

# **Graphentechnologien in den Digitalen Geisteswissenschaften**

**Modellierung – Import – Analyse**

Andreas Kuczera

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>6</b>
1.1 Warum ein Buch über Graphentechnologien in den digitalen Geisteswissenschaften? . . . . .	6
1.2 Warum ein Buch auf github ? . . . . .	6
1.3 Zum Aufbau des Buches . . . . .	7
<b>2 Was ist eine Graphdatenbank</b>	<b>8</b>
<b>3 Installation und Start</b>	<b>9</b>
<b>4 Das Projekt Regesta Imperii oder “Wie suchen Onlinenutzer Regesten”</b>	<b>10</b>
4.1 Das Projekt Regesta Imperii . . . . .	10
4.2 Die Digitalisierung der Regesta Imperii . . . . .	10
4.3 Wie suchen Online-Nutzer Regesten ? . . . . .	12
4.4 Historische Netzwerkanalyse in den Registern . . . . .	14
4.5 Zusammenfassung . . . . .	16
<b>5 Regestenmodellierung im Graphen</b>	<b>21</b>
5.1 Wie kommen die Regesten in den Graphen . . . . .	21
5.1.1 Import mit dem LOAD CSV-Befehl . . . . .	21
5.1.2 Google-Docs für den CSV-Download . . . . .	23
5.1.3 Regestenmodellierung im Graphen . . . . .	24
5.1.4 Indexe Erstellen . . . . .	24
5.1.5 Erstellen der Regestenknoten . . . . .	26
5.1.6 Erstellen der Ausstellungsorte . . . . .	29
5.1.7 Koordinaten der Ausstellungsorte . . . . .	31
5.1.8 Ausstellungsdatum . . . . .	31
5.2 Exkurs: Herrscherhandeln in den Regesta Imperii . . . . .	32
5.3 Zitationsnetzwerke in den Regesta Imperii . . . . .	32
5.4 Import der Registerdaten in die Graphdatenbank . . . . .	34
5.4.1 Vorbereitung der Registerdaten . . . . .	34
5.4.2 Import der Registerdaten in die Graphdatenbank . . . . .	36
5.4.3 Exkurs: Die Hierarchie des Registers der Regesten Kaiser Friedrichs III. . . . .	37
5.5 Auswertungsperspektiven . . . . .	39
5.5.1 Personennetzwerke in den Registern . . . . .	39
5.5.2 Herrscherhandeln ausgezählt . . . . .	46

5.5.3	Herrscherhandeln pro Ausstellungsort ausgezählt . . . . .	47
5.5.4	Herrscherhandeln und Anwesenheit . . . . .	48
5.5.5	Regesten 200 km rund um Augsburg . . . . .	54
5.5.6	Welche Literatur wird am meisten zitiert . . . . .	54
5.5.7	Der Import zusammengefasst . . . . .	55
5.6	Zusammenfassung . . . . .	55
<b>6</b>	<b>Import von strukturierten XML-Daten in neo4j</b>	<b>56</b>
6.1	Das XML-Beispiel . . . . .	56
6.2	Knotentypen . . . . .	56
6.3	Kantentypen . . . . .	58
6.4	Der Import mit apoc.load.xmlSimple . . . . .	60
6.5	Zusammenfassung . . . . .	63
<b>7</b>	<b>Verwandtschaft im Graphen</b>	<b>64</b>
7.1	Das Projekt Nomen et Gens . . . . .	64
7.2	Nomen et Gens im Graphen . . . . .	64
7.3	Sind Berchar und Karl der Große verwandt ? . . . . .	66
7.4	Zusammenfassung . . . . .	67
<b>8</b>	<b>Das DTA im Graphen</b>	<b>68</b>
8.1	Das deutsche Textarchiv . . . . .	68
8.2	Die Downloadformate des DTA . . . . .	68
8.3	Vorbereitungen . . . . .	70
8.4	Import des TCF-Formats . . . . .	70
8.4.1	Tokenimport . . . . .	70
8.4.2	Satzstrukturen . . . . .	70
8.4.3	Lemmaimport . . . . .	71
8.4.4	Beispielabfrage . . . . .	71
8.5	Import der TEIP5-Fassung . . . . .	72
8.6	Zusammenfassung . . . . .	72
<b>9</b>	<b>Graph-Refactoring mit DTA-XML</b>	<b>74</b>
9.1	Modellierungsüberlegungen . . . . .	74
9.2	Granularität des Modells – Was ist ein Token ? . . . . .	74
9.3	Import der DTA-XML-Daten . . . . .	76
9.4	Erläuterung der entstandenen Graphstrukturen . . . . .	77
9.5	Das DTA-Basisformat im Graphen . . . . .	79
9.6	Layoutstrukturen des Dokuments . . . . .	80
9.6.1	Graphenmodellierung von Zeilen . . . . .	80
9.6.2	Zeilenwechsel mit Worttrennungen . . . . .	82
9.6.3	Seitenfall und Faksimilezählung . . . . .	85
9.6.4	Absätze . . . . .	88
9.6.5	Kapiteleinteilung . . . . .	90

9.7	Editorische Eingriffe . . . . .	93
9.7.1	Hinzufügungen und Tilgungen . . . . .	93
9.7.2	<add>-Element . . . . .	93
9.7.3	Umbau von <add>- und <del>-Elementen in einer <subst>-Umgebung . . . . .	94
9.8	Zusammenfassung . . . . .	96
<b>10</b>	<b>JSON Import mit den Daten der Germania Sacra</b>	<b>98</b>
10.1	Das Projekt Germania Sacra . . . . .	98
10.2	Germania Sacra JSON . . . . .	98
10.2.1	Personen . . . . .	98
10.2.2	Klöster . . . . .	100
10.3	Zusammenfassung . . . . .	102
<b>11</b>	<b>Netzwerkanalyse mit neo4j</b>	<b>104</b>
11.1	Grundlagen zur Netzwerkanalyse . . . . .	104
11.1.1	Vorbemerkungen . . . . .	104
11.1.2	Überblick zu den Netzwerkmaßen . . . . .	104
11.1.3	Beispiel: Zentralitätsmaße . . . . .	105
11.2	Die Register der Regesta Imperii . . . . .	106
11.3	Explorative Datenanalyse oder was ist in der Datenbank . . . . .	107
11.4	Zentralitätsalgorithmen in der historischen Netzwerkanalyse . . . . .	107
11.4.1	PageRank . . . . .	107
11.4.2	Degree Centrality . . . . .	108
11.4.3	Betweenes Centrality . . . . .	108
11.4.4	Closeness Centrality . . . . .	108
11.5	Community Detection Algorithmen . . . . .	108
11.5.1	Strongly Connected Components . . . . .	108
11.5.2	Weakly Connected Components . . . . .	109
11.5.3	Label Propagation . . . . .	109
11.5.4	Louvain Modularity . . . . .	109
11.5.5	Triangle count and Clustering Coefficient . . . . .	109
11.6	Zusammenfassung . . . . .	110
<b>12</b>	<b>Zusammenfassung</b>	<b>111</b>
<b>13</b>	<b>Anhang</b>	<b>112</b>
13.1	cypher-Dokumentation . . . . .	112
13.2	Analyse der Graphdaten . . . . .	112
13.2.1	Welche und jeweils wieviele Knoten enthält die Datenbank . . . . .	112
13.2.2	Welche Verknüpfungen gibt es in der Datenbank und wie häufig sind sie . . . . .	112
13.2.3	Welche Knoten haben keine Kanten . . . . .	113
13.3	Weitere Labels für einen Knoten . . . . .	113

13.4 CSV-Feld enthält mehrere Werte . . . . .	114
13.5 Regluäre Ausdrücke . . . . .	114
13.6 Vorkommende Wörter in einer Textproperty zählen . . . . .	115
13.7 MERGE schlägt fehl da eine Property NULL ist . . . . .	115
13.8 Knoten hat bestimmte Kante nicht . . . . .	116
13.9 Häufigkeit von Wortketten . . . . .	117
13.10 Der WITH-Befehl . . . . .	117
13.11 Die Apoc-Bibliothek . . . . .	117
13.11.1 Installation in neo4j . . . . .	118
13.11.2 Installation unter neo4j-Desktop . . . . .	118
13.11.3 Liste aller Funktionen . . . . .	119
13.11.4 Dokumentation aller Funktionen . . . . .	119
13.12 apoc.xml.import . . . . .	119
13.13 (apoc.load.json) . . . . .	120

# 1 Einleitung

## 1.1 Warum ein Buch über Graphentechnologien in den digitalen Geisteswissenschaften?

Graphentechnologien sind hervorragend für die Modellierung, Speicherung und Analyse hochvernetzter Daten geeignet. Obgleich als Konzept schon länger etabliert erlebten sie mit dem Aufkommen des Internets und der Social-Media-Welle einen Aufschwung. Gegenüber relationalen Datenbankmodellen, bei denen die Daten in miteinander verknüpften Tabellen gespeichert werden, sind die Informationen in Graphdatenbanken in Knoten und Kanten modelliert und auf Speicherebene auch genau so abgelegt. Mit diesem, einer Mind-Map sehr ähnlichen Modell lassen sich Forschungsdaten und Forschungsfragestellungen in einer Weise modellieren, die dem menschlichen Denken oft sehr nahe kommt.

Gerade in den digitalen Geisteswissenschaften gelingt es mit dem Graphenmodell bei der Modellierung und Strukturierung von Forschungsdaten und Forschungsfragestellung die Kluft zwischen Informatiker und Geisteswissenschaftler zu schließen, da der Graph eine für beide Seiten verständliche Plattform bietet. Für den Informatiker ist er hinreichend genau und berechenbar und für den Geisteswissenschaftler wegen seiner Schema- und Hierarchiefreiheit ausreichend flexibel. Gerade diese Eigenschaften, mit denen sich die beiden zentralen Zweige der Digitalen Geisteswissenschaften vereinen lassen, machen Graphen zu einem Schlüsselkonzept der Geisteswissenschaften des 21. Jahrhunderts.

## 1.2 Warum ein Buch auf github ?

Geschwindigkeit Die Digitalisierung unserer Gesellschaft spielt sich in einem rasanten Tempo ab. Diese Geschwindigkeit ist zu einem gewissen Teil auch im Bereich der digitalen Geisteswissenschaften zu spüren. Auch dieses Buch sollte eigentlich ganz traditionell gedruckt werden. Die Entwicklung verläuft aktuell aber so rasant, dass sich allein in der Phase des Lektorats technische Neuentwicklungen ergaben, die berücksichtigt werden mussten. Zum anderen sollte das Buch in einer kostenlosen Open-Access-Fassung zur Verfügung stehen, da die für mich wichtige Zielgruppe der wissenschaftlichen Mitarbeiter und Promovierenden oft finanziell nicht optimal ausgestattet sind.

Die Publikation über Github Pages bietet einerseits ein am Computer gut lesbares Format und die kostenlose Bereitstellung im Internet. Andererseits können aktuelle Entwicklungen

in das Buch eingearbeitet werden. Da die Änderungen in git vorgenommen werden, sind sie für alle Nutzer transparent nachvollziehbar.

### **1.3 Zum Aufbau des Buches**

Das vorliegende Buch ist als Studienbuch konzipiert und richtet sich an Studierende, Lehrende und Wissenschaftler gleichermaßen.

Zu Beginn wird anhand einfacher Beispiele in die Grundlagen der Graphentechnologie eingeführt und die Verwendung von Graphdatenbanken erklärt.

Im nächsten Abschnitt werden Beispiele für die Modellierung und den Import bereits vorhandener Forschungsdaten aus den Projekten Nomen-et-Gens und den Regesta Imperii in eine Graphdatenbank vorgestellt. Anschließend wird auf ein aktuelles Forschungsvorhaben aus dem Bereich Architektur/Geschichtswissenschaften eingegangen, in dessen Verlauf Graphen für die Modellierung der Forschungsdaten und deren anschließende Analyse verwendet wurden.

Im vierten Abschnitt wird der Import von XML-Daten in eine Graphdatenbank anhand von Beispieldaten aus der Epidat-Datenbank des Steinheim-Instituts in Essen erklärt. In einem zweiten Schritt werden die in verschiedenen XML-Varianten vorliegenden Textdaten des Deutschen Textarchivs im Graph modelliert.

Schließlich folgen im Anhang Konzepte zum Graph-Refactoring, zum projektspezifischen Datenimport und zur Struktur von Cypher-Queries.

Mit der Vermittlung der Konzepte von Graphen und Graphdatenbanken werden Kompetenzen in den Bereichen Modellierung (was hängt wie zusammen), Quellenkritik (welche Qualität haben die Informationen in Knoten und Kanten) und der Verknüpfung von wissenschaftlicher Fragestellung und ihrer digitalen Modellierung geschärft. Daher vermittelt dieses Studienbuch dem Leser wichtiges Rüstzeug für die zunehmende Digitalisierung immer größerer, auch die Geisteswissenschaften umfassender, Bereiche der Gesellschaft.

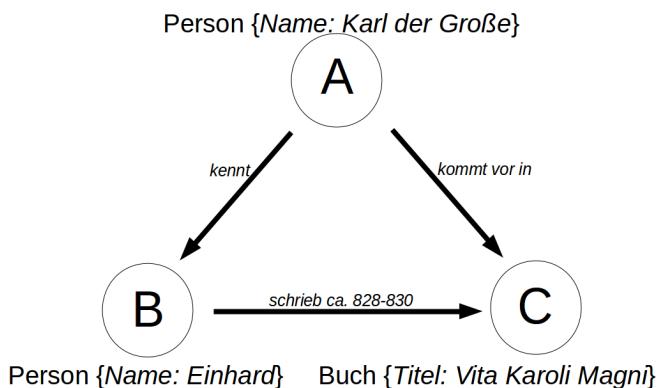
Für Fragen und Rückmeldungen stehe ich gerne zur Verfügung und wünsche beim Studium des Buches viel Freude.

Gießen, im August 2018

Andreas Kuczera

## 2 Was ist eine Graphdatenbank

In den seit den 70er Jahren etablierten, relationalen Datenbanken werden die Daten in Tabellen abgespeichert, die untereinander über Schlüssel oder Schlüsseltabellen verknüpft sind. In Graphdatenbanken werden die Daten dagegen in Knoten und Kanten gespeichert.



Der zeigt oben einen Knoten (engl. Nodes) vom Typ Person mit der Eigenschaft Name. Diese hat den Wert "Karl der Große". Links unten ist ein weiter Knoten vom Typ Person mit dem Namen "Einhard". Rechts unten ist ein Knoten vom Typ Buch und dem Titel "Vita Karoli Magni" abgebildet. Die Kanten (engl. Edges) geben an, dass Karl der Große Einhart kannte, Einhard ca. 828-830 das Buch "Vita Karoli Magni" schrieb und Karl der Große in dem Buch vorkommt.

Knoten und Kanten können also noch zusätzliche Eigenschaften besitzen, in denen weitere Informationen gespeichert sind. Diese Eigenschaften sind spezifisch für die jeweiligen Knotentypen. So sieht man in der Abbildung, dass die beiden Knoten vom Typ Person jeweils noch die Eigenschaft Namen haben, deren Wert dann die Namen der Person angibt, während der Knoten vom Typ Buch die Eigenschaft Titel trägt, in dem der Titel des Buches abgespeichert wird.

## 3 Installation und Start

Informationen zur Installation von neo4j finden Sie auf den Dokumentationsseiten unter <https://neo4j.com/docs/operations-manual/current/installation/>. Für den normalen Nutzer empfiehlt sich die Installation von neo4j-Desktop. Unter <https://neo4j.com/blog/this-week-in-neo4j-getting-started-with-neo4j-desktop-and-browser-graphileon-personal-edition-intuitive-detections-research-with-neo4j/?ref=twitter#features-1> finden sich Videos, in denen die Installation von neo4j-Desktop und erste Schritte im neo4j-Browser erklärt werden.

## 4 Das Projekt Regesta Imperii oder “Wie suchen Onlinenutzer Regesten”

### 4.1 Das Projekt Regesta Imperii

Das Projekt Regesta Imperii wurde von Johann-Friedrich Böhmer im Jahr 1829 begonnen. Ursprünglich als Vorarbeit zu den [Monumenta Germaniae Historica](#) angelegt wurde es mit einem erweitereten Regestenkonzept bald zu einem unverzichtbaren Grundlagenwerk. In den Regesta Imperii werden Inhaltsangaben von Urkunden erstellt, die rechtlich relevante Personen, Inhalte, Orte und Sachverhalte in deutscher Sprache zusammenfassen. Zeitlich umfassen sie den Rahmen von den [Karolingern](#) (7. Jahrhundert) bis Kaiser [Maximilian](#) (gestorben 1519).

#### Beispielbild Urkunden

Ursprünglich von der DFG gefördert sind die Regesta Imperii heute Teil des Bundesländergeförderten Akademienprogramms und werden von der Akademie der Wissenschaften und der Literatur, Mainz, der Berlin-Brandenburgischen Akademie der Wissenschaften und der Akademie der Wissenschaften, Wien betreut.

Die Regesta Imperii arbeiten vor allem herrscherzentriert, d.h. in den Regesten muss der Herrscher eine zentrale Rolle spielen. Bei Urkundenregesten hat er selbst die Urkunde ausgestellt, bei historiographischen Regesten werden den Herrscher betreffende historische Hintergründe zusammengefasst.

In der Kopfzeile des Regests werden der Herrscher sowie Abteilung, Band und Regestennummer genannt. Die darunterliegende Datierungszeile nennt das Ausstellungsdatum der Urkunden und den Handlungs- bzw. Ausstellungsort. Es folgt der Regestentext mit der Zusammenfassung der Urkunde, Hinweise zur Originaldatierung, die Kanzleivermerke und schließlich Angaben zur Überlieferungssituation (Gibt es eine Originalurkunden, wo liegt sie, gibt es ggf. Abschriften etc.).

### 4.2 Die Digitalisierung der Regesta Imperii

Im Rahmen eines von der DFG geförderten Projekts wurden die Regesta Imperii gemeinsam von der Akademie der Wissenschaften, Mainz und der Bayrischen Staatsbibliothek München von 2001 bis 2006 komplett digitalisiert. Alle seit 2006 erschienenen Regesten

**Heinrich IV. - RI III,2,3 n. 1487**

[URI](#)

[Merken](#)

## **1103 Juni 29, Lüttich**

Heinrich feiert das Fest der Apostel, wobei sich Graf Robert von Flandern im Beisein mehrerer Fürsten unterwirft, namentlich der Erzbischöfe Friedrich von Köln und Bruno von Trier, der Bischöfe Otbert von Lüttich, Burchard von Münster, Burchard von Utrecht, Herzog Heinrich von Niederlothringen sowie mehrerer Grafen.

### **Überlieferung/Literatur**

Tagesdatum bei Ann. Patherbr. 1103 ([Scheffer-Boichorst](#) 107 f.): *in festo apostolorum Petri et Pauli; Gesta Galcheri Episcopi Cameracensis* ([SS 14](#), 202); Sigebr. Gembl. 1103 ([SS 6](#), 368); Ann. Elnon. maior. 1103 ([SS 5](#), 14); Ann. Leod., Cont. 1103 ([SS 4](#), 29); Ann. Aquens. 1103 ([SS 16](#), 685); Ann. necrol. Prum. 1103 ([SS 13](#), 223).

### **Kommentar**

Zur Lehensterminologie in den *Gesta Galcheri* (*Facto palam hominio, iurat Robertus Henrico, promittit, miles domino, quia fidelis amodo*) vgl. [G a n s h o f](#), Was ist das Lehnswesen? (1961) 72 f. – Zum Ereigniskontext vgl. Reg. [1475](#); Heinrich V. führte bereits 1107 wieder einen Feldzug gegen Robert von Flandern; vgl. [B o s h o f](#), Bischofskirchen von Passau und Regensburg (Salier 2, 1991) 148. – Vgl. [Kilian](#), Itinerar 127 mit der Vermutung einer weiteren, der Unterwerfung Roberts vorangehenden Heerfahrt Heinrichs nach Flandern; [Meyer von Knonau](#), Jbb. 5, 179 f.; [T. Reuter](#), Unruhestiftung (Salier 3, 1991) 324-326.

