unter Umständen Numba verwendet werden. Numba ist ein just-in-time Compiler für Python, der bei der richtigen Anwendung fast an Geschwindigkeiten von C oder FORTRAN herankommen soll. Numba erstellt für die Python-

Funktionen sogenannte Decorator, die ein eigenes neues Objekt darstellen. Das erste Ausführen einer Methode mithilfe eines solchen Decorators bringt noch keine Vorteile in der Geschwindigkeit. Dabei wird jedoch der verwendete Maschinencode von Numba gespeichert und bei erneutem Aufruf ausgeführt. Dadurch können ab der zweiten Verwendung einer Methode enorme Geschwindigkeitsgewinne erzielt werden. Ursprünglich ist Numba allerdings auf Beschleunigung numerischer Operationen, Berechnungen und Schleifen ausgelegt. Es steht in engem Verbund zu NumPy und ist insofern nicht für jeden beliebigen Python-Code sinnvoll. Auf das genutzte Mapping trifft davon nur die Schleife zu. Die Methode `impCSVFile()` beruht durch das Lesen der Quell-CSV-Datei auf einer einzigen großen Schleife. Es bleibt also zu untersuchen, ob sich die Verwendung von Numba positiv auf die Laufzeit des Mapping-Skripts auswirkt. (vgl. [32] [33])
Ein anderer Aspekt bezüglich der Laufzeit des Mappings ist die Tatsache, dass sie bereits gering ist. Für die Quelldatei mit über 130.000 Zeilen wurden Zeiten im Bereich von 15 Sekunden gemessen, für die kleinere Datei mit etwa 1300 Einträgen im Durchschnitt 0,3 Sekunden. Durch das lineare Wachstum lässt sich anhand der beiden gegebenen Datenpunkte der Durchlaufzeiten eine Gerade als obere Laufzeitschranke approximieren: $g(n) = 0,0001142191142n + 0.151515$. Die enorm geringe Steigung zeigt, dass selbst für Quelldaten mit einer Milliarde Einträgen eine Laufzeit von etwas über einem Tag nicht überschritten wird. Da eine solche Menge an klinischen Patientendaten aber unwahrscheinlich ist, ist ein tatsächlich signifikanter Laufzeitgewinn durch Numba zweifelhaft, zumal selbst dann mehrere Instanzen des Programms parallel Teildatensätze bearbeiten könnten. Voraussetzung dafür ist, dass mit einem geeigneten Sortieralgorithmus garantiert wurde, dass ein einzelner Patient nicht in mehreren Dateien auftaucht.
Zu guter Letzt kann die Verknüpfung von Graphentheorie und Biomedizin natürlich auch auf andere Bereiche ausgeweitet werden, was den Bogen zum Eingangszitat dieser Arbeit spannt.