Abbildung 4.1: RI III,2,3 n. 1487, in: *Regesta Imperii Online*, URI: <http://www.regesta-imperii.de/id/cf75356b-bd0d-4a67-8aeb-3ae27d1dcefa>.

wurden sofort im Volltext online gestellt. Glücklicherweise hatte die Mainzer Akademie die Rechte selbst inne, so dass der Veröffentlichung als Volltext im Internet keine rechtlichen Hürden im Wege standen. Rückblickend lässt sich feststellen, dass der Absatz der gedruckten Bände nicht gelitten sondern teilweise sogar etwas zugelegt hat.

### 4.3 Wie suchen Online-Nutzer Regesten ?

Ende 2013 wurde das Suchverhalten der Nutzer der Online-Regestensuche im Rahmen eines Vortrages auf der Digital-Diplomatics-Konferenz in Paris in den Blick genommen.<sup>1</sup> Ein interessantes Ergebnis war die Häufigkeitsverteilung der Treffermengen pro Suchanfrage.

Im Tortendiagramm ist die Treffermenge in Zehnerschritten angegeben. Die hellgraue Gruppe oben rechts hat keine Treffer, die dunkelgraue Gruppe einen bis zehn Treffer, die gelbe Gruppe 11 bis 20 usw. Die lila Gruppe hat mehr als hundert Treffer. Überraschend war die große Gruppe mit über 100 Treffern. Hinzu kam, dass über 68% der Nutzer nur ein Suchwort in die Suchmaske eingegeben haben, wobei das beliebteste Suchwort *Heinrich* Ende 2013 zu über 18.000 Treffern führte. Auf der Ergebnisseite hieß es dann: „Sie suchten nach *Heinrich*. Ihre Suche erzielte 18884 Treffer [...] Sie sehen die Treffer 1 bis 20.“

Zusammenfassend könnte man feststellen, dass die Gruppe mit 1 bis 10 Treffern mit ihrem Ergebnis zufrieden war. 10 Regesten lassen sich gut ausdrucken und können anschließend gelesen, ausgewertet und in die eigene Forschungsarbeit einfließen. Die Gruppe mit keinem Treffer hatte möglicherweise die Suche zu sehr eingeschränkt oder einen Tippfehler beim Suchbegriff und wäre lieber in der Gruppe mit einem bis 10 Treffern. Selbstverständlich lassen sich auch 20 und mehr Treffer gut verarbeiten aber bei größeren Treffermengen steigt natürlich auch der Aufwand stark an, so dass davon auszugehen ist, dass die Nutzer kleinere, präzisere Ergebnisse bevorzugen.

Sehr gut lässt sich am Tortendiagramm auch ablesen, dass über die Hälfte unserer Nutzer vor der Suche eine genaue Vorstellung vom Ergebnis haben. Sie sind CIN-Nutzer (concrete information need). Die Gruppe mit über 100 Treffern können der Gruppe der POIN-Nutzer (problem-oriented information need) zugeordnet werden, die problemorientierte Anfragen haben. Für diese Nutzergruppe ist die aktuelle Trefferanzeige der Regestensuche unzureichend, da sie für ihre großen Treffermengen weitere Einschränkungsmöglichkeiten brauchen.<sup>2</sup>

---

<sup>1</sup>Vgl. Kuczera, Andreas; Schrade, Torsten: From Charter Data to Charter Presentation: Thinking about Web Usability in the Regesta Imperii Online. Vortrag auf der Tagung ›Digital Diplomatics 2013 – What ist Diplomatics in the Digital Environment?‹ Folien: <https://prezi.com/vvacmdndthqg/from-charta-data-to-charta-presentation/>.

<sup>2</sup>Näheres dazu in Kuczera, Andreas: Digitale Perspektiven mediävistischer Quellenrecherche, in: Mittelalter. Interdisziplinäre Forschung und Rezeptionsgeschichte, 18.04.2014. URL: [mittelalter.hypotheses.org/3492](http://mittelalter.hypotheses.org/3492).

## Distribution of Result Set Sizes

Result set sized returned for search queries between November 2012 and October 2013 (101220 queries).

- 0 Hits
- 0 - 10 Hits
- 10 - 20 Hits
- 20 - 30 Hits
- 40 - 50 Hits
- 50 - 60 Hits
- 60 - 70 Hits
- 70 - 80 Hits
- 80 - 90 Hits
- 90 - 100 Hits
- > 100 Hits

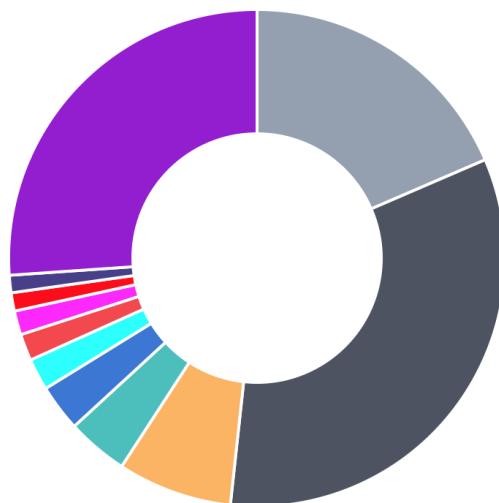


Abbildung 4.2: Treffermengen pro Suchanfragen im Jahr 2013.

## 4.4 Historische Netzwerkanalyse in den Registern

Im Bereich der historischen Netzwerkanalyse gab es in den letzten Jahren sehr interessante Arbeiten.<sup>3</sup> von Seiten der Regesta Imperii bieten sich hier vor allem die Register der Regesta Imperii als sehr interessante Quelle an. Geht man davon aus, dass alle Personen, die gemeinsam in einem Regest genannt sind, etwas miteinander zu tun haben, könnte man auf Grundlage der Registerdaten ein Personennetzwerk erstellen. Über die Qualität der Beziehungen lässt sich nichts sagen und dies schränkt die Aussage der Daten ein. Andererseits stehen wiederum sehr viele Verknüpfungen zur Verfügung.

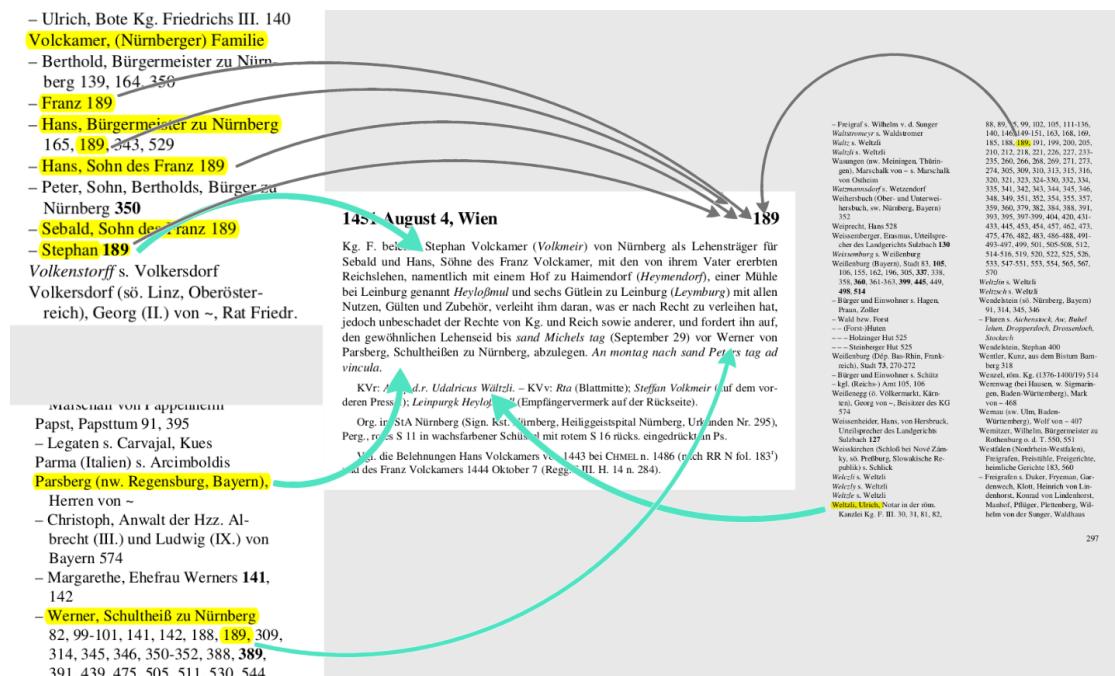


Abbildung 4.3: Registereinträge im Regest als Grundlage für ein Personennetzwerk.

Allein die Einträge in den Registern der Regesten Kaiser Friedrichs III. sind über 143.000 mal in Regesten genannt. Daraus ergeben sich dann über 460.000 1zu1-Beziehungen.<sup>4</sup>

In der folgenden Abbildung sind die in den Registern des Regestenbandes von Joseph

<sup>3</sup>Vgl. beispielsweise Gramsch, Robert: Das Reich als Netzwerk der Fürsten - Politische Strukturen unter dem Doppelkönigtum Friedrichs II. und Heinrichs (VII.) 1225-1235. Ostfildern, 2013. Einen guten Überblick bietet das Handbuch Historische Netzwerkforschung - Grundlagen und Anwendungen. Herausgegeben von Marten Düring, Ulrich Eumann, Martin Stark und Linda von Keyserlingk. Berlin 2016.

<sup>4</sup>Der cypher-Befehl zur Erstellung der 1zu1-Beziehungen lautet: `MATCH (n1:Registereintrag)-[:APPEARS_IN]->(r:Regest)<-[APPEARS_IN]-(n2:Registereintrag) MERGE (n1)-[:KNOWS]->(n2);` Dabei werden die gerichteten KNOWS-Kanten jeweils in beide Richtungen erstellt. Mit folgendem Befehl lassen sich die KNOWS-Kanten zählen: `MATCH p=()-[r:KNOWS]->() RETURN count(p);` Für die Bestimmung der 1zu1-Beziehungen muss der Wert noch durch 2 geteilt werden.

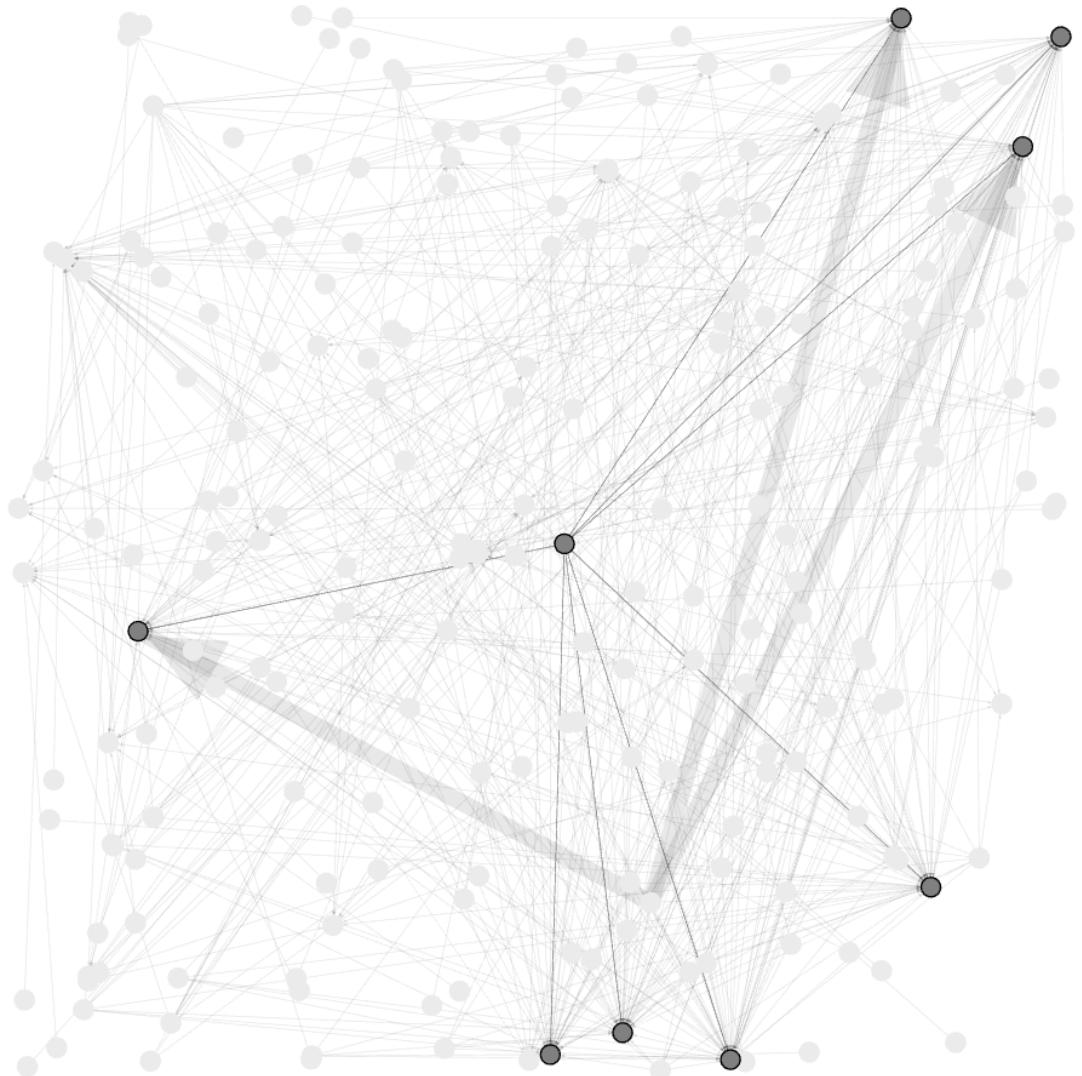


Abbildung 4.4: Ausschnitt der 1zu1-Beziehungen in Gephi.

Chmel gewonnenen 1zu1-Beziehungen mit Gephi visualisiert.<sup>5</sup>

Bei der Analyse ergaben sich aber verschiedene Probleme. Zum einen werden in den Registern auch Kanzleibeamte genannt, die mit der eigentlichen Urkundenhandlung gar nichts zu tun hatten sondern später lediglich ihr Kürzel auf der Urkunde hinterließen. Dies mag archivgeschichtlich interessant sein, für die Urkundenhandlung ist es aber irrelevant. Ein zweites Problem ist der Aufbau des Registers, in dem Orte und Personen in einem Register zusammengefasst werden. Zum einen handelt es sich hierdurch nicht mehr um ein reines Personennetzwerk sondern um ein gemischtes Personen- und Ortsnetzwerk. Zum anderen überragen die über sehr lange Zeit bestehenden Orte, die in ihrer Lebensdauer begrenzten natürlichen Personen in den Netzwerkstrukturen. Schließlich zeigte sich, dass die Algorithmen zur Netzwerkanalyse mit zeitbehafteten Daten (wie Regesten mit ihrem Ausstellungsdatum) nur schlecht umgehen konnten.

Aus Historikersicht war der Ansatz also weniger zielführend jedoch ergaben sich aus Modellierungssicht interessante Einblicke. Um die Netzwerke näher analysieren zu können, wurden kurze Zeitschnitte der Regesten untersucht. Hierfür musste das in Java geschriebene Programm zur Erstellung der Netzwerkdaten jedesmal umgeschrieben werden. Mein Kollege Ulli Meybohm, der das Programm damals betreute, wies mich nach dem wiederholten Umschreiben des Programms darauf hin, dass ich für meine Daten besser eine Graphdatenbank verwenden solle, beispielsweise neo4j. Erste Versuche des Imports der Registerdaten in neo4j erwiesen sich aber als sehr komplex, obwohl das Datenmodell *Person kennt Person* eigentlich relativ einfach ist.

Schließlich ergaben Nachfragen bei neo4j, dass bei Problemen mit dem Datenmodell oft einfach ein Typ von Knoten vergessen worden sein könnte. Und tatsächlich wurden in den ersten Modellen die Regestenknoten nicht berücksichtigt. Mit den Regestenknoten im Modell war der Import schließlich mit weniger rechnerischem Aufwand möglich.

## 4.5 Zusammenfassung

In diesem Kapitel wurde zunächst das Akademieprojekt Regesta Imperii vorgestellt. Seit der Anfang der 2000er Jahre erfolgten Digitalisierung stehen die Regesten unter [www.regesta-imperii.de](http://www.regesta-imperii.de) unter Creative-Commons-Lizenz frei im Internet zur Nutzung zur Verfügung. Für die Auswertung gibt es eine einfache Suchmaske und eine erweiterte Suche. Für die Jahre 2012 und 2013 wurde die Suchstrategien der Nutzer in der Online-Regestensuche untersucht und es zeigte sich, dass sich zwei Nutzungsszenarien unterscheiden lassen, von denen aber nur eines von den aktuellen Suchmasken der Regesta Imperii Online optimal bedient wird. Im zweiten Teil des Kapitels wurde die Visualisierung von Registernetzwerken und die anschließende Modellierung in Graphdatenbanken

---

<sup>5</sup>Regesta chronologico-diplomatica Friderici III. Romanorum imperatoris (regis IV.) : Auszug aus den im K.K. Geheimen Haus-, Hof- und Staats-Archive zu Wien sich befindenden Registraturbüchern vom Jahre 1440 - 1493 ; nebst Auszügen aus Original-Urkunden, Manuscripten und Büchern / von Joseph Chmel, Wien 1838 und 1840.

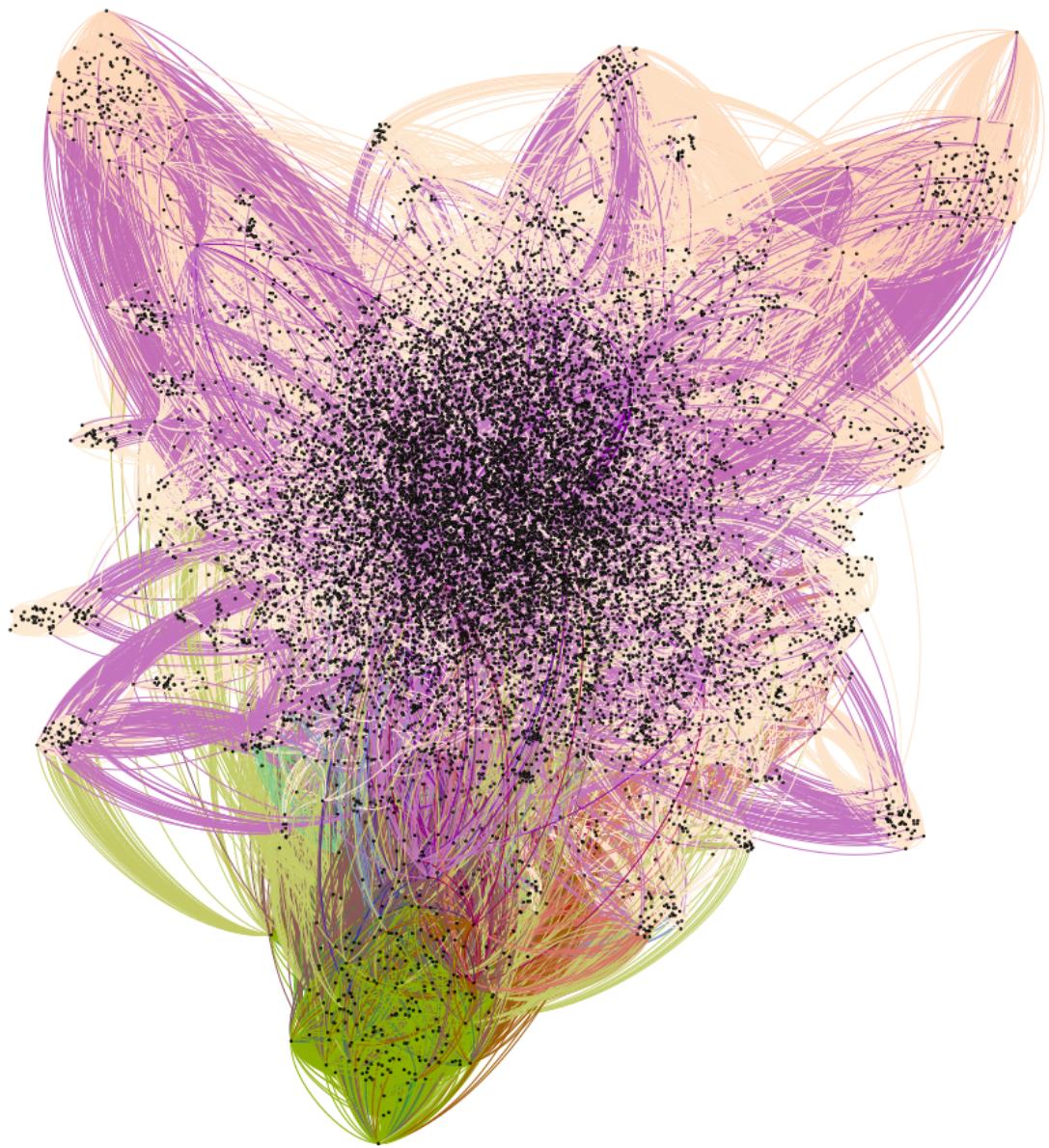


Abbildung 4.5: Personennetzwerk aus den Registern der Regesten Chmels.

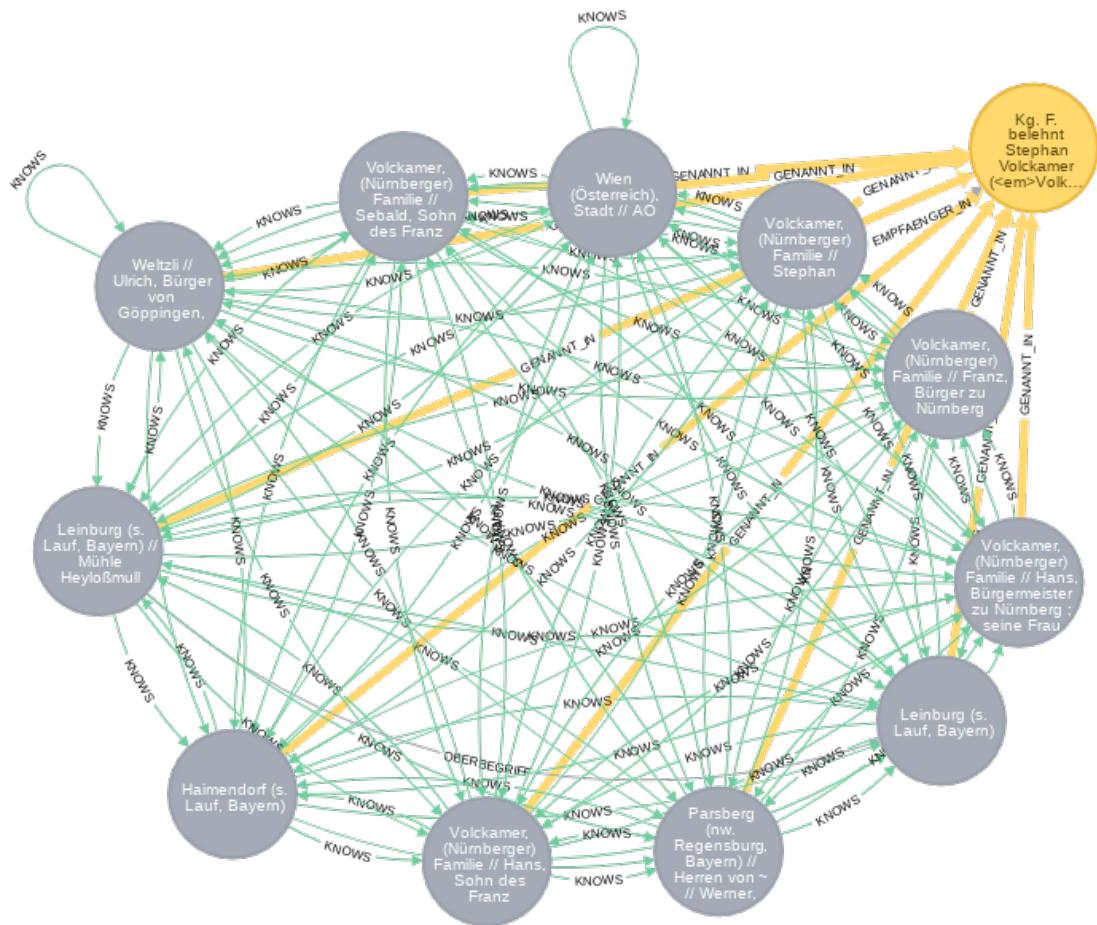


Abbildung 4.6: Regest und Registereinträge mit GENANNT\_IN-Kanten und den KNOWS-Kanten.



Abbildung 4.7: Graphmodell ohne KNOWS-Kanten. Diese können bei Bedarf einfach errechnet werden.

dargestellt und Nutzungs- und Auswertungsszenarien diskutiert. Im folgenden Kapitel wird die Modellierung von Regesten im Graphen detailliert erklärt.

# 5 Regestenmodellierung im Graphen

## 5.1 Wie kommen die Regesten in den Graphen

In diesem Abschnitt wird beispielhaft an Hand der Regesten Kaiser Heinrichs IV. der Import der Online-Regesten in die Graphdatenbank neo4j durchgespielt.<sup>1</sup> Die Webseite der Regesta Imperii Online basiert auf dem Content-Management-System typo3, welches auf eine mysql-Datenbank aufbaut. In der Datenbank werden die Regesteninformationen in verschiedenen Tabellen vorgehalten. Die Webseite bietet zum einen die Möglichkeit, die Regesten über eine REST-Schnittstelle im CEI-XML-Format oder als CSV-Dateien herunterzuladen. Für den Import in die Graphdatenbank bietet sich das CSV-Format an.

In der CSV-Datei finden sich die oben erläuterten einzelnen Elemente der Regesten in jeweils eigenen Spalten. Die Spaltenüberschrift gibt Auskunft zum Inhalt der jeweiligen Spalte.

### 5.1.1 Import mit dem LOAD CSV-Befehl

Mit dem Befehl `LOAD CSV` können die CSV-Dateien mit den Regesten in die Graphdatenbank neo4j importiert werden.<sup>2</sup> Hierfür muss die Datenbank aber Zugriff auf die CSV-Daten haben. Dies ist einerseits über den im Datenbankverzeichnis vorhandene Ordner `import` oder über eine URL, unter der die CSV-Datei abrufbar ist, möglich. Da sich

<sup>1</sup>Die den Regesten Kaiser Heinrichs IV. umfassen folgende Bände: Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1056 (1050) - 1106. 1. Lief.: 1056 (1050) – 1065, bearb. von Struve, Tilman - Köln (u.a.) (1984). Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1056 (1050) - 1106. 2. Lief.: 1065–1075, bearb. von Struve, Tilman unter Mitwirkung von Lubich, Gerhard und Jäckel, Dirk - Köln (u.a.) (2010). Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1056 (1050) - 1106. 3. Lief.: 1076–1085, bearb. von Lubich, Gerhard nach Vorarbeiten von Struve, Tilman unter Mitwirkung von Jäckel, Dirk - Köln (u.a.) (2016). Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1056 (1050) - 1106. 4. Lief.: 1086–1105/06, bearb. von Lubich, Gerhard nach Vorarbeiten von Brauch, Daniel unter Mitwirkung von Weber, Matthias - Köln (u.a.) (2016). Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1056 (1050) - 1106. 5. Lief.: Die Regesten Rudolfs von Rheinfelden, Hermanns von Salm und Konrads (III.). Verzeichnisse, Register, Addenda und Corrigenda, bearbeitet von Lubich, Gerhard unter Mitwirkung von Junker, Cathrin; Klocke, Lisa und Keller, Markus - Köln (u.a.) (2018).

<sup>2</sup>Zu Installation und ersten Schritten von neo4j vgl. in der Einleitung den Abschnitt zu Installation und Start.

Regesten

Datei Bearbeiten Ansicht Einfügen Format Daten Tools Add-ons Hilfe Alle Änderungen in Drive gespeichert

100% € .0 .00 123 Arial 10 B I S A

fx uid

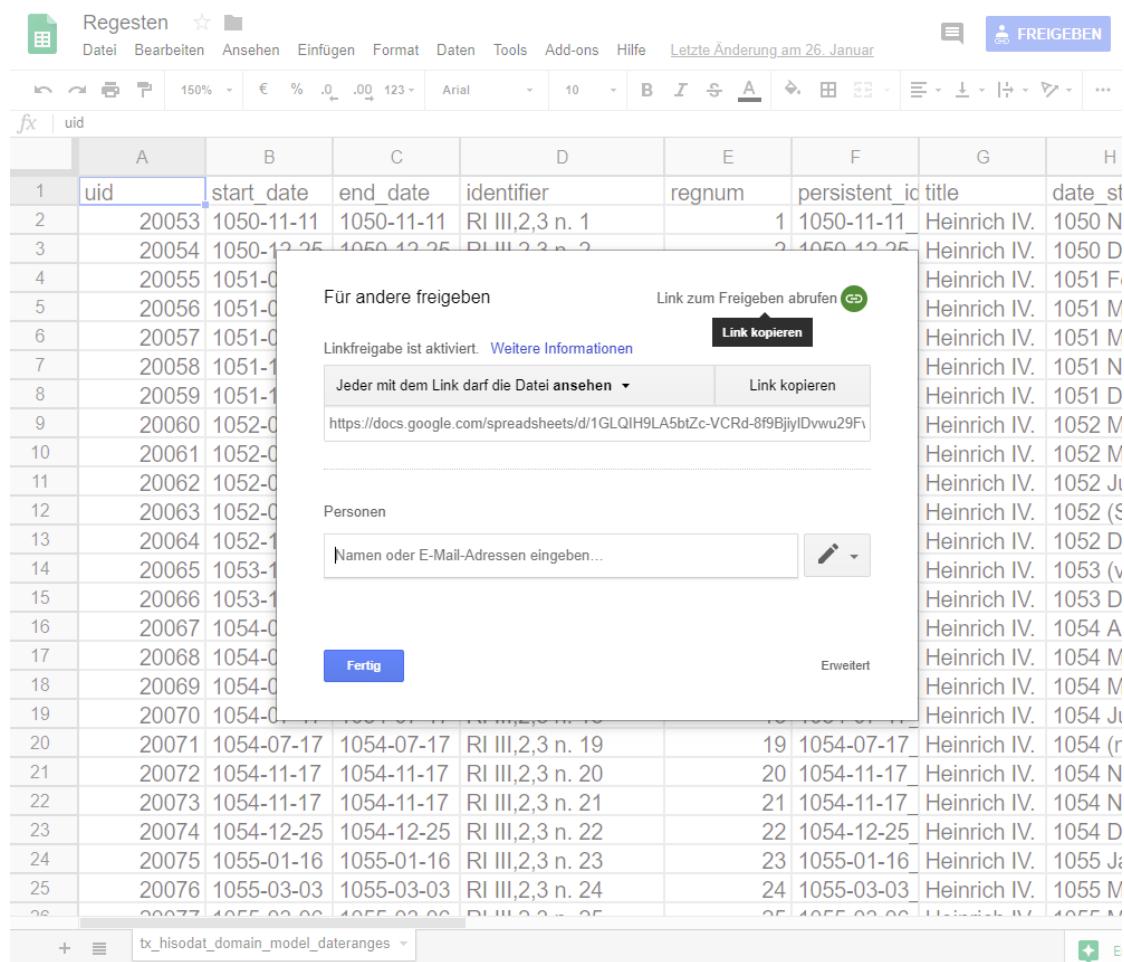
1	uid	start_date	end_date	identifier	regn	persistent_id	title	date_string	locality_string	summary	archival_history	L
2	20053	1050-11-11	1050-11-11	RI III,2,3 n. 1	1	1050-11-11_	Heinrich IV.	1050 Novem (Goslar?)	Heinrich wirc Herim. Aug. 1050			
3	20054	1050-12-25	1050-12-25	RI III,2,3 n. 2	2	1050-12-25_	Heinrich IV.	1050 Dezem Pöhlde	Heinrichs Va Herim. Aug. 1051			
4	20055	1051-02-02	1051-02-02	RI III,2,3 n. 3	3	1051-02-02_	Heinrich IV.	1051 Februa Augsburg	Heinrich folg Herim. Aug. 1051			
5	20056	1051-03-04	1051-03-04	RI III,2,3 n. 4	4	1051-03-04_	Heinrich IV.	1051 März (-) Speyer	Heinrich folg Herim. Aug. 1051			
6	20057	1051-03-31	1051-03-31	RI III,2,3 n. 5	5	1051-03-31_	Heinrich IV.	1051 März 3 Köln	Heinrich emj Herim. Aug. 1051			
7	20058	1051-11-12	1051-11-12	RI III,2,3 n. 6	6	1051-11-12_	Heinrich IV.	1051 Novem Regensburg	Während des Ungarnfeldzuges			
8	20059	1051-12-25	1051-12-25	RI III,2,3 n. 7	7	1051-12-25_	Heinrich IV.	1051 Dezem Goslar	Weihnachtsf Herim. Aug. 1052			
9	20060	1052-03-05	1052-03-05	RI III,2,3 n. 8	8	1052-03-05_	Heinrich IV.	1052 März 5 Kaiserswertf	Heinrich inte <link http://opac.re			
10	20061	1052-03-27	1052-03-27	RI III,2,3 n. 9	9	1052-03-27_	Heinrich IV.	1052 März 2 Goslar	Heinrichs wii <link http://opac.re			
11	20062	1052-06-07	1052-06-07	RI III,2,3 n. 10	10	1052-06-07_	Heinrich IV.	1052 Juni (7) Zürich	Heinrich folg Herim. Aug. 1052			
12	20063	1052-09-01	1052-09-30	RI III,2,3 n. 11	11	1052-09-00_	Heinrich IV.	1052 (Septe -	Heinrichs Elf Herim. Aug. 1052			
13	20064	1052-12-25	1052-12-25	RI III,2,3 n. 12	12	1052-12-25_	Heinrich IV.	1052 Dezem Worms	Weihnachtsf Herim. Aug. 1053			
14	20065	1053-11-03	1053-11-03	RI III,2,3 n. 13	13	1053-11-03_	Heinrich IV.	1053 (vor de Tribur	Heinrich wirc Herim. Aug. 1053			
15	20066	1053-12-25	1053-12-25	RI III,2,3 n. 14	14	1053-12-25_	Heinrich IV.	1053 Dezem Altötting	Heinrich erhü Herim. Aug. 1053			
16	20067	1054-04-03	1054-04-03	RI III,2,3 n. 15	15	1054-04-03_	Heinrich IV.	1054 April (u) Mainz	Heinrich folg Herim. Aug. 1054			
17	20068	1054-05-29	1054-05-29	RI III,2,3 n. 16	16	1054-05-29_	Heinrich IV.	1054 Mai 29 (Goslar?)	Heinrich inte <link http://opac.re			
18	20069	1054-05-31	1054-05-31	RI III,2,3 n. 17	17	1054-05-31_	Heinrich IV.	1054 Mai 31 Goslar	Heinrich inte <link http://opac.re			
19	20070	1054-07-17	1054-07-17	RI III,2,3 n. 18	18	1054-07-17_	Heinrich IV.	1054 Juli 17 Aachen	Heinrich wirc Lampert 1054 (<link			
20	20071	1054-07-17	1054-07-17	RI III,2,3 n. 19	19	1054-07-17_	Heinrich IV.	1054 (nach c (Aachen?)	Vermutlich anlässlich Heinrichs i			
21	20072	1054-11-17	1054-11-17	RI III,2,3 n. 20	20	1054-11-17_	Heinrich IV.	1054 Novem Mainz	Heinrich inte <link http://opac.re			
22	20073	1054-11-17	1054-11-17	RI III,2,3 n. 21	21	1054-11-17_	Heinrich IV.	1054 Novem Mainz	Heinrichs wii <link http://opac.re			
23	20074	1054-12-25	1054-12-25	RI III,2,3 n. 22	22	1054-12-25_	Heinrich IV.	1054 Dezem Goslar	Weihnachtsf Ann. Altah. 1055 (			
24	20075	1055-01-16	1055-01-16	RI III,2,3 n. 23	23	1055-01-16_	Heinrich IV.	1055 Januar Quedlinburg	Heinrichs wii <link http://opac.re			
25	20076	1055-03-03	1055-03-03	RI III,2,3 n. 24	24	1055-03-03_	Heinrich IV.	1055 März 3 Regensburg	Heinrich inte <link http://opac.re			
26	20077	1055-03-06	1055-03-06	RI III,2,3 n. 25	25	1055-03-06_	Heinrich IV.	1055 März 6 Regensburg	Heinrichs wii <link http://opac.re			
27	20078	1055-03-12	1055-03-12	RI III,2,3 n. 26	26	1055-03-12_	Heinrich IV.	1055 März 1 „Utingen“ (?)	Heinrichs wii <link http://opac.re			

Abbildung 5.1: Regesten als CSV-Datei

die einzelnen Zugriffswege auf den **import**-Ordner von Betriebssystem zu Betriebssystem unterscheiden, wird hier beispielhaft der Import über eine URL vorgestellt. Hierfür wird ein Webserver benötigt, auf den man die CSV-Datei hochlädt und sich anschließend die Webadresse für den Download der Datei notiert.

### 5.1.2 Google-Docs für den CSV-Download

Es ist aber auch möglich, CSV-Daten in Google-spreadsheets zu speichern und dort eine URL für den Download der Daten zu erstellen. Zunächst benötigt man hierfür einen Google-Account. Anschließend öffnet man Google-Drive und erstellt dort eine leere Google-Tabellen-Datei (Google-Spreadsheet) in der man dann die CSV-Datei kopieren kann.



The screenshot shows a Google Sheets document titled 'Regesten'. The table has columns labeled 'uid', 'start\_date', 'end\_date', 'identifier', 'regnum', 'persistent\_id', 'title', and 'date\_st'. The data consists of several rows of historical records, such as Heinrich IV. (1050 N) and Heinrich IV. (1051 F). A sharing interface is overlaid on the right side of the table, showing a 'FREIGEBEN' button and a 'Link zum Freigeben abrufen' button. A tooltip for 'Linkfreigabe ist aktiviert' is visible. Below this, there is a 'Link kopieren' button and a copied link URL: <https://docs.google.com/spreadsheets/d/1GLQIH9LA5btZc-VCRd-8f9BjylDvwu29Fv>. There is also a 'Personen' section with a text input field for names or email addresses and a 'Link kopieren' button. At the bottom right of the sharing interface is a 'Fertig' (Done) button.

Abbildung 5.2: Freigabe der Datei zum Ansehen für Dritte!

Wichtig ist nun, die Datei zur Ansicht freizugeben (Klick auf **Freigeben** oben rechts im

Fenster dann **Link zum Freigeben** abrufen und anschließend **Fertig** bestätigen). Jetzt ist die CSV-Datei in Google-Docs gespeichert und kann auch von anderen Personen über den Freigabelink angesehen werden. Für den Import in die Graphdatenbank benötigen wir aber einen Download im CSV-Format. Diesen findet man unter **Datei/Herunterladen als/Kommagetrennte Werte.csv aktuelles Tabellenblatt**.

Damit erhält man das aktuelle Tabellenblatt als CSV-Download. Anschließend muss nun im Browser unter Downloads der Download-Link der Datei gesucht und kopiert werden.

### 5.1.3 Regestenmodellierung im Graphen

Mit dem **LOAD CSV**-Befehl stehen die Informationen der Regestentabelle nun für die weitere Verarbeitung zur Verfügung. Nun muss festgelegt werden, wie diese Informationen im Graphen modelliert werden sollen. Daher wird im nächsten Schritt das Modell der Regesten im Graphen vorgestellt (siehe Abbildung).