A Anhang

A.1 *Cypher*-Querys

```
1: MATCH (e:Entity {source:'HGNC'})<-[:hasRelation]-(p:Patients) RETURN
p.patient AS Person, COUNT(DISTINCT e) AS number ORDER BY number DESC LIMIT
10

2: MATCH (r:Entity {identifier: "high"})-[:hasPatient]->(p:Patients) RETURN
COUNT(DISTINCT p) AS numberOfPatients

3: MATCH (e:Entity {source:'HGNC'})<-[:hasRelation]-(p:Patients)-[:hasValue]
->(v:Unstructured {value:'2'}) RETURN e.preferredLabel AS HGNC_Wert, COUNT(e)
AS number ORDER BY number DESC LIMIT 10

4: MATCH (u:Unstructured)<-[:hasValue]-(p:Patients {patient: '22504'})-
[:hasRelation]->(e:Entity {source: 'HGNC'}) MATCH (r:RiskGroups)-[:hasPatient]
->(p) RETURN e.preferredLabel AS HGNC, r.riskgroup AS RiskGroup, u.value
AS VisitNo ORDER BY u.value DESC

5: MATCH (e:Entity {source:'HGNC'})<-[:hasRelation]-(p:Patients)-[:hasValue]
->(v:Unstructured {value:'0'}) RETURN COUNT(DISTINCT p) AS p_count

6: MATCH (:Unstructured {value:"2"})<-[:hasValue]-(p:Patients)-[:hasRelation]
->(:Entity {preferredLabel: Älzheimer's disease"}) WITH p MATCH (:Unstructured
{value:"1"})<-[:hasValue]-(p)-[:hasRelation]->(:Entity {identifier:"35357"})
RETURN p LIMIT 10

7: MATCH (s:Sex)<-[:hasSex]-(:Patients)-[:hasValue]->(v:Unstructured) WHERE
v.value > '6' RETURN DISTINCT s.sex AS patientSex, COUNT(s) as sex_count

8: MATCH (e:Entity {source: 'HGNC'})<-[g:hasRelation]-(p:Patients)
```

```
<-[:hasPatient]-(r:Entity {category:'rg'}) WITH p,r, COUNT(g) AS numbOfHGNC
WHERE numbOfHGNC = 2 RETURN p.patient, r.identifier, numbOfHGNC

9: CALL gds.alpha.degree.stream({
nodeProjection: ['Entity','Patients'],
relationshipProjection: 'hasPatient'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).identifier AS name, score AS numberOfPatients
ORDER BY numberOfPatients DESC LIMIT 3

10: MATCH p=(e1:Entity {preferredLabel:Älzheimer's disease"})<-[:hasRelation]
-(A:Patients {patient: "12864"})-[:hasRelation]->(e2:Entity {source:"HGNC"})
RETURN p

11: MATCH (h:Entity {identifier: '37785'})<-[:hasRelation]-(p:Patients)
-[:hasRelation]->(e:Entity {identifier: 'DOID:14332'}) RETURN DISTINCT
p.patient as Patient, e.identifier as Diagnosis, h.identifier as HGNC_-
Value

12: MATCH (s:Sex)<-[:hasSex]-(p:Patients)-[:hasRelation]->(e:Entity
{identifier: 'DOID:0040021'}) RETURN DISTINCT p.patient, s.sex

13: MATCH(entity1:Entity {identifier: 'DOID:0040005'})
MATCH(entity2:Entity {identifier: '41022'})
CALL gds.alpha.shortestPath.stream({startNode: entity1, endNode: entity2,
nodeProjection: '*',
relationshipProjection: {all:{type: '*', orientation: 'UNDIRECTED'}}})
Yield nodeId, cost
RETURN gds.util.asNode(nodeId), cost

14: CALL gds.graph.create('myUndirectedGraph', [Patients",Ëntity"],
{hasRelation: {orientation: 'UNDIRECTED'}})
CALL gds.alpha.betweenness.stream('myUndirectedGraph')
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).preferredLabel AS entityLabel, centrality
AS number ORDER BY number DESC LIMIT 10
```

A.2 Import Laufzeitmessungen

#	klein	groß
1	5203	28837
2	5493	32962
3	4999	32743
4	5228	28195
5	5131	31809
6	5615	28366
7	5284	31462
8	5217	32037
9	5577	31865
10	5679	28379

Tabelle A.1: Exakte Messwerte Importzeiten in Millisekunden

A.3 Exakte Laufzeitmessungen der *Cypher*-Querys

A.3.1 RPQ

#	Q1	Q2	Q3	Q5	Q7	Q11	Q12
1	2085	353	8602	3303	154	229	83
2	3314	434	3698	2394	127	59	138
3	2476	40	2771	3120	118	63	132
4	2434	40	3308	2419	113	68	145
5	2089	36	3385	4725	324	61	211
6	2335	37	4596	2388	119	66	146
7	2714	39	2881	2945	117	98	166
8	3335	36	2904	3403	192	107	124
9	2314	41	4102	2519	125	87	134
10	2489	42	3660	2889	214	71	266

Tabelle A.2: Exakte Messwerte der RPQs in Millisekunden

A.3.2 (E)CRPQ

#	Q4	Q6	Q8	Q10
1	50	3951	8279	52
2	54	1439	10131	53
3	48	1238	7755	60
4	69	1157	7488	61
5	54	1802	6754	79
6	56	1107	7330	34
7	47	1181	8834	36
8	46	1245	7800	51
9	45	1589	7892	48
10	67	2008	11763	48

Tabelle A.3: Exakte Messwerte der (E)CRPQs in Millisekunden

A.3.3 Algorithmen

#	Q9	Q13	Q14
1	726	9980050	1236
2	269	16448918	1403
3	378	12740225	937
4	412	16048348	1030
5	266	17587662	1212
6	276	13244072	1262
7	298	11977288	507
8	330	18874953	1027
9	267	14407742	1438
10	281	15461768	1133