In den Abbildungen finden sich beispielhaft das Regest RI III,2,3 Nr. 1487, einmal in der Ansicht der Onlineregisten und in der zweiten Abbildung als Modell im Graphen (neben anderen Regesten).

Die gelben Knoten sind die Regesten. Aus den Angaben des Regests werden mit dem o.a. Befehl noch ein Datumsknoten und ein Ortsknoten erstellt. Mit dem ersten **CREATE**-Befehl werden die Regesten erstellt. Die **MERGE**-Befehle erzeugen ergänzende Knoten für die Datumsangaben und die Ausstellungsorte. Nun ist es aber so, dass Ausstellungsort und Ausstellungsdatum mehrfach vorkommen können. Daher wird hier nicht der **CREATE**-Befehl sondern der **MERGE**-Befehl verwendet. Dieser funktioniert wie der **CREATE**-Befehl, prüft aber vorher, ob in der Datenbank ein solcher Knoten schon existiert. Falls es ihn noch nicht gibt, wird er erzeugt, wenn es ihn schon gibt, wird er der entsprechenden Variable zugeordnet. Anschließend wird die Kante zwischen Regestenknoten und Ausstellungsortsknoten und Regestenknoten und Datumsknoten erstellt. In der folgenden Tabelle werden die einzelnen Befehle dargestellt und kommentiert.

### 5.1.4 Indexe Erstellen

Bevor nun mit dem Import begonnen wird, ist es für die Beschleunigung des Importprozesses von Vorteil vorher Indexe für häufig genutzte Properties zu erstellen.

```
// vorab Index erzeugen -> Import wird schneller
CREATE INDEX ON :Regesta(ident);
CREATE INDEX ON :Regesta(regnum);
CREATE INDEX ON :Regesta(persistentIdentifier);
CREATE INDEX ON :Regesta(registerId);
CREATE INDEX ON :Regesta(heftId);
CREATE INDEX ON :Regesta(placeOfIssue);
```

Regesten

Datei Bearbeiten Ansehen Einfügen Format Daten Tools Add-ons Hilfe Letzte Änderung am 2

Freigeben... Neu Öffnen... Importieren... Kopie erstellen... Herunterladen als Als E-Mail-Anhang senden... Versionsverlauf Umbenennen... Verschieben nach... In Papierkorb verschieben Im Web veröffentlichen... E-Mail an Mitbearbeiter senden... Dokumentdetails... Tabelleneinstellungen... Drucken

00 123 Arial 10 B I S A

C D E

end\_date identifier regnum per

050-11-11 RI III,2,3 n. 1 1 10

050-12-25 RI III,2,3 n. 2 2 10

Microsoft Excel (.xlsx) 3 10

OpenDocument-Format (.ods) 4 10

PDF-Dokument (.pdf) 5 10

Webseite (HTML, komprimiert) 6 10

Kommagetrennte Werte (.csv, aktuelles Tabellenblatt) 7 10

Tabulatorgetrennte Werte (.tsv, aktuelles Tabellenblatt) 8 10

052-03-27 RI III,2,3 n. 9 9 10

052-06-07 RI III,2,3 n. 10 10 10

052-09-30 RI III,2,3 n. 11 11 10

052-12-25 RI III,2,3 n. 12 12 10

053-11-03 RI III,2,3 n. 13 13 10

053-12-25 RI III,2,3 n. 14 14 10

054-04-03 RI III,2,3 n. 15 15 10

20068 1054-05-29 1054-05-29 RI III,2,3 n. 16 16 10

20069 1054-05-31 1054-05-31 RI III,2,3 n. 17 17 10

20070 1054-07-17 1054-07-17 RI III,2,3 n. 18 18 10

20071 1054-07-17 1054-07-17 RI III,2,3 n. 19 19 10

20072 1054-11-17 1054-11-17 RI III,2,3 n. 20 20 10

20073 1054-11-17 1054-11-17 RI III,2,3 n. 21 21 10

20074 1054-12-25 1054-12-25 RI III,2,3 n. 22 22 10

20075 1055-01-16 1055-01-16 RI III,2,3 n. 23 23 10

20076 1055-03-03 1055-03-03 RI III,2,3 n. 24 24 10

20077 1055-03-03 1055-03-03 RI III,2,3 n. 25 25 10

+ tx\_hisodat\_domain\_model\_dateranges

Abbildung 5.3: Herunterladen als CSV-DAtei

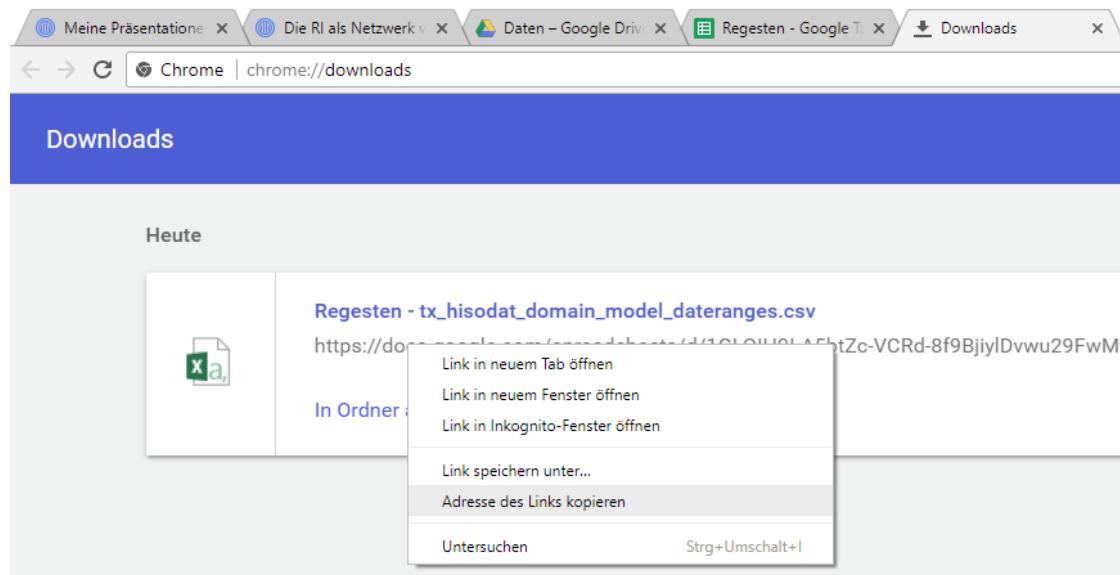


Abbildung 5.4: Download-Link der CSV-Datei

```
CREATE INDEX ON :Regesta(origPlaceOfIssue);
CREATE INDEX ON :Date(startDate);
CREATE INDEX ON :Place(original);
CREATE INDEX ON :Place(normalizedGerman);
CREATE INDEX ON :Lemma(lemma);
CREATE INDEX ON :Literature(literatur);
CREATE INDEX ON :Reference(reference);
CREATE INDEX ON :IndexEntry(registerId);
CREATE INDEX ON :IndexEntry(nodeId);
CREATE INDEX ON :Regesta(latLong);
CREATE INDEX ON :IndexPlace(registerId);
CREATE INDEX ON :IndexEvent(registerId);
CREATE INDEX ON :IndexPerson(registerId);
```

## 5.1.5 Erstellen der Regestenknoten

Mit dem folgenden cypher-Query werden die Regestenknoten in der Graphdatenbank erstellt:

```
// Regestenknoten erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/1GLQIH9LA5btZc-VCRd-8i
CREATE (r:Regesta {regid:line.persistentIdentifier, text:line.summary,
  archivalHistory:line.archival_history, date:line.date_string,
  ident:line.identifier, regnum:line.regnum,
```

**Heinrich IV. - RI III,2,3 n. 1487**

[URI](#)

[Merken](#)

### **1103 Juni 29, Lüttich**

Heinrich feiert das Fest der Apostel, wobei sich Graf Robert von Flandern im Beisein mehrerer Fürsten unterwirft, namentlich der Erzbischöfe Friedrich von Köln und Bruno von Trier, der Bischöfe Otbert von Lüttich, Burchard von Münster, Burchard von Utrecht, Herzog Heinrich von Niederlothringen sowie mehrerer Grafen.

#### **Überlieferung/Literatur**

Tagesdatum bei Ann. Patherbr. 1103 ([Scheffer-Boichorst](#) 107 f.): *in festo apostolorum Petri et Pauli; Gesta Galcheri Episcopi Cameracensis* ([SS 14](#), 202); Sigebr. Gembl. 1103 ([SS 6](#), 368); Ann. Elnon. maior. 1103 ([SS 5](#), 14); Ann. Leod., Cont. 1103 ([SS 4](#), 29); Ann. Aquens. 1103 ([SS 16](#), 685); Ann. necrol. Prum. 1103 ([SS 13](#), 223).

#### **Kommentar**

Zur Lehensterminologie in den *Gesta Galcheri* (*Facto palam hominio, iurat Robertus Henrico, promittit, miles domino, quia fidelis amodo*) vgl. [G a n s h o f](#), Was ist das Lehnswesen? (1961) 72 f. – Zum Ereigniskontext vgl. Reg. [1475](#); Heinrich V. führte bereits 1107 wieder einen Feldzug gegen Robert von Flandern; vgl. [B o s h o f](#), Bischofskirchen von Passau und Regensburg (Salier 2, 1991) 148. – Vgl. [Kilian](#), Itinerar 127 mit der Vermutung einer weiteren, der Unterwerfung Roberts vorangehenden Heerfahrt Heinrichs nach Flandern; [Meyer von Knonau](#), Jbb. 5, 179 f.; [T. Reuter](#), Unruhestiftung (Salier 3, 1991) 324-326.

Abbildung 5.5: RI III,2,3 n. 1487, in: *Regesta Imperii Online*, URI: <http://www.regesta-imperii.de/id/cf75356b-bd0d-4a67-8aeb-3ae27d1dcefa>.

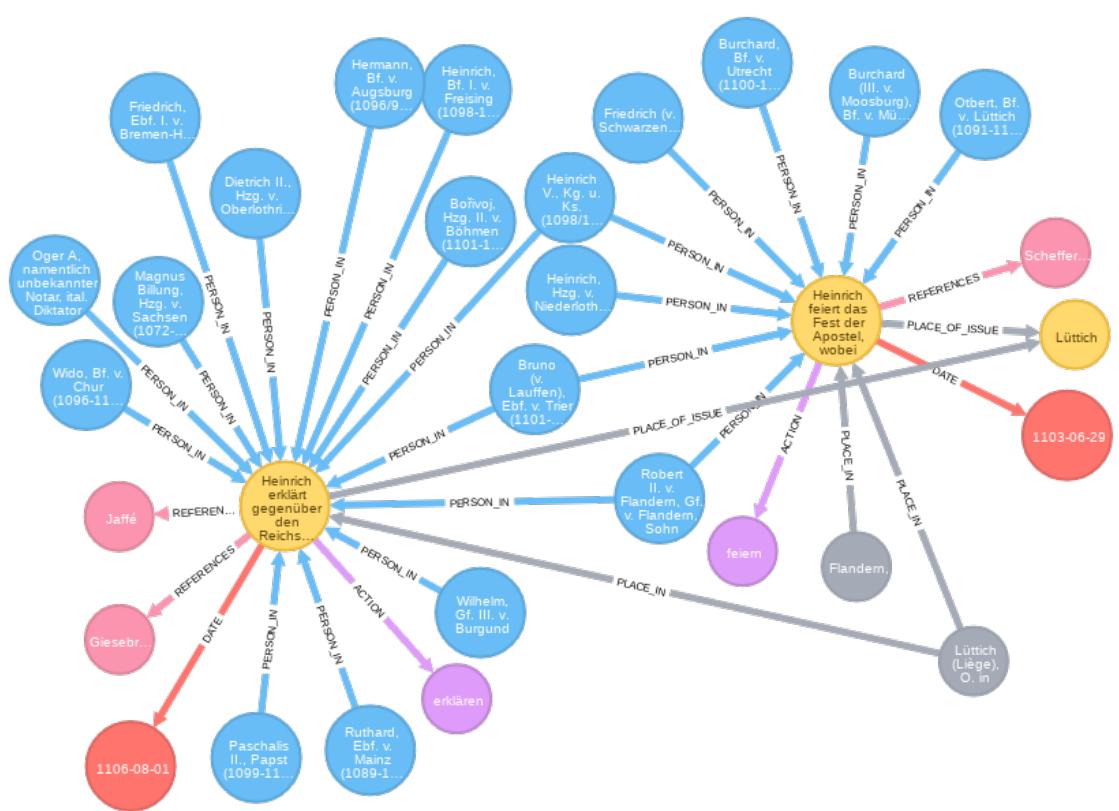


Abbildung 5.6: Das Regest im Graphen.

```

    origPlaceOfIssue:line.locality_string, startDate:line.start_date,
    endDate:line.end_date})
MERGE (d:Date {startDate:line.start_date, enddate:line.end_date})
MERGE (r)-[:DATE]->(d)
RETURN count(r);

```

Im folgenden werden die einzelnen Teile des Import-Befehls erläutert:

Befehl	Variablen	Bemerkungen
LOAD CSV WITH HEADERS FROM “ <a href="https://docs.google.com/">https://docs.google.com/</a> ...” AS line	line	Import der CSV-Dateien. Es wird jeweils eine Zeile an die Variable line weitergegeben
CREATE(r:Regesta {re- gid:line.persistentIdentifier, text:line.summary, archivalHisto- ry:line.archival_history, date:line.date_string MERGE (d:Date {startDate:line.start_date, enddate:line.end_date})	line.persistent_ identifier, line.summary etc. line.start_date und line.end_date	Erstellung des Regestenknotens. Für die weiteren Befehle steht der neu erstellt Regestenknoten unter der Variable <b>r</b> zur Verfügung.
MERGE (r)-[:HAT_DATUM]->(d)	(r) ist der Regestenknoten, (d) ist der Datumsknoten	Es wird geprüft, ob ein Datumsknoten mit der Datumsangabe schon existiert, falls nicht, wird er erstellt. In jedem Fall steht anschließend der Datumsknoten unter der Variable <b>d</b> zur Verfügung. Zwischen Regestenknoten und Datumsknoten wird eine HAT_DATUM-Kante erstellt.

### 5.1.6 Erstellen der Ausstellungsorte

In den Kopfzeilen der Regesta ist, soweit bekannt, der Ausstellungsort der Urkunde vermerkt. Im Rahmen der Arbeiten an den Regesta Imperii Online wurden diese Angaben zusammengestellt und soweit möglich die Orte identifiziert, so dass diese Angabe nun beim Import der Regesta in den Graphen berücksichtigt werden kann. Insgesamt befinden sich in den Regesta Imperii über 12.000 verschiedene Angaben für Ausstellungsorte, wobei sie sich aber auch teilweise auf den gleichen Ort beziehen können (Wie z.B. Aachen, Aquisgrani, Aquisgradi, Aquisgranum, coram Aquisgrano etc.). Allein mit der Identifizierung der 1.000 häufigsten Ortsangaben konnte schon die überwiegende Mehrzahl der Ausstellungsorte georeferenziert werden. Die Daten zur Ortsidentifizierung liegen

auch in einer Google-Tabelle vor.

Mit dem folgenden cypher-Query werden die Ausstellungsorte in die Graphdatenbank importiert:

```
// RI-Ausstellungsorte-geo erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/
13_f6Vja4Hf0pju9RVDubHiMLzS6Uoa7MIOHFEg5V7lw/
export?format=csv&id=13_f6Vja4Hf0pju9RVDubHiMLzS6Uoa7MIOHFEg5V7lw
&gid=420547059"
AS line
WITH line
WHERE line.Lat IS NOT NULL
AND line.normalisiertDeutsch IS NOT NULL
MATCH (r:Regesta {origPlaceOfIssue:line.Original})
MERGE (p:Place {normalizedGerman:line.normalisiertDeutsch,
    longitude:line.Long, latitude:line.Lat})
WITH r, p, line
MERGE (r)-[rel:PLACE_OF_ISSUE]->(p)
SET p.wikidataId = line.wikidataId
SET p.name = line.name
SET p.gettyId = line.GettyId
SET p.geonamesId = line.GeonamesId
SET rel.original = line.Original
SET rel.alternativeName = line.Alternativname
SET rel.commentary = line.Kommentar
SET rel.allocation = line.Zuordnung
SET rel.state = line.Lage
SET rel.certainty = line.Sicherheit
SET rel.institutionInCity = line.InstInDerStadt
RETURN count(p);
```

Da Import-Query etwas komplexer ist, wird er im folgenden näher erläutert. Nach dem `LOAD CSV WITH HEADERS FROM`-Befehl wird zunächst überprüft, ob der jeweils eingelesene Eintrag in der Spalte `line.lat` und in der Spalte `line.normalisiertDeutsch` Einträge hat. Ist dies der Fall wird überprüft, ob es einen Regestenknoten gibt, der einen Ausstellungsorteintrag hat, der der Angabe in der Spalte `Original` entspricht. Diese Auswahl ist notwendig, da in der Tabelle die Ausstellungsorte der gesamten Regesta Imperii enthalten sind. Für diesen Import sollen aber nur jene angelegt werden, die für die Regesten Kaiser Heinrichs IV. relevant sind. Mit dem `MERGE`-Befehl wird der `Place`-Knoten erstellt (falls es ihn nicht schon gibt) und anschließend mit dem Regestenknoten verknüpft. Schließlich werden noch weitere Details der Ortsangabe im `Place`-Knoten und in den `PLACE_OF_ISSUE`-Kanten ergänzt.

### 5.1.7 Koordinaten der Ausstellungsorte

Mit dem folgenden Query werden die Koordinatenangaben zu Höhen- und Breitengraten der Ausstellungsorte (Place-Knoten), die in den Propertys `latitude` und `longitude` abgespeichert sind, in der neuen Property `LatLong` zusammengefasst und in `point`-Werte umgewandelt. Seit Version 3 kann neo4j mit diesen Werten Abstandsberechnungen durchführen (Mehr dazu siehe unten bei den Auswertungen).

```
// Regesten und Ausstellungsorte mit Koordinaten der Ausstellungsorte versehen
MATCH (r:Regesta)-[:PLACE_OF_ISSUE]->(o:Place)
SET r.latLong = point({latitude: tofloat(o.latitude),
    longitude: tofloat(o.longitude)})
SET o.latLong = point({latitude: tofloat(o.latitude),
    longitude: tofloat(o.longitude)})
SET r.placeOfIssue = o.normalizedGerman
SET r.latitude = o.latitude
SET r.longitude = o.longitude;
```

### 5.1.8 Ausstellungsdatum

In den Regesta Imperii Online sind die Datumsangaben der Regesten iso-konform im Format `JJJJ-MM-TT` (also Jahr-Monat-Tag) abgespeichert. neo4j behandelt diese Angaben aber als String. Um Datumsberechnungen durchführen zu können, müssen die Strings in neo4j-interne Datumswerte umgerechnet werden. Der cypher-Query hierzu sieht wie folgt aus:

```
// Date in neo4j-Datumsformat umwandeln
MATCH (n:Regesta)
SET n.isoStartDate = date(n.startDate);
MATCH (n:Regesta)
SET n.isoEndDate = date(n.endDate);
MATCH (d:Date)
SET d.isoStartDate = date(d.startDate);
MATCH (d:Date)
SET d.isoEndDate = date(d.endDate);
```

Zunächst werden mit dem `MATCH`-Befehl alle Regestenknoten aufgerufen. Anschließend wird für jeden Regestenknoten aus der String-Property `startDate` die Datumsproperty `isoStartDate` berechnet und im Regestenknoten abgespeichert. Mit Hilfe der Property können dann Datumsangaben und Zeiträume abgefragt werden (Beispiel hierzu unten in der Auswertung).

## 5.2 Exkurs: Herrscherhandeln in den Regesta Imperii

Bisher wurden beim Import der Regesten in den Graphen nur die in den Online-Regesten bereits angelegten Angaben importiert. Im folgenden Schritt werden nun in einem kleinen Exkurs die Regestentexte selbst analysiert und anschließend die Graphdatenbank um eine weitere Informationsebene ergänzt. Regesten sind in ihrer Struktur stark formalisiert. Meist wird mit dem ersten Verb im Regest das Herrscherhandeln beschrieben. Um dies auch digital auswerten zu können, haben wir in einem kleinen Testprojekt mit Hilfe des Stuttgart-München Treetaggers<sup>3</sup> aus jedem Regest das erste Verb extrahiert und normalisiert. Die Ergebnisse sind in folgender Tabelle einsehbar. Diese Tabelle wird mit dem folgenden cypher-Query in die Graphdatenbank eingelesen.

```
// ReggH4-Herrscherhandeln
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/1nlbZmQYcT1E3Z58yPmcn
AS line FIELDTERMINATOR ',''
MATCH (r:Regesta{ident:line.regid})
MERGE (l:Lemma{lemma:line.Lemma})
MERGE (r)-[:ACTION]->(l);
```

Dabei wird zunächst mit dem MATCH-Befehl das jeweilige Regest gesucht, anschließend mit dem MERGE-Befehl der Lemma-Knoten für das Herrscherhandeln angelegt (falls noch nicht vorhanden) und schließlich der Regesta-knoten mit dem Lemma-Knoten über eine ACTION-Kante verbunden. In der folgenden Abbildung ist ein Ausschnitt mit Regesten und den verknüpften Lemmaknoten dargestellt.

## 5.3 Zitationsnetzwerke in den Regesta Imperii

In vielen Online-Regesten ist die zitierte Literatur mit dem Regesta-Imperii-Opac verlinkt. Da es sich um URLs handelt, sind diese Verweise eindeutig. Andererseits lassen sie sich mit regulären Ausdrücken aus den Regesten extrahieren. Mit folgendem Query werden aus den Überlieferungsteilen der Regesten die mit dem Opac verlinkten Literaturangaben extrahiert und jede Literaturangabe als Reference-Knoten angelegt.

```
// ReggH4-Literaturnetzwerk erstellen
MATCH (reg:Regesta)
WHERE reg.archivalHistory CONTAINS "link"
UNWIND apoc.text.regexGroups(reg.archivalHistory,
"<link (\\"S+)>(\\"S+)</link>") as link
MERGE (ref:Reference {url:link[1]})
ON CREATE SET ref.title=link[2]
MERGE (reg)-[:REFERENCES]->(ref);
```

---

<sup>3</sup>Zum Treetagger vgl. <http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/>.

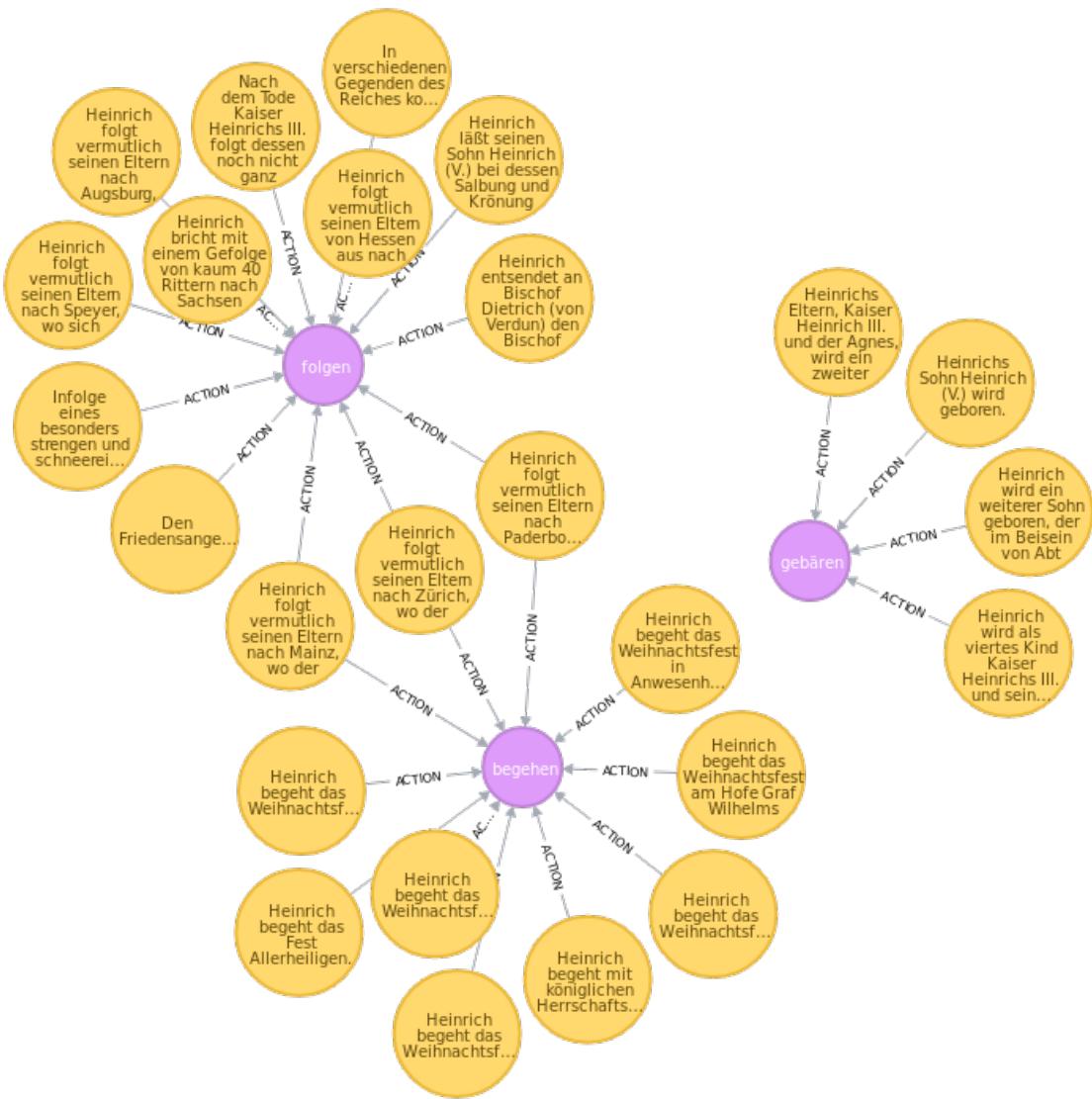


Abbildung 5.7: Herrscherhandeln im Graphen.

Da dies mit dem **MERGE**-Befehl geschieht, wird in der Graphdatenbank jeder Literaturtitel nur einmal angelegt. Anschließend werden die **Reference**-Knoten mit den Regesten über **REFERENCES**-Kanten verbunden. Zu den Auswertungsmöglichkeiten vgl. unten den Abschnitt zu den [Auswertungsperspektiven](#).

## 5.4 Import der Registerdaten in die Graphdatenbank

### 5.4.1 Vorbereitung der Registerdaten

Register spielen für die Erschließung von gedrucktem Wissen eine zentrale Rolle, da dort in alphabetischer Ordnung die im Werk vorkommenden Entitäten (z.B. Personen und Orte) hierarchisch gegliedert aufgeschlüsselt werden. Für die digitale Erschließung der Regesta Imperii sind Register von zentraler Bedeutung, da mit ihnen die in den Regesten vorkommenden Personen und Orte bereits identifiziert vorliegen. Für den Import in die Graphdatenbank wird allerdings eine digitalisierte Fassung des Registers benötigt. Im Digitalisierungsprojekt Regesta Imperii Online wurden Anfang der 2000er Jahre auch die gedruckt vorliegenden Register digitalisiert. Sie dienen nun als Grundlage für die digitale Registererschließung der Regesta Imperii. Im hier gezeigten Beispiel werden die Regesten Kaiser Heinrichs IV. und das dazugehörige Register importiert. Da der letzte Regestenband der Regesten Kaiser Heinrichs IV. mit dem Gesamtregister erst vor kurzem gedruckt wurde, liegen hier aktuelle digitale Fassung von Registern und Regesten vor. Die für den Druck in Word erstellte Registerfassung wird hierfür zunächst in eine hierarchisch gegliederte XML-Fassung konvertiert, damit die Registerhierarchie auch maschinenlesbar abgelegt ist.

In der XML-Fassung sind die inhaltlichen Bereiche und die Abschnitte für die Regestennummern jeweils extra in die Tags **<Inhalt** und **Regestennummer** eingefasst. Innerhalb des Elements **Regestennummer** ist dann nochmal jede einzelne Regestennummer in **<r>**-Tags eingefasst. Die aus dem gedruckten Register übernommenen Verweise sind durch ein leeres **<vw/>**-Element gekennzeichnet.

Die in XML vorliegenden Registerdaten werden anschließend mit Hilfe von TuStep in einzelne CSV-Tabellen zerlegt.

In einer Tabelle werden alle Entitäten aufgelistet und jeweils mit einer ID versehen.

In der anderen Tabelle werden die Verknüpfungen zwischen Registereinträgen und den Regesten aufgelistet. Der Registereintrag Adalbero kommt also in mehreren Regesten vor. Da das Register der Regesten Heinrichs IV. nur zwei Hierarchiestufen enthält, in denen beispielsweise verschiedene Amtsphasen ein und derselben Person unterschieden werden, wurden diese beim Import zusammengefasst.<sup>4</sup> Damit gibt es pro Person jeweils

---

<sup>4</sup>Vgl. die Vorbemerkung zum Register in Böhmer, J. F., Regesta Imperii III. Salisches Haus 1024-1125. Tl. 2: 1056-1125. 3. Abt.: Die Regesten des Kaiserreichs unter Heinrich IV. 1050 - 1106. 5. Lief.: Die Regesten Rudolfs von Rheinfelden, Hermanns von Salm und Konrads (III.). Verzeichnisse,

```

<Stufe0 id="H4P00005">
  <Inhalt>Abiram, biblische Gestalt, Sohn Eliabs, Bruder Dathans, Auflehner gegen
  Moses</Inhalt>
  <Regestennummer>
    <r>762</r>
  </Regestennummer>
</Stufe0>
<Stufe0 id="H4P00006">
  <Inhalt>AC → Gottschalk v. Aachen</Inhalt>
</Stufe0>
<Stufe0 id="H4P00007">
  <Inhalt>Achalmer, Adelsgeschlecht aus Schwaben</Inhalt>
  <Regestennummer>
    <r>363</r>
  </Regestennummer>
  <Stufe1>
    <Inhalt><vw/>- Liutold, Gf. v. Achalm</Inhalt>
  </Stufe1>
  <Stufe1>
    <Inhalt><vw/>- Werner (v. Achalm), Bf. II. v. Straßburg</Inhalt>
  </Stufe1>
  <Stufe1>
    <Inhalt><vw/>- Williberga</Inhalt>
  </Stufe1>
</Stufe0>

```

Abbildung 5.8: Ausschnitt aus dem XML-Register der Regesten Heinrichs IV.

6	H4P00005	Abiram, biblische Gestalt, Sohn Eliabs, Bruder Dathans, Auflehner gegen Moses		
7	H4P00006	AC → Gottschalk v. Aachen		
8	H4P00007	Achalmer, Adelsgeschlecht aus Schwaben		
9	H4P00008	Aczo, Sohn Rudolf Rubias, Bruder Attos		
10	H4P00009	Adalbero, Mgf. d. karantanischen Mark		
11	H4P00010	Adalbero, Gf. v. Ebersberg, Gem. Richildes		
12	H4P00011	Adalbero, Edelfreier		
13	H4P00012	Adalbero A (AA), namentlich unbekannter Bamberger Diktator		

Abbildung 5.9: Ausschnitt der Entitätentabelle des Registers der Regesten Heinrichs IV.

ID	regnum	regnum2	name1	name2			
H4P00001	805	805		<i>A,</i>,			
H4P00002	1517	1517		<i>A,</i>,			
H4P00003	1519	1519		<j>A.</i>, <i>centurio</i>			
H4P00005	762	762		Abiram, biblische Gestalt, Sohn Eliabs, Bruder Dathans, Auflehrer gegen Moses			
H4P00007	363	363		Achalmer, Adelsgeschlecht aus Schwaben			
H4P00008	1056	1056		Aczo, Sohn Rudolf Rubias, Bruder Attos			
H4P00009	714	714		Adalbero, Mgf. d. karantanischen Mark			
H4P00010	78	78		Adalbero, Gf. v. Ebersberg, Gem. Richildes			
H4P00011	331	331		Adalbero, Edelfreier			
H4P00012	666	666		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	717	717		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	959	959		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	1400	1400		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	1403	1403		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	1420	1420		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	1481	1481		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			
H4P00012	1485	1485		Adalbero A (AA), namentlich unbekannter Bamberger Diktator			

Abbildung 5.10: Ausschnitt der Verknüpfungstabelle des Registers der Regesten Heinrichs IV.

nur einen Knoten.

#### 5.4.2 Import der Registerdaten in die Graphdatenbank

Im Gegensatz zu den Regesten Kaiser Friedrichs III., bei denen Orte und Personen in einem Register zusammengefasst sind, haben die Regesten Kaiser Heinrich IV. getrennte Orts- und Personenregister. Die digitalisierten Registerdaten können [hier](#) eingesehen werden. In dem Tabellendokument befinden sich insgesamt drei Tabellen. In der Tabelle Personen sind die Einträge des Personenregisters aufgelistet und in der Tabelle Orte befindet sich die Liste aller Einträge des Ortsregisters. Schließlich enthält die Tabelle **APPEARS\_IN** Information dazu, welche Personen oder Orte in welchen Regesten genannt sind. Der folgende cypher-Query importiert die Einträge der Personentabelle in die Graphdatenbank und erstellt für jeden Eintrag einen Knoten vom Typ :IndexPerson:

```
// Registereinträge Personen erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/12T-RD1Ct4aAUNNNxipjM
AS line
CREATE (:IndexPerson {registerId:line.ID, name1:line.name1});
```

Mit dem folgenden cypher-Query werden nach dem gleichen Muster aus der Tabelle **Orte** die Ortseinträge in die Graphdatenbank importiert.

```
// Registereinträge Orte erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/12T-RD1Ct4aAUNNNxipjM
AS line
```

---

Register, Addenda und Corrigenda, bearbeitet von Lubich, Gerhard unter Mitwirkung von Junker, Cathrin; Klocke, Lisa und Keller, Markus - Köln (u.a.) (2018), S. 291.

```
CREATE (:IndexPlace {registerId:line.ID, name1:line.name1});
```

Die beiden Befehle greifen also auf verschiedene Tabellenblätter des gleichen Google-Tabellendokuments zu, laden es als CSV-Daten und übergeben die Daten zeilenweise an die weiteren Befehle (Hier an den MATCH- und den CREATE-Befehl). Im nächsten Schritt werden nun mit den Daten der APPEARS\_IN-Tabelle die Verknüpfungen zwischen den Registereinträgen und den Regesten erstellt.

```
// PLACE_IN-Kanten für Orte erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/12T-RD1Ct4aAUNNNxipjM
AS line
MATCH (from:IndexPlace {registerId:line.ID})
MATCH (to:Regesta {regnum:line.regnum2})
CREATE (from)-[:PLACE_IN {regnum:line.regnum, name1:line.name1, name2:line.name2}]->(to)
```

Mit zwei MATCH-Befehlen wird jeweils das Regest und der Registereintrag aufgerufen und mit dem CREATE-Befehl eine PLACE\_IN-Kante zwischen den beiden Knoten angelegt, die als Attribute den Inhalt der Spalten name1 und name2 erhält. Analog werden die Verknüpfungen zwischen Regestenknoten und Personenknoten angelegt:

```
// PERSON_IN-Kanten für Person erstellen
LOAD CSV WITH HEADERS FROM "https://docs.google.com/spreadsheets/d/12T-RD1Ct4aAUNNNxipjM
AS line
MATCH (from:IndexPerson {registerId:line.ID}), (to:Regesta {regnum:line.regnum2})
CREATE (from)-[:PERSON_IN {regnum:line.regnum, name1:line.name1, name2:line.name2}]->(to)
```

### 5.4.3 Exkurs: Die Hierarchie des Registers der Regesten Kaiser Friedrichs III.

In anderen Registern der Regesta Imperii, wie beispielsweise den Regesten Kaiser Friedrichs III., sind teilweise fünf oder mehr Hierarchiestufen vorhanden, die jeweils auch Entitäten repräsentieren.

In diesen Fällen müssen die Hierarchien auch in der Graphdatenbank abgebildet werden, was durch zusätzliche Verweise auf die ggf. vorhandenen übergeordneten Registereinträge möglich wird.

Im Tabellenausschnitt wird jedem Registereintrag in der ersten Spalte eine `nodeID` als eindeutige Kennung zugewiesen. Bei Registereinträgen, die kein Hauptlemma sind, enthält die dritte Spalte `topnodeID` den Verweis auf die eindeutige Kennung `nodeID` des übergeordneten Eintrages. Beim Import in die Graphdatenbank wird diese Hierarchie über CHILD\_OF-Kanten abgebildet, die vom untergeordneten Eintrag auf das übergeordnete Lemma verweisen. Damit ist die komplette Registerhierarchie im Graphen abgebildet. In der Spalte `name1` ist das Lemma angegeben. In der Spalte `name3` ist zusätzlich zum Lemma noch der gesamte Pfad vom Hauptlemma bis zum Registereintrag, jeweils mit Doppelslashes (//) getrennt. Bei tiefer gestaffelten Registern ist teilweise ohne Kenntnis

- Bamberg (Bayern), Stadt
- Bürger und Einwohner s. Herr
  - Bischof 178
  - Anton (von Rotenhan) (1431-1459) **29, 35, 45, 46, 177, 216, 238, 564**
  - Domkapitel **35**
    - Domdekan s. Rotenhan
      - Gericht des Domdekans 29
    - Dompropst 375, **376, 377**
      - Georg von Schaumberg **45, 46, 410, 411-413, 426, 430**
      - Lehengericht des Dompropstes 410-413, 426, 430
        - Lehenrichter s. Truchseß
    - Tag 20, 30, 31, 102

Abbildung 5.11: Ausschnitt aus dem Register des Heftes 19 der Regesten Kaiser Friedrichs III.

nodeID	xmlID	topnodeID	name1	name3
1	A00000001		Aa, Johann von	Aa, Johann von ~
2	A00000002	1	Sophie von ~, Tc	Aa, Johann von ~ // Sophie von ~, Tochter Johanns, Bürgerin zu Köln
3	A00000003		Aach	Aach (Fluß durch Aach, n. Singen, Baden-Württemberg)
4	A00000004		Aach	Aach (n. Singen, Baden-Württemberg), Stadt
5	A00000005		Aache s. Aacher	Aache s. Aachen
6	A00000006		Aachen	Aachen (Aache; Nordrhein-Westfalen), Stadt
7	A00000007	6	Einwohner und B	Aachen (Aache; Nordrhein-Westfalen), Stadt // Einwohner und Bürger ; s. Col
8	A00000008	6	Fischmarkt	Aachen (Aache; Nordrhein-Westfalen), Stadt // Fischmarkt (Parwisch)
9	A00000009	6	Gerichte	Aachen (Aache; Nordrhein-Westfalen), Stadt // Gerichte
10	A00000010	6	Kurgericht	Aachen (Aache; Nordrhein-Westfalen), Stadt // Gerichte // Kurgericht
11	A00000011	6	Schöffenstuhl, k	Aachen (Aache; Nordrhein-Westfalen), Stadt // Gerichte // Schöffenstuhl, kgl.
12	A00000012	6	Richter und Sch	Aachen (Aache; Nordrhein-Westfalen), Stadt // Gerichte // Schöffenstuhl, kgl.
13	A00000013	6	Schöffen	Aachen (Aache; Nordrhein-Westfalen), Stadt // Gerichte // Schöffenstuhl, kgl.
14	A00000014	6	„Grafschaften“	Aachen (Aache; Nordrhein-Westfalen), Stadt // „Grafschaften“
15	A00000015	6	„Grasgebot“	Aachen (Aache; Nordrhein-Westfalen), Stadt // „Grasgebot“
17	A00000017	6	Meierei	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei
19	A00000019	6	Brothaus	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Brothaus
21	A00000021	6	Gewandhaus	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Gewandhaus
22	A00000022	6	Grashaus	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Grashaus
23	A00000023	6	Plankenhaus	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Plankenhaus
24	A00000024	6	Tuchhaus	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Tuchhaus
25	A00000025	6	Haus zum Haner	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // Haus zum Haner (zu
26	A00000026	6	zo der Geiss	Aachen (Aache; Nordrhein-Westfalen), Stadt // Meierei // zo der Geiss
27	A00000027	6	Rentmeister	Aachen (Aache; Nordrhein-Westfalen), Stadt // Rentmeister
29	A00000029	6	„Stadtbücher“	Aachen (Aache; Nordrhein-Westfalen), Stadt // „Stadtbücher“
30	A00000030	6	Vogtei	Aachen (Aache; Nordrhein-Westfalen), Stadt // Vogtei

Abbildung 5.12: Ausschnitt der Entitätentabelle des Registers der Regesten Friedrichs III.

der übergeordneten Einträge eine eindeutige Identifizierung eines Eintrages nicht möglich. So wird in Zeile 17 der o.a. Abbildung allein mit der Angabe aus der Spalte `name1` nicht klar, um welche `Meierei` es sich handelt. Mit dem kompletten Pfad des Registereintrages in der Spalte `name3` wird dagegen deutlich, dass die Aachener `Meierei` gemeint ist.