Tabelle A.4: Exakte Messwerte der Algorithmen in Millisekunden

A.4 Import-Befehl

```
bin/neo4j-admin import --nodes:Patients="import/CSVHeaderFiles/nodes/
patients-header.csv,import/nodes/patients.*" --nodes:Sex="import/
CSVHeaderFiles/nodes/sex-header.csv,import/nodes/sex.*"
--relationships:hasSex="import/CSVHeaderFiles/relations/hasSex-header.csv,
import/relations/hasSex.*" --nodes:Unstructured="import/
CSVHeaderFiles/nodes/unstructured-header.csv,import/nodes/unstructured.*"
--nodes:Entity="import/CSVHeaderFiles/nodes/entityNodes-header.csv,import/
nodes/entities.*" --nodes:Epsilon="import/CSVHeaderFiles/nodes/
epsilon-header.csv,import/nodes/epsilon.*" --nodes:Entity="import/
CSVHeaderFiles/nodes/epsilonCombination-header.csv,import/nodes/
epsilonCombination.*" --nodes:rsCombination="import/CSVHeaderFiles/
nodes/rsCombination-header.csv,import/nodes/rsCombination.*"
--nodes:Entity="import/CSVHeaderFiles/nodes/riskgroups-header.csv,import/
nodes/riskgroups.*" --relationships:hasRelation="import/CSVHeaderFiles/
relations/hasDiagnosis-header.csv,import/relations/hasDiagnosis.*"
--relationships:hasRelation="import/CSVHeaderFiles/relations/
hasGeneNomenclature-header.csv,import/relations/hasGeneNomenclature.*"
--relationships:hasPatient="import/CSVHeaderFiles/relations/
hasPatient-header.csv,import/relations/hasPatient.*" --relationships:
hasValue="import/CSVHeaderFiles/relations/hasValue-header.csv,import/
relations/hasValue.*" --relationships:belongsTo="import/CSVHeaderFiles/
relations/belongsTo-header.csv,import/relations/belongsTo.*"
--relationships:correspondsTo="import/CSVHeaderFiles/relations/
correspondsTo-header.csv,import/relations/correspondsTo.*" --relationships:
isIncludedIn="import/CSVHeaderFiles/relations/isIncludedIn-header.csv,
import/relations/isIncludedIn.*" --nodes:rs7412="import/CSVHeaderFiles/
nodes/rs7412-header.csv,import/nodes/rs7412.*" --nodes:rs429358="import/
CSVHeaderFiles/nodes/rs429358-header.csv,import/nodes/rs429358.*"
--relationships:rs7412isPartOf="import/CSVHeaderFiles/relations/
rs7412isPartOf-header.csv,import/relations/rs7412isPartOf.*"
--relationships:rs429358isPartOf="import/CSVHeaderFiles/relations/
rs429358isPartOf-header.csv,import/relations/rs429358isPartOf.*" --nodes:
source="import/CSVHeaderFiles/nodes/source-header.csv,import/nodes/
source.csv" --nodes:sourceAll="import/CSVHeaderFiles/nodes/
sourceAll-header.csv,import/nodes/sourceAll.csv" --relationships:
entityHasSourceAll="import/CSVHeaderFiles/relations/entityHasSourceAll-
header.csv,import/relations/entityHasSourceAll.*" --relationships:
```

```
rgHasSourceAll="import/CSVHeaderFiles/relations/rgHasSourceAll-header.csv,
import/relations/rgHasSourceAll.*" --relationships:
epsilonCombinationHasSourceAll="import/CSVHeaderFiles/relations/
epsilonCombinationHasSourceAll-header.csv,import/relations/
epsilonCombinationHasSourceAll.*" --relationships:patientHasSourceAll=
"import/CSVHeaderFiles/relations/patientHasSourceAll-header.csv,import/
relations/patientHasSourceAll.*" --relationships:unstructuredHasSourceAll=
"import/CSVHeaderFiles/relations/unstructuredHasSourceAll-header.csv,import/
relations/unstructuredHasSourceAll.csv" --relationships:patientHasSource=
"import/CSVHeaderFiles/relations/patientHasSource-header.csv,import/
relations/patientHasSource.csv" --relationships:isSubSource="import/
CSVHeaderFiles/relations/isSubSource-header.csv,import/relations/
isSubSource.*" --relationships:hasRelation="import/CSVHeaderFiles/relations/
hasRelation-header.csv,import/relations/relations.*" --relationships:
7412refersTo="import/CSVHeaderFiles/relations/7412refersTo-header.csv,
import/relations/rs7412refersTo.*" --relationships:429358refersTo=
"import/CSVHeaderFiles/relations/429358refersTo-header.csv,import/relations/
429358refersTo.*" --relationships:hasRelation="import/CSVHeaderFiles/
relations/hasEpsCombination-header.csv,import/relations/hasEpsCombination.*"
--multiline-fields=true --ignore-duplicate-nodes=true --ignore-missing-nodes
=true --high-io=true
```

Literaturverzeichnis

[1] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE*, 104(1):11–33, Jan 2016.