## 5.5 Auswertungsperspektiven

### 5.5.1 Personennetzwerke in den Registern

#### 5.5.1.1 Graf Robert II. von Flandern in seinem Netzwerk

Nach dem Import können nun die Online-Regesten und die Informationen aus den Registern der Regesten Kaiser Heinrichs IV. in einer Graphdatenbank aus einer Vernetzungsperspektive abgefragt werden.<sup>5</sup>

Ausgangspunkt ist der Registereintrag von `Graf Robert II. von Flandern`. Diesen Knoten finden wir mit folgendem Query.

<sup>5</sup>Die nun folgenden Abfragen sind zum Teil einer Präsentation entnommen, die für die Summerschool der [Digitalen Akademie](#) im Rahmen des [Mainzed](#) entwickelt wurden. Die Präsentation findet sich unter der URL <https://digitale-methodik.adwmainz.net/mod5/5c/slides/graphentechnologien/RI.html>.

```
// Robert II. von Flandern
MATCH (n:IndexPerson) WHERE n.registerId = 'H4P01822'
RETURN *;
```

Mit einem Doppelklick auf den `IndexPerson`-Knoten öffnen sich alle `Regesta`-Knoten, in denen Robert genannt ist. Klickt man nun wiederum alle Regestenknoten doppelt an, sieht man alle Personen und Orte, mit denen Robert gemeinsam in den Regesten genannt ist.

Dies kann auch in einem cypher-Query zusammengefasst werden.

```
// Robert II. von Flandern mit Netzwerk
MATCH (n:IndexPerson)-[:PERSON_IN]->
(r:Regesta)<-[PERSON_IN]-
(m:IndexPerson)
WHERE n.registerId = 'H4P01822'
RETURN *;
```

In der folgenden Abb. wird das Ergebnis dargestellt.

Hier wird der `MATCH`-Befehl um einen Pfad über `PERSON_IN`-Kanten zu `Regesta`-Knoten ergänzt, von denen jeweils eine `PERSON_IN`-Kante zu den anderen, in den Regesten genannten `IndexPerson`-Knoten führt.

Nimmt man noch eine weitere Ebene hinzu, wächst die Ergebnismenge stark an. Der folgende Query kann daher je nach Rechnerleistung etwas länger dauern.

```
// Robert II. von Flandern mit Netzwerk und Herrscherhandeln (viel)
MATCH
(n1:IndexPerson)-[:PERSON_IN]->(r1:Regesta)<-[PERSON_IN]-
(n2:IndexPerson)-[:PERSON_IN]->(r2:Regesta)<-[PERSON_IN]-
(n3:IndexPerson)
WHERE n1.registerId = 'H4P01822'
RETURN *;
```

### 5.5.1.2 Graf Robert II. von Flandern und Herzog Heinrich von Niederlothringen

In der Graphdatenbank ist es über die Exploration der Beziehungen einer Person hinaus möglich, explizit die Verbindungen von zwei Personen abzufragen. In unserem nächsten Beispiel suchen wir jene Regesten, in denen [Graf Robert II. von Flandern](#) und [Herzog Heinrich von Niederlothringen](#) gemeinsam genannt sind.

```
// Robert II. von Flandern und Herzog Heinrich von Niederlothringen mit Netzwerk
MATCH
(n:IndexPerson)-[:PERSON_IN]->
(r:Regesta)<-[PERSON_IN]-(m:IndexPerson)
WHERE n.registerId = 'H4P01822'
```



Abbildung 5.13: Robert mit den Personen, mit denen er gemeinsam in Regesten genannt wird.

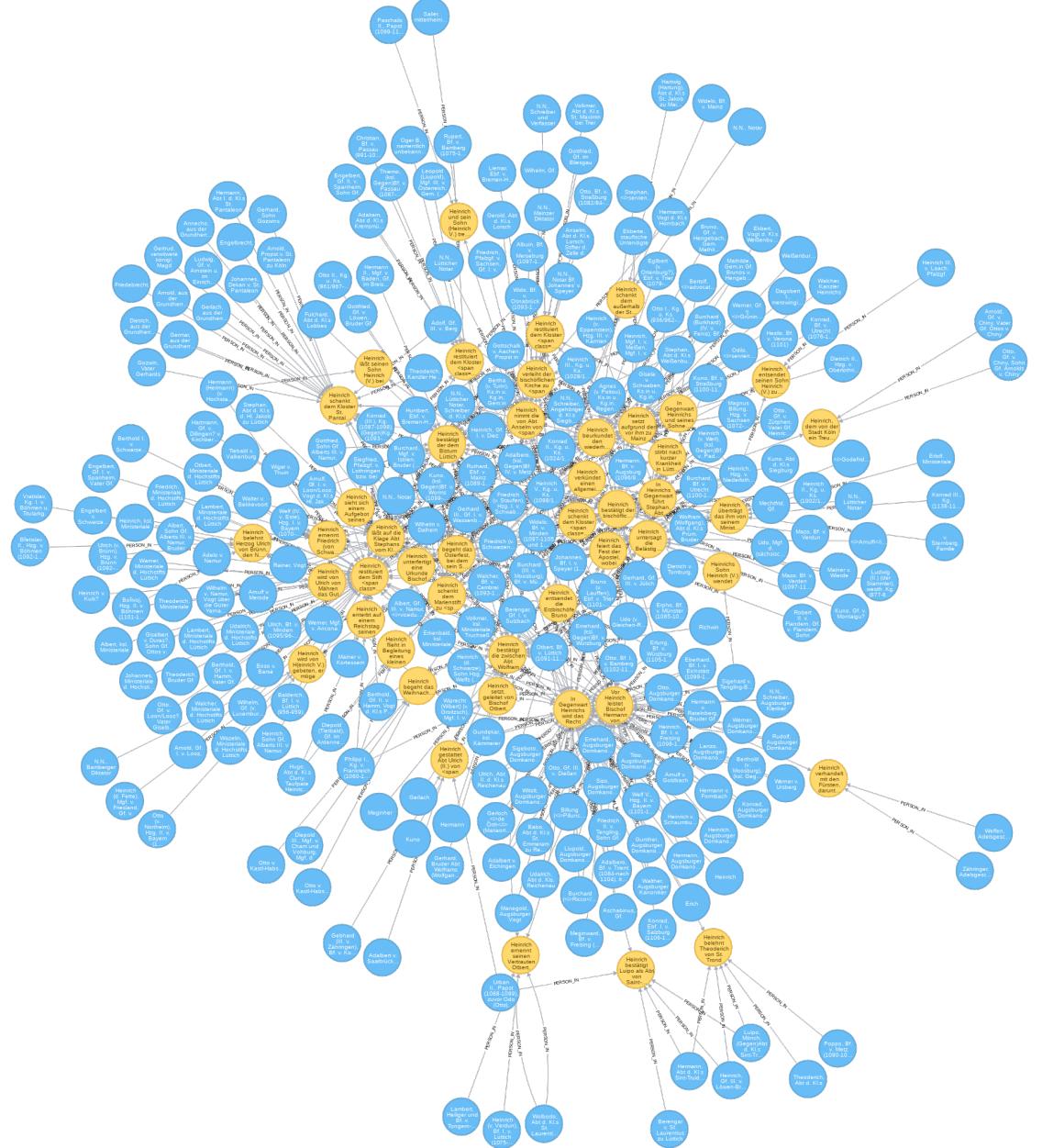


Abbildung 5.14: Robert mit Personen, die wiederum mit Personen gemeinsam in Regesten genannt sind.

```
AND m.registerId = 'H4P00926'
RETURN *;
```



Abbildung 5.15: Robert und Heinrich sind in einem Regest gemeinsam genannt.

Es zeigt sich, dass Robert und Heinrich in einem Regest gemeinsam genannt sind.

Und dieses [Regest](#) berichtet von der Unterwerfung Roberts unter Heinrich IV.<sup>6</sup>

Heinrich feiert das Fest der Apostel, wobei sich Graf Robert von Flandern im Beisein mehrerer Fürsten unterwirft, namentlich der Erzbischöfe Friedrich von Köln und Bruno von Trier, der Bischöfe Otbert von Lüttich, Burchard von Münster, Burchard von Utrecht, Herzog Heinrich von Niederlothringen sowie mehrerer Grafen.

Möglicherweise haben beide aber gemeinsame Bekannte, also Personen mit denen sowohl Heinrich als auch Robert in unterschiedlichen Regesten gemeinsam genannt sind. Hierfür wird der cypher-Query um eine Ebene erweitert.

```
// Robert und Heinrich mit allen gemeinsamen Personen und Regesten
MATCH (n1:IndexPerson)
- [:PERSON_IN] -> (r1:Regesta) <- [:PERSON_IN] -
(n2:IndexPerson) - [:PERSON_IN] -> (r2:Regesta)
<- [:PERSON_IN] -> (n3:IndexPerson)
WHERE n1.registerId = 'H4P00926'
AND n3.registerId = 'H4P01822'
RETURN *;
```

Ein erster Blick auf das Ergebnis zeigt, dass Heinrich allgemein besser vernetzt ist. Für die weitere Analyse ihres Verhältnisses ist nun die Lektüre der angegebenen Regesten notwendig. Hierfür lässt sich das Ergebnis noch etwas weiter aufbereiten, indem die zwischen den Personen liegenden Regesten in KNOWS-Kanten umgewandelt werden, die als zusätzliche Information die Angaben zu den Regesten enthalten.

```
// Rausrechnen der dazwischenliegenden Knoten
MATCH
(startPerson:IndexPerson) - [:PERSON_IN] ->
(regest:Regesta) <- [:PERSON_IN] -> (endPerson:IndexPerson)
```

---

<sup>6</sup>Vgl. RI III,2,3 n. 1487.

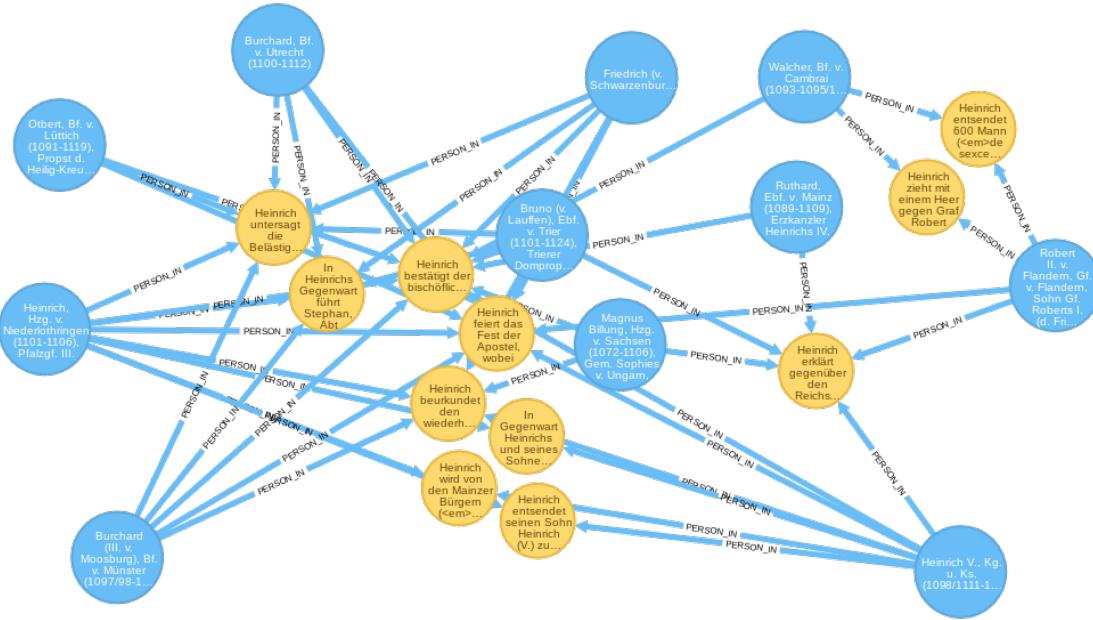


Abbildung 5.16: Robert und Heinrich mit den gemeinsamen Bekanntschaften.

```

WHERE startPerson.registerId in ['H4P01822', 'H4P00926']
WITH startPerson, endPerson, count(regestd) as anzahl,
collect(regestd.ident) as idents
CALL apoc.create.vRelationship(startPerson, "KNOWS",
{anzahl:anzahl, regesten:idents}, endPerson) YIELD rel
RETURN startPerson, endPerson, rel;

```

In der Abbildung sind die zwei Ego-Netzwerke von Heinrich (links) und Robert (rechts) mit den dazwischen liegenden gemeinsamen Bekanntschaften dargestellt. Es zeigt sich, dass Heinrich stärker sowohl mit Geistlichen als auch Weltlichen vernetzt war, während Robert insgesamt weniger Kontakte aber mit einem Schwerpunkt in der Geistlichkeit hatte.

Für den Historiker ist aber vor allem interessant, was in den Regesten steht, die Robert und Heinrich über die Mittelsmänner verbinden. Hierfür wird der cypher-Query angepasst und sowohl Personen als auch die Regestentexte ausgegeben.

```

// Liste der Regesten als Ergebnis
MATCH
(startPerson:IndexPerson)-[:PERSON_IN]->
(regest1:Regesta)<-[PERSON_IN]-(middlePerson:IndexPerson)
-[:PERSON_IN]->(regest2:Regesta)
-<[:PERSON_IN]-(endPerson:IndexPerson)
WHERE startPerson.registerId in ['H4P00926']

```

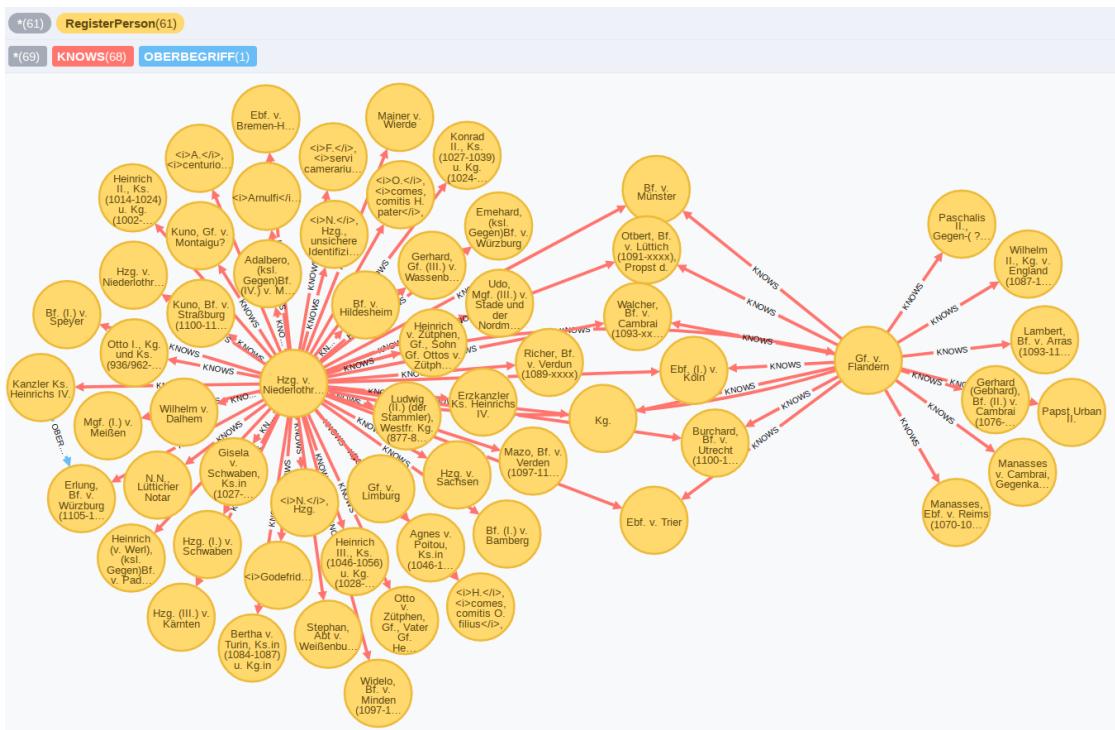


Abbildung 5.17: Robert und Heinrich mit den gemeinsamen Bekanntschaften.

```

AND endPerson.registerId in ['H4P01822']
RETURN DISTINCT startPerson.name1,
regest1.ident, rekest1.text,
middlePerson.name1, rekest2.ident,
rekest2.text, endPerson.name1;

```

In der folgenden Abbildung wird ein Ausschnitt der Ergebnistabelle gezeigt. In der ersten Spalte der Tabelle finden sich Robert, anschließend die Angaben zum Regest, mit dem er mit der mittleren Person (middlePerson.name1) verknüpft ist. Dem folgen schließlich die Angaben zum Regest, mit dem die mittlere Person mit Robert in der letzten Spalte verbunden ist. Die Tabelle bietet einen Überblick zur Überlieferungssituation aus der Perspektive der Regesta Imperii.

startPerson.name1	rekest1.ident	rekest1.text	middlePerson.name1	rekest2.ident	rekest2.text	endPerson.name1
"Heinrich, Hzg. v. Niederlothringen (1101-1106), Pfalzgf. III. v. Lothringen, Gf. v. Limburg"	"RI III,2,3 n. 1489"	"Heinrich bestätigt der bischöflichen Kirche zu <span class="spaced">Bamberg</span> unter Bischof Otto zum Gedenken an seine Großeltern, Kaiser Konrad (II.) und Kaiserin Gisela, seine Eltern, Kaiser Heinrich (III.) und Agnes, seine Gemahlin, die Kaiserin Bertha, sowie besonders an seinen Verwandten, Kaiser Heinrich (II.), den Gründer der Bamberger Kirche, aufgrund der Intervention seines Sohnes, König Heinrichs V., der Erzbischöfe Friedrich von Köln, Bruno von Trier und Humbert von Bremen, der Bischöfe Obert von	"Bruno (v. Lauffen), Ebf. v. Trier (1101- 1124), Trierer Dompropst"	"RI III,2,3 n. 1487"	"Heinrich feiert das Fest der Apostel, wobei sich Graf Robert von Flandern im Beisein mehrerer Fürsten unterwirft, namentlich der Erzbischöfe Friedrich von Köln und Bruno von Trier, der Bischöfe Obert von Lüttich, Burchard von	"Robert II. v. Flandern, Gf. v. Flandern, Sohn Gf. Roberts I. (d. Friesen) v. Flandern, Neffe Gf. Baldwins VI. v. Flandern"

Abbildung 5.18: Robert und Heinrich mit den gemeinsamen Bekanntschaften.