[2] Integrative Daten-Semantik für die Neurodegenerationsforschung `https://www.idsn.info/de/idsn.html`. Juli 2020.

[3] Andreas Stefan. Implementierung und Evaluierung eines Polyglot Persistence Entwurfs zum Speichern und Abfragen eines Knowledge Graphen. Master's thesis, Hochschule Bonn-Rhein-Sieg, 11.12.2019.

[4] SNPedia `https://www.snpedia.com/index.php/APOE`. Juli 2020.

[5] Renzo Angles. The Property Graph Databse Model. *Alberto Mendelzon Workshop on Foundations of Data Management*, 2018.

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., USA, 1998.

[7] Reinhard Diestel. *Graph Theory*. Springer-Lehrbuch. Springer, 2017.

[8] Heiko Paulheim. Knowledge Graph Refinement:A Survey of Approaches and EvaluationMethods. *Semantic Web 8 (2017), Nr. 3, 489–508*, 2017.

[9] Peter Pin-Shan Chen. The Entity Relationship Model - Toward a Unified View of Data. *ACM Transaotions on Database Systems, Vol. 1, No. 1*, 1976.

[10] Jens Dörpinghaus and Andreas Stefan. Knowledge Extraction and Applications utilizing Context Data in Knowledge Graphs. *Computer Science and Information Systems pp. 265–272 ISSN 2300-5963 ACSIS, Vol. 18*.

[11] Giannis Nikolentzos, George Dasoulas, and Michalis Vazirgiannis. k-hop Graph Neural Networks, 2019.

[12] Martin Fink. Zentralitätsmaße in komplexen Netzwerken auf Basis kürzester Wege. Master's thesis, Julius-Maximilians-Universität Würzburg: Institut für Informatik, 2009.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[14] Chris Kemper. *Beginning Neo4j*. Apress, 2015.

[15] Gregory Jordan. *Practical Neo4j*. Apress, 2014.

[16] Janna Hastings. *The Gene Ontology Handbook*, chapter Primer on Ontologies, pages 3–13. Springer, 2017.

[17] Ingo Wegener. *Complexity Theory - Exploring the Limits of Efficient Algorithms*. Springer.

[18] Jens Dörpinghaus and Andreas Stefan. Optimization of Retrieval Algorithms on Large Scale Knowledge Graphs, 2020.

[19] Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Rec.*, 41(1):50–60, April 2012.

[20] Amy E. Hodler and Mark Needham. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media, Incorporated, 2019.

[21] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), Sep 2007.

[22] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008.

[23] UCUM- The Unified Code for Units of Measure `http://unitsofmeasure.org`. Juli 2020.

[24] Dublin Core Metadata Initiative `https://www.dublincore.org/specifications/dublin-core/`. Juli 2020.

[25] Import in Neo4j `https://neo4j.com/docs/operations-manual/current/tools/import/`. Juli 2020.

[26] Micha Gorelick and Ian Ozsvald. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, 2014.

[27] Ute Mons, Laura Perna, and Hermann Brenner. Hat der Cholesterinspiegel Einfluss auf die Kognition? *Deutsches Ärzteblatt*, 2016.

[28] Neo4j Community Server Download https://neo4j.com/download-center/#community.

[29] Neo4j Graph Data Science https://neo4j.com/docs/graph-data-science/current/installation/#_neo4j_server.

[30] Alberto O. Mendelzon and Peter T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[31] Neo4j Betweenness Centrality https://neo4j.com/docs/graph-data-science/current/algorithms/betweenness-centrality/. Juli 2020.

[32] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Y. C. Chung, Suhyun Kim, K. Ebcioglu, and E. Altman. LaTTe: a Java VM just-in-time compiler with fast and efficient register allocation. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, pages 128–138, 1999.

[33] Numba - Accelerate Python Functions http://numba.pydata.org/numba-doc/latest/index.html. 2020.

Abbildungsverzeichnis

Tabellenverzeichnis

Eidesstaatliche Erklärung

Name: Tobias Hübenthal

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Unterschrift

30.07.2020

Köln, den