### 5.5.2 Herrscherhandeln ausgezählt

Wie bereits oben erwähnt wurde in einem ersten Test jeweils das erste Verb des Regesttextes extrahiert, lemmatisiert und in die Graphdatenbank eingespielt. Im folgenden werden nun einige cypher-Querys vorgestellt, die dies beispielhaft auswerten.

```

// Herrscherhandeln ausgezählt
MATCH (n:Lemma)-[h:ACTION]-(m:Regesta)
RETURN n.lemma, count(h) as ANZAHL ORDER BY ANZAHL desc LIMIT 10;

```

n.lemma	ANZAHL
werden	145
schenken	133
bestätigen	109

n.lemma	ANZAHL
begehen	95
verleihen	48
ernennen	36
nehmen	35
treffen	34
empfangen	29
erhalten	26

Die Ergebnisliste zeigt gleich die Einschränkungen, da das Hilfsverb *werden* aus dem textuellen Zusammenhang gerissen ist. Andererseits ergeben sich aber auch interessante Erkenntnisse zur Häufigkeitsverteilung von Herrscherhandeln in Regestentexten. Die Anwendung des Verfahrens auf Regestentexte ist dabei auf der einen Seite positiv, da bei der Erstellung der Regesten sehr stark auf formale Kriterien geachtet wird und so die Zusammenhänge gut zu erfassen sind. Auf der anderen Seite ist die Auswertung aber wiederum einen weiteren Schritt von der ursprünglichen Quelle entfernt.

### 5.5.3 Herrscherhandeln pro Ausstellungsort ausgezählt

Im folgenden Query kommt eine räumliche Komponente zur Abfrage hinzu, da das Lemma hier jeweils abhängig vom Ausstellungsort der Urkunde abgefragt wird.

```
// Herrscherhandeln pro Ausstellungsort
MATCH (n:Lemma)<-[h:ACTION]-(:Regesta)-[:PLACE_OF_ISSUE]->(p:Place)
WHERE p.normalizedGerman IS NOT NULL
RETURN p.normalizedGerman, n.lemma, count(h) as ANZAHL ORDER BY ANZAHL desc LIMIT 10;
```

p.normalizedGerman	n.lemma	ANZAHL
Mainz	begehen	15
Mainz	schenken	14
Goslar	schenken	13
Rom	werden	12
Regensburg	schenken	12
Goslar	begehen	11
Speyer	schenken	10
Worms	begehen	8
Regensburg	bestätigen	7
Regensburg	werden	7

In der ersten Spalte befindet sich der Ortsname, der aus der Property `normalizedGerman` des `Place`-Knotens stammt. In der zweiten Spalte wird das Lemma angegeben und in

der dritten Spalte schließlich die Anzahl der jeweiligen Regesten. Interessant wäre hier auch noch die Ergänzung der zeitlichen Dimension, mit der dann der zeitliche Verlauf in die Auswertung miteinbezogen werden könnte.

#### 5.5.4 Herrscherhandeln und Anwesenheit

Im nächsten Beispiel werden in einem Regest genannten Personen in die Auswertung des Herrscherhandelns mit einbezogen.

```
MATCH (p:IndexPerson)-[:PERSON_IN]-(r:Regesta)-[:ACTION]-(l:Lemma)
RETURN p.name1, l.lemma, count(l) AS Anzahl ORDER BY p.name1, Anzahl DESC;
```

<u>p.name1 l.lemma Anzahl</u>
...
Adalberschenken 21
Met-
zer
Dom-
kano-
ni-
ker,
Kanz-
ler
Hein-
richs
IV.,
Kanz-
ler
(Ge-
gen)Kg.
Ru-
dolfs
v.
Rheinfelden

<hr/> p.name\ll.lemmaAnzahl <hr/>	
Adalber <b>bestätigen</b>	9
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	
Adalber <b>verleihen</b>	4
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	

p.name    lemmaAnzahl	
Adalberɔ̄lassen	2
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	
Adalberɔ̄bertragen	2
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	

---

p.name\ll.lemmaAnzahl

---

Adalberɔ̄mäßigen 2

Met-

zer

Dom-

kano-

ni-

ker,

Kanz-

ler

Hein-

richs

IV.,

Kanz-

ler

(Ge-

gen)Kg.

Ru-

dolfs

v.

Rheinfelden

Adalberɔ̄gestatten 2

Met-

zer

Dom-

kano-

ni-

ker,

Kanz-

ler

Hein-

richs

IV.,

Kanz-

ler

(Ge-

gen)Kg.

Ru-

dolfs

v.

Rheinfelden

---

p.name\ll.lemmaAnzahl

---

Adalber~~o~~llziehen 1

Met-

zer

Dom-

kano-

ni-

ker,

Kanz-

ler

Hein-

richs

IV.,

Kanz-

ler

(Ge-

gen)Kg.

Ru-

dolfs

v.

Rheinfelden

Adalber~~o~~nehmen 1

Met-

zer

Dom-

kano-

ni-

ker,

Kanz-

ler

Hein-

richs

IV.,

Kanz-

ler

(Ge-

gen)Kg.

Ru-

dolfs

v.

Rheinfelden

p.name \ll. lemmaAnzahl	
Adalber <b>m</b> indern	1
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	
Adalber <b>s</b> etzen	1
Met-	
zer	
Dom-	
kano-	
ni-	
ker,	
Kanz-	
ler	
Hein-	
richs	
IV.,	
Kanz-	
ler	
(Ge-	
gen)Kg.	
Ru-	
dolfs	
v.	
Rheinfelden	
...	...
...	...

Die Ergebnistabelle zeigt den Abschnitt zu Adalbero, einem Metzer Domkanoniker mit der Häufigkeit des jeweiligen Herrscherhandeln-Lemmas.

### 5.5.5 Regesten 200 km rund um Augsburg

Mit dem folgenden Query werden für den Umkreis von 200 km rund um Augsburg alle Regesten aufgerufen.

```
// Entfernung von Orten berechnen lassen
MATCH (n:Place)
WHERE n.normalizedGerman = 'Augsburg'
WITH n.latLong as point
MATCH (r:Regesta)
WHERE distance(r.latLong, point) < 200000
AND r.placeOfIssue IS NOT NULL
AND r.placeOfIssue <> 'Augsburg'
RETURN r.ident, r.placeOfIssue,
distance(r.latLong, point) AS Entfernung
ORDER BY Entfernung;
```

Solche Queries lassen sich auch mit zeitlichen Abfragen kombinieren und bieten sehr flexible Abfragemöglichkeiten.

### 5.5.6 Welche Literatur wird am meisten zitiert

Beim Import der Regesten in die Graphdatenbank werden die mit dem RI-Opac verlinkten Literaturtitel als eigenständige **Reference**-Knoten angelegt und jeweils mit dem **Regesta**-Knoten verknüpft. Diese Verknüpfung wird mit dem folgenden Query abgefragt, ausgezählt und aufgelistet.

```
MATCH (n:Reference)-[r:REFERENCES]-(m:Regesta)
RETURN n.title, count(r) AS Anzahl
ORDER BY Anzahl DESC LIMIT 10;
```

n.title	ANZAHL
Stumpf	215
Böhmer	201
Ldl	101
Jaffé	60
Schmale	56
Buchholz	51
Scheffer-Boichorst	50
Wauters	39

n.title	ANZAHL
Dobenecker	33
Remling	28

Mit diesen Daten lassen sich Zitationsnetzwerke in den Regesten darstellen mit denen Regesten gefunden werden können, die auf Grund der gemeinsam zitierten Literatur die gleichen inhaltlichen Schwerpunkte aufweisen können.

### 5.5.7 Der Import zusammengefasst

Den komplette [cypher-Code](#) für die Erstellung der Graphdatenbank ist zusammengefasst über ein [Textdatei](#) abrufbar. Es ist zu empfehlen, die aktuelle Version von neo4j-Desktop zu installieren, eine Graphdatenbank anzulegen und in der Graphdatenbank die APOC-Bibliothek zu installieren. Inzwischen ist es möglich, in der Befehlszeile des neo4j-Browsers auch mehrere Befehle nacheinander ausführen zu lassen. Alternativ kann man nach dem Start der Graphdatenbank im Reiter **Terminal** mit dem Befehl `bin/cypher-shell` die cypher-shell aufgerufen werden. In diese Shell werden dann alle Befehl gemeinsam reinkopiert und ausgeführt. Alternativ zur Installation von neo4j kann auch auf den Internetseiten von neo4j seine [Sandbox](#) erstellt werden.

## 5.6 Zusammenfassung

In diesem Kapitel wurden die Schritte zum Import der Regesten Kaiser Heinrichs IV. in die Graphdatenbank neo4j erläutert sowie verschiedene Auswertungsbeispiele vorgestellt.

# 6 Import von strukturierten XML-Daten in neo4j

In diesem Kapitel wird der Import von strukturierten XML-Daten in die Graphdatenbank neo4j beschrieben. Strukturiert meint hierbei, dass es sich nicht um mixed-content handelt, beim dem Text und Auszeichnung gemischt vorliegen können, sondern um Daten in einer datenbank-ähnlichen Struktur. Die Daten stammen aus einem Projekt meines Kollegen Thomas Kollatz, der sie mir freundlicherweise zur Verfügung gestellt hat. Ziel des Kapitels ist es, zunächst die Struktur der XML-Daten im Graphen zu analysieren, dann ein Graphmodell zu entwickeln und anschließend den Import durchzuführen.

## 6.1 Das XML-Beispiel

Das XML-Beispiel enthält eine Liste von Buchwerken (`<work>`) die in einer Sammlung (`<collection>`) zusammengefasst sind. Innerhalb der einzelnen Bücher-Einträge sind neben dem Titel noch Angaben zu Autoren, Kommentatoren und dem Druckort zu finden. In der folgenden Abbildung wird ein Auszug aus den Daten gezeigt.

Das root-Element in den XML-Beispiel ist `<collection>`. Innerhalb von `<collection>` finden sich Angaben zu verschiedenen Büchern, die jeweils wieder mit einem `<work>`-Element zusammengefasst sind. Zu jedem Buch werden folgende Angaben gemacht:

- Titel des Buches im `<title>`-Element
- Autor(en) des Buches um `<autor>`-Element, ggf. durchnummieriert mit Zahlen in eckigen Klammern (z.B. [1])
- Kommentator des Buches im `<kommentator>`-Element
- Druckort des Buches im `<druckort>`-Element

## 6.2 Knotentypen

Für die Modellierung dieser Datenstruktur in der Graphdatenbank müssen zunächst die verschiedenen Entitäten identifiziert werden um festzulegen, welche Knotentypen notwendig sind. Als erstes scheint es sinnvoll einen Knoten vom Typ `Werk` anzulegen, wie es auch im XML über das `<work>`-Element im XML modelliert ist. Die dem `<work>`-Element untergeordneten Elemente `<title>`, `<autor>`, `<kommentator>` und `<druckort>`

```
<collection>
  <work id="1">
    <title>Be'ur millot ha-higgajon</title>
    <autor>[1] Mose ben Maimon</autor>
    <autor>[2] Nieto, David ben Pinchas</autor>
    <kommentator>Mendelssohn, Moses</kommentator>
    <druckort>Berlin</druckort>
  </work>
  <work id="2">
    <title>Be'ur millot ha-higgajon</title>
    <autor>Mose ben Maimon</autor>
    <kommentator>Mendelssohn, Moses</kommentator>
    <druckort>Berlin</druckort>
  </work>
  <work id="3">
    <title>Be'ur millot ha-higgajon</title>
    <autor>Mose ben Maimon</autor>
    <kommentator>Mendelssohn, Moses</kommentator>
    <druckort>Berlin</druckort>
  </work>
```

Abbildung 6.1: Auszug aus dem XML-Beispiel (Quelle: Kuczera)

sind für das Werk jeweils spezifisch. Den Titel eines Werkes können wir in einem **Titel**-Knoten ablegen, den Druckort in einem **Ortsknoten** und Autoren sowie Kommentatoren werden in **Personen**-Knoten gespeichert. Hier ist zu beachten, dass die identifizierten Entitäten, wie z.B. Personen nicht in Knotentypen gespeichert werden, die ihre Rolle wieder geben (wie z.B. Autor oder Kommentator) sondern unabhängig von ihrer Rolle in der allgemein gehaltenen Kategorie Person. Im Graphen werden die verschiedenen Rollen, wie Autor oder Kommentator dann über die Kanten modelliert, was im nächsten Abschnitt näher erläutert wird.

### 6.3 Kantentypen

Nach den Knotentypen sind nun die Kantentypen festzulegen. Sie geben an, in welcher Beziehung die verschiedenen Knoten zueinander stehen. Sieht man sich die XML-Vorlage an, ergeben sich folgende Typen von Kanten:

- **GEDRUCKT\_IN**
- **AUTOR\_VON**
- **KOMMENTIERT\_VON**

Mit der **GEDRUCKT\_IN**-Kante werden ein Werk und ein Ort verbunden und damit angegeben, dass dieses Buch in jenem Ort gedruckt worden ist.

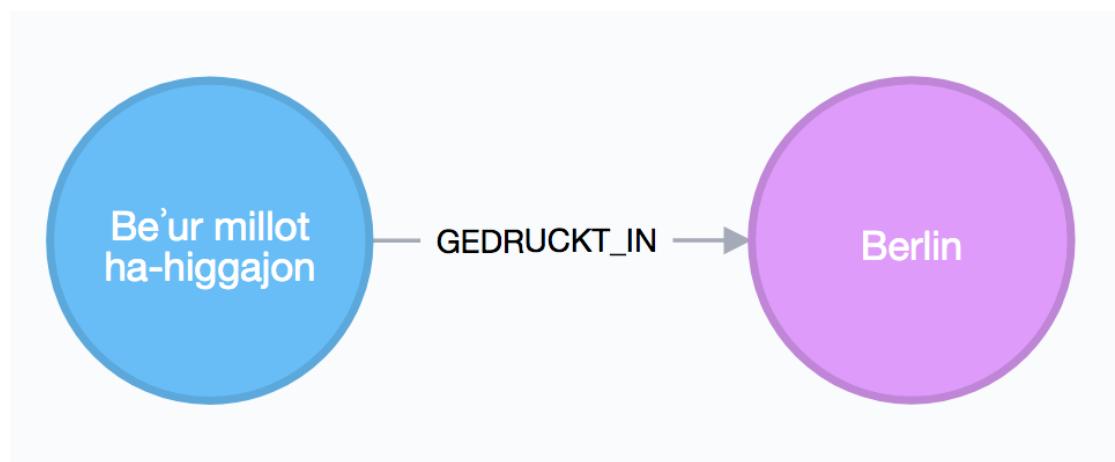


Abbildung 6.2: Verbindung zwischen einem **Werk**- und einem **Ort**-Knoten (Quelle: Kuczera).

Die **AUTOR\_VON**-Kante verbindet einen Personenknoten mit einem Werkknoten und ordnet damit den Autor dem von ihm geschriebenen Buch zu.

Mit der **KOMMENTIERT\_VON**-Kante wird auch ein Personenknoten einem Werkknoten zugeordnet, diesmal nimmt die Person aber die Rolle des Kommentierenden ein.

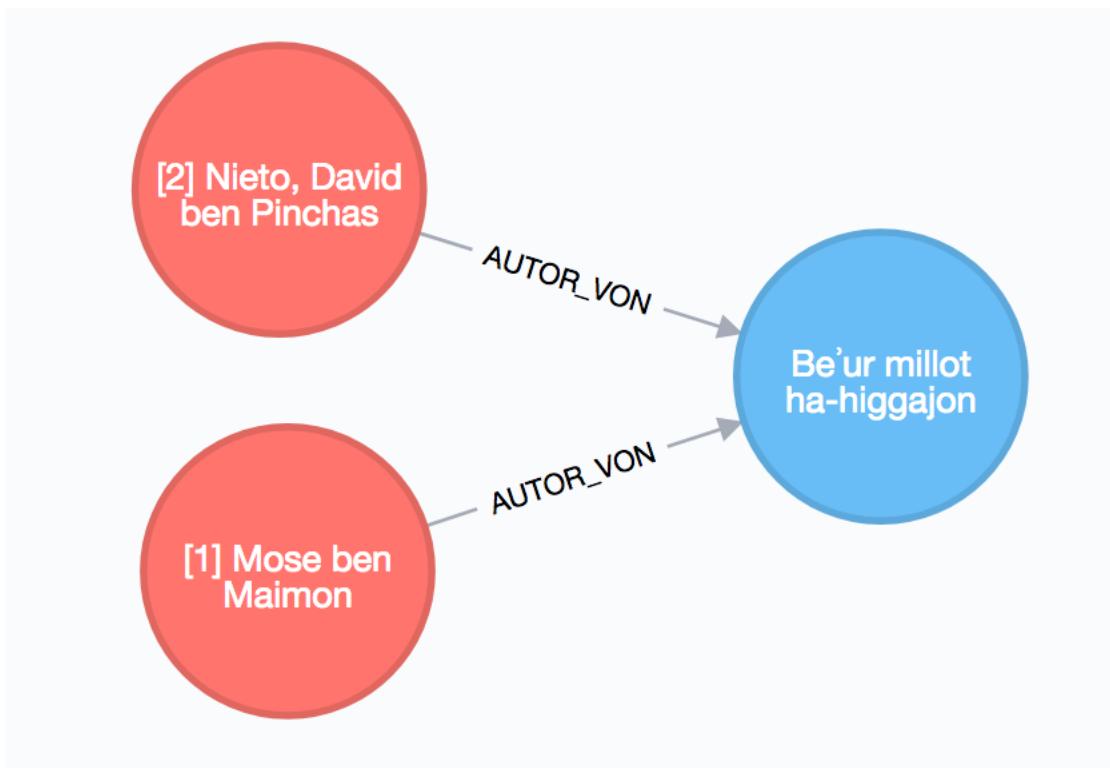


Abbildung 6.3: Verbindung zwischen einem Werk- und einem Ort-Knoten (Quelle: Kuczera).



Abbildung 6.4: Verbindung zwischen einem Werk- und einem Ort-Knoten (Quelle: Kuczera).

Im der folgenden Abbildung werden alle Knoten und Kanten des Beispiels gemeinsam dargestellt.

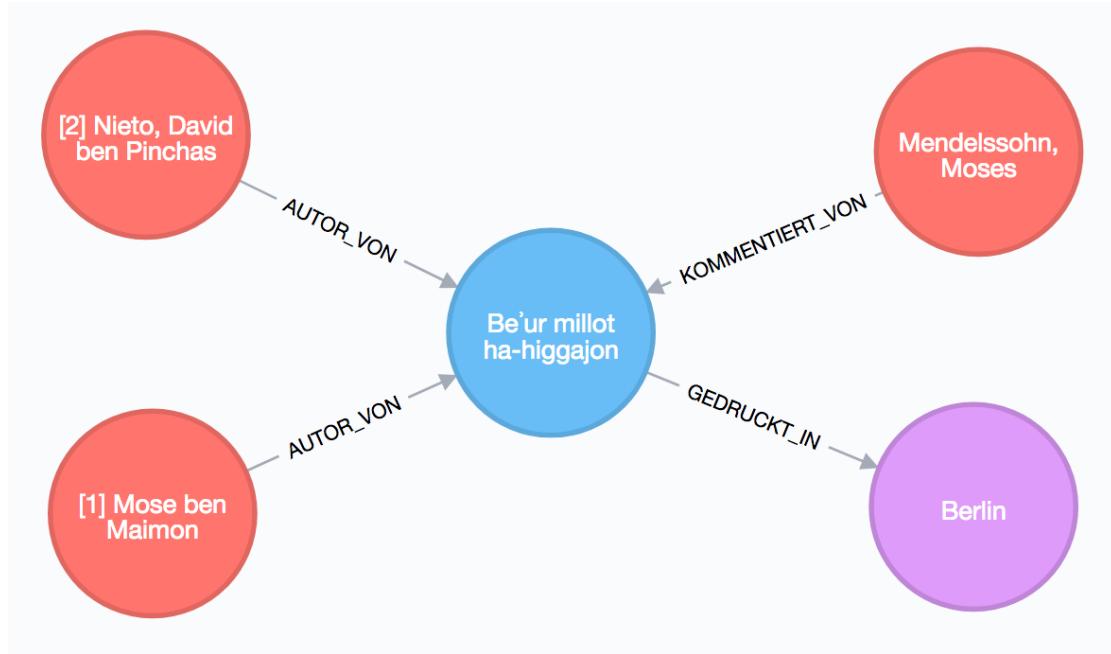


Abbildung 6.5: Verbindung zwischen einem Werk- und einem Ort-Knoten (Quelle: Kuczera).

Damit steht das Graphmodell fest und im nächsten Abschnitt geht es an den Import.

## 6.4 Der Import mit apoc.load.xmlSimple

Für den Import von XML-Daten steht in der apoc-Bibliothek der Befehl apoc.load.xml zur Verfügung. Im folgenden wird zunächst der gesamte Befehl für den Import gelistet.

```
CALL apoc.load.xmlSimple("https://raw.githubusercontent.com/kuczera/Graphentechnologien/master/data/beur_millot_ha_higgajon.xml")
UNWIND xmlFile._work as wdata
    MERGE (w1:Werk{eid:wdata.id})
    set w1.name=wdata._title._text
    FOREACH (name in wdata._autor |
        MERGE (p1:Person {Name:name._text})
        MERGE (p1)-[:AUTOR_VON]->(w1) )
    FOREACH (name in wdata._kommentator |
        MERGE (p1:Person {Name:name._text})
        MERGE (p1)-[:KOMMENTIERT_VON]->(w1))
    FOREACH (druckort in [x in
```

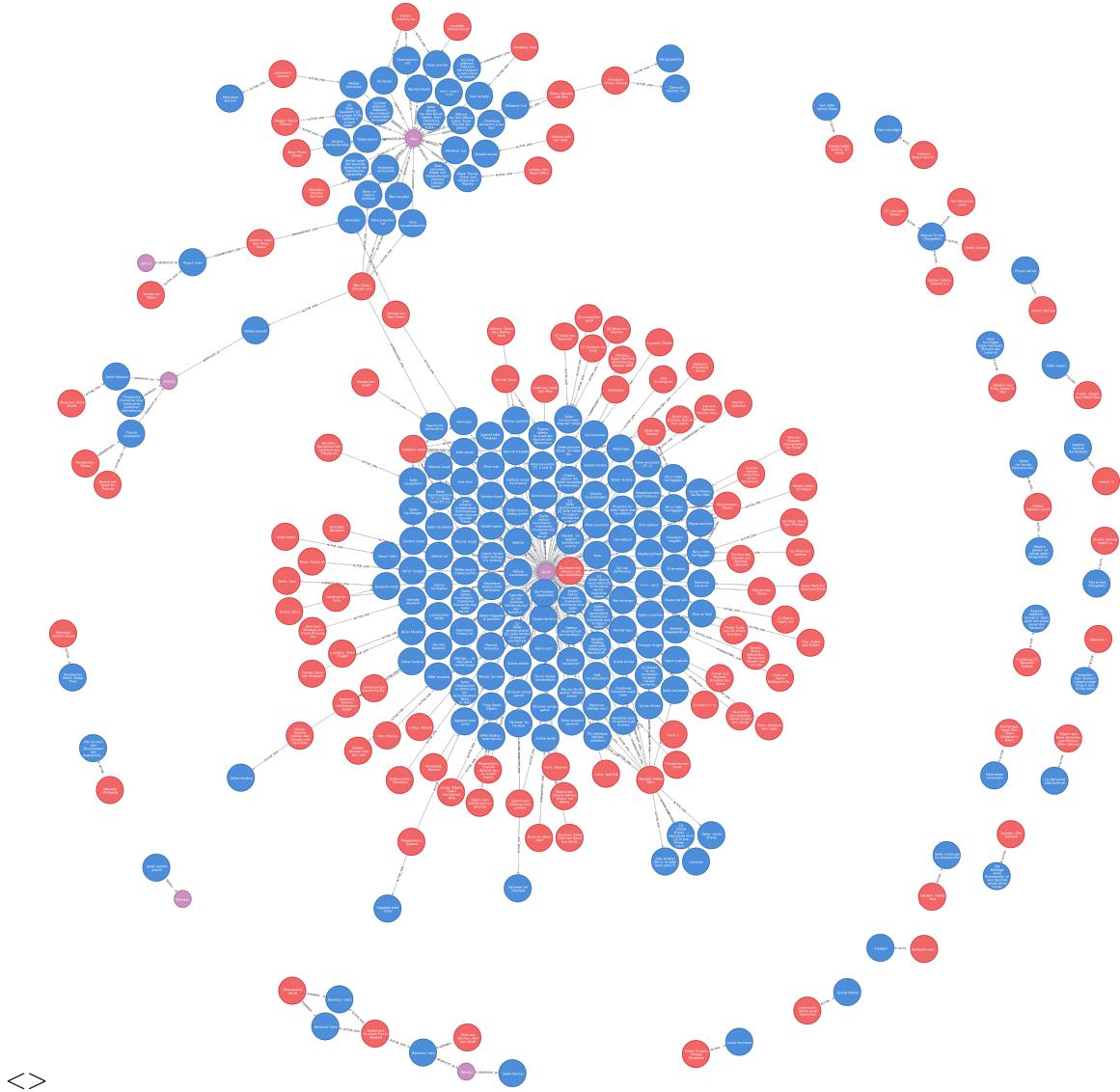
```
wdata._druckort._text where x is not null] |
MERGE (o1:Ort{name:druckort})
MERGE (w1)-[:GEDRUCKT_IN]->(o1);
```

Für den Import wird die apoc-Funktion `apoc.load.xmlSimple` verwendet<sup>1</sup>. Diese Funktion nimmt XML-Dateien oder eine URL und stellt die Daten geparst für die weitere Verarbeitung in einer Map-Struktur zur Verfügung (vgl. die Zeilen 1-4 des Codebeispiels). In der Variable `xmlFile` befindet sich nun diese Map-Struktur. In Zeile 5 folgt der **UNWIND**-Befehl, der jeweils ein Werk (das ist der Inhalt des *work*-Elements in der XML-Datei) an die Variable `value` weitergibt, mit der es dann weiter verarbeitet werden kann. Dies wiederholt sich so lange, bis alle *work*-Elemente der XML-Datei abgearbeitet sind.

Nach dem **UNWIND**-Befehl folgt eine Gruppe von Befehlen, die immer wieder für jedes *work*-Element ausgeführt werden. Als erstes wird mit dem **MERGE**-Befehl ein Knoten vom Typ **Werk** für das Buch mit der Titelangabe in der Eigenschaft `name` erstellt. Dies ist nicht weiter schwierig, da in der XML-Datei für jedes Werk nur ein Titel existiert. Anders ist dies bei den Autoren, von denen einen oder mehrere geben kann, die dann auch in mehreren *autor*-Elementen verzeichnet sind. In der gleichen Weise wird anschließend mit den Angaben zu Autor, Kommentator (die beide Personenknoten ergeben) und mit dem Druckort verfahren. Mit der Erstellung bzw. Prüfung auf Existenz durch den **Merge**-Befehl werden gleichzeitig die **AUTOR\_VON**-, **KOMMENTIERT\_VON**-, und **GEDRUCKT\_IN**-Kanten erstellt und der Graph vervollständigt.

---

<sup>1</sup>Die apoc-Bibliothek muss nach der Installation von neo4j zusätzlich installiert werden. Nähere Informationen finden sich im Anhang im Abschnitt zur Die Apoc-Bibliothek.



<>

Die Funktion `apoc.loadxmlSimple` ist inzwischen veraltet und wird von der Funktion `apoc.loadxml` abgelöst. Diese ist allgemeiner aber dadurch in der Anwendung etwas komplizierter.

CALL

```
apoc.load.xml("https://raw.githubusercontent.com/kuczera/Graphentechnologien/master/docs",
WITH
[x in work._children where x._type="title" | x._text][0] as titel,
[x in work._children where x._type="autor" | x._text] as autoren,
[x in work._children where x._type="kommentator" | x._text] as kommentatoren,
[x in work._children where x._type="druckort" | x._text] as druckorte,
work.id as eid
```

```

MERGE (w:Werk{eid:eid})
SET w.name = titel
FOREACH (x in autoren |
  MERGE (p:Person {name:x})
  MERGE (p)-[:AUTOR_VON]->(w) )
FOREACH (x in kommentatoren |
  MERGE (p:Person {name:x})
  MERGE (w)-[:KOMMENTIERT_VON]->(p) )
FOREACH (x in druckorte |
  MERGE (o:Ort {name:x})
  MERGE (w)-[:GEDRUCKT_IN]->(o) );

```

## 6.5 Zusammenfassung

In diesem Abschnitt wurde die Analyse einer XML-Datei, die daraus resultierende Graphmodellierung und der Import des XMLs in die Graphdatenbank neo4j beschrieben. Für den Import wurden die Funktionen `apoc.load.xmlSimple` und `apoc.load.xml` aus der apoc-Bibliothek verwendet. Der cypher-Code kann als Grundlage für weitere Importe von XML in die Graphdatenbank neo4j dienen.

# 7 Verwandtschaft im Graphen

In diesem Kapitel wird am Beispiel eines Ausschnitts der Daten des Projekts **Nomen et Gens**<sup>1</sup> die Modellierung von Verwandtschaft in der Graphdatenbank neo4j dargestellt.<sup>2</sup>

## 7.1 Das Projekt Nomen et Gens

Das Projekt **Nomen et Gens** (NeG) zielt darauf ab, alle schriftlich belegten Namen und Personen Kontinentaleuropas in den vier Jahrhunderten vor Karl dem Großen (also von 400 bis 800 nach Christus) zu erfassen. Die Datenbank des Projekts geht auf ein erfolgreich abgeschlossenes DFG-Projekt zurück und wird aktuell von den Projektbeteiligten weiter betreut und sukzessive ausgebaut. Neben den Quellen der Personennennung, den unterschiedlichen Namensformen usw. werden auch die Verwandtschaftsbeziehungen zwischen identifizierten Personen in der Datenbank abgelegt. Dabei werden bis zu 16 verschiedene Verwandtschaftsbeziehungen in der Datenbank gespeichert, wie z. B. Bruder, Schwester, Sohn, Tochter, Vater, Mutter, Onkel oder Nichte. Bei einem Personendatensatz werden dann jeweils die Verwandtschaftsbeziehungen aufgelistet, so dass man sich ggf. jeweils von Person zu Person durchklicken muss, bis man am Ziel angelangt ist.

## 7.2 Nomen et Gens im Graphen

Vor diesem Hintergrund bot es sich an, die Personen und die zugehörigen Verwandtschaftsbeziehungen in die Graphdatenbank neo4j zu transferieren und anschließend graphbasierte Abfrageperspektiven zu testen.

Die Abbildung zeigt die ersten Ergebnisse des Datenbankimports. Aus der Visualisierung werden die zahlreichen redundanten Beziehungen deutlich, die in der Folge zu neuen Modellierungsansätzen für die Verwandtschaftsbeziehungen führten. Ergebnis der Überlegungen war die Reduzierung der möglichen Verwandtschaftsbeziehungen auf die zwei Kantentypen KIND und VERHEIRATET\_MIT. Dabei wird eine Kante vom Typ KIND

<sup>1</sup>Informationen zum Projekt „Nomen et Gens“ finden Sie unter <http://www.neg.uni-tuebingen.de/> (abgerufen am 10.08.2018).

<sup>2</sup>Dieses Kapitel geht in großen Teilen zurück auf meinem Aufsatz Graphentechnologien in den Digitalen Geisteswissenschaften, in: ABI Technik 2017; 37(3): 179–196, <https://doi.org/10.1515/abitech-2017-0042>. URL: <https://www.degruyter.com/downloadpdf/j/abitech.2017.37.issue-3/abitech-2017-0042/abitech-2017-0042.pdf>, insbesondere die Seiten 179 bis 182 und wurde nur geringfügig ergänzt.

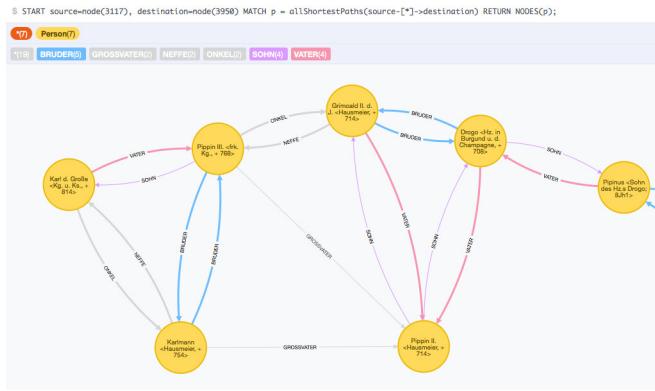


Abbildung 7.1: Erste Importergebnisse

für eine Eltern-Kind-Beziehung nur einmal vergeben, während eine Kante vom Typ VERHEIRATET\_MIT immer zweifach in jeweils umgekehrter Richtung angelegt wird. Dies ergibt sich aus der Überlegung, dass eine Eltern-Kind-Beziehung gerichtet ist und zwar in unserem Fall vom Elternteil zum Kind hin, während eine VERHEIRATET\_MIT-Beziehung ungerichtet ist: Wenn eine Person mit einer anderen Person verheiratet ist, ist die andere Personen automatisch auch mit der ersten verheiratet. Da im Property-Graph-Modell von neo4j jede Kante genau eine Richtung haben muss, wird die VERHEIRATET\_MIT-Kante zweimal in jeweils unterschiedliche Richtung angelegt, während bei der hierarchischen Eltern-Kind-Beziehung eine Kante ausreicht.

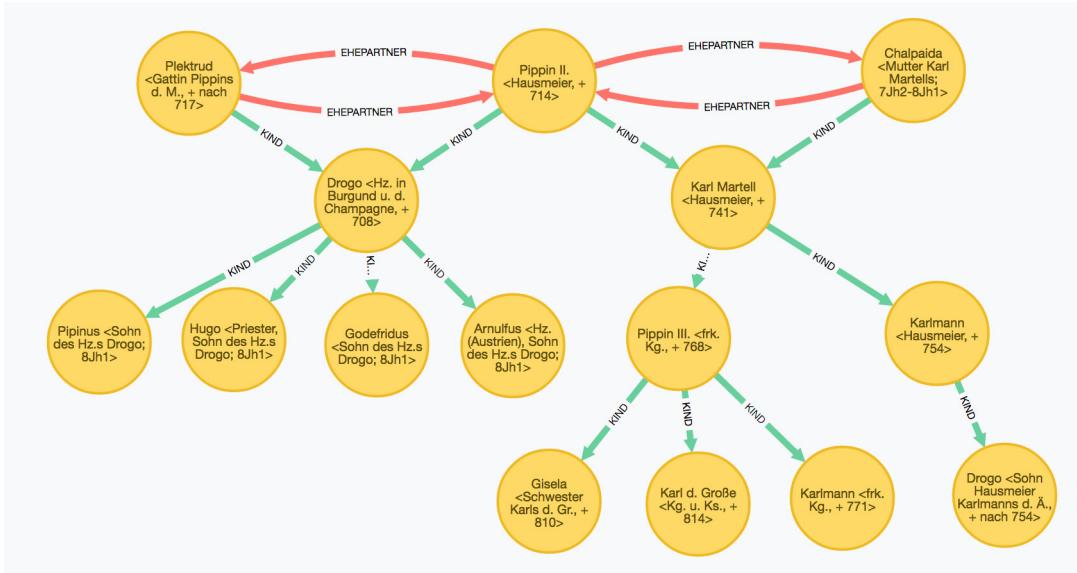


Abbildung 7.2: Die Urenkel Pippins

## 7.3 Sind Berchar und Karl der Große verwandt ?

Im folgenden Beispiel soll das Potential der Graphmodellierung von Verwandtschaftsbeziehungen demonstriert werden. In der Datenbank gibt es die Person Berchar. Berchar war ein Hausmeier König Theuderichs III. Die Frage ist nun, ob dieser Berchar mit Karl dem Großen verwandt ist. In der NeG-Datenbank ist ein Verwandtschaftsverhältnis von Berchar zu Karl dem Großen nicht direkt ableitbar.

Prosopographisches			ID: P7119
Person	Berchar <Hausmeier Kg. Theuderichs III., + um 688/90>		
andere Namen	Berthar		
Geschlecht	m		
Kommentar	Berchar folgte um 686 seinem Schwiegervater Waratto als neutrischer Hausmeier nach. Er war im Konflikt mit Pipin II. d. Mittleren und wurde im Zuge dieser Auseinandersetzungen um 690 (oder 688) erschlagen (Berchar, LMA 1, 1980, 1931).		
Personenkommentar			
Stand	vir inluster		
Ämter	Amt	Zeitraum	
	maior domus	686-690	
	consiliarius	687	
Ethnie	Ethnie	Typus der Zuschreibung	
Verwandte	Name d. Person	Verwandtschaftsgrade	
	Drogo	Schwiegersohn	
	Waratto	Schwiegervater	
	Anstrud	Ehepartner	
	Adaltrud	Tochter	
	Ansflid	Schwiegermutter	

Abbildung 7.3: Berchar in der Nomen-et-Gens-Datenbank

In der Graphdatenbank neo4j wird für eine solche Fragestellung eine Shortest\_Path-Abfrage verwendet, die den kürzesten möglichen Weg zwischen zwei Knoten zurück liefert, sofern es einen gibt. Der folgende cypher-Befehl liefert den Pfad zwischen dem Personenknoten Karls des Großen mit der NeG-ID 7404 und dem Personenknoten von Berchar mit der NeG-ID 7119. Dabei wird die Länge des abzufragenden Pfades auf 15 Kanten begrenzt.

```
// shortest_path-Abfrage von Karl dem Großen zu Berchar
MATCH (KdG:Person { nid:'7404' })
MATCH (Berchar:Person { nid:'7119' })
p = shortestPath((KdG)-[*..15]-(Berchar))
RETURN p;
```

Das Ergebnis zeigt, dass Berchar tatsächlich mit Karl dem Großen verwandt ist. Er ist nämlich der Schwiegervater von Drogo (Herzog in Burgund und der Champagne, gest. 708), der wiederum der Bruder des Großvaters Karls des Großen ist.

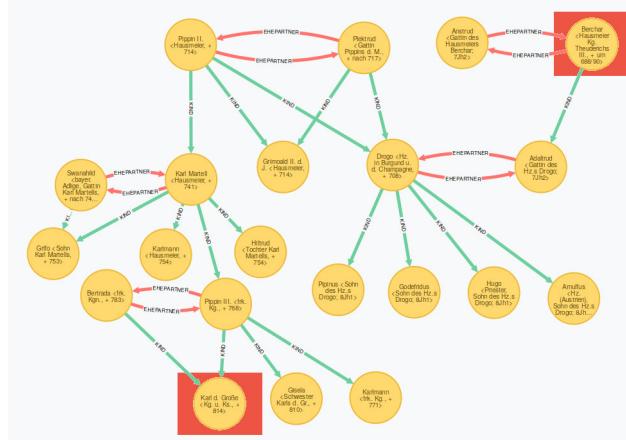


Abbildung 7.4: Der kürzeste Pfad (shortestPath) von Karl zu Berchar.

## 7.4 Zusammenfassung

Mit diesem Beispiel sind die interessanten Erschließungs- und Modellierungsperspektiven für die digitale Genealogie nur angedeutet. Mit Graphentechnologien lässt sich intuitive Datenmodellierung mit sehr flexiblen Erschließungs- und Abfragemöglichkeiten kombinieren.

# 8 Das DTA im Graphen

In diesem Abschnitt werden zwei Verfahren zum Import von XML-Texten des Deutschen Textarchivs (DTA) in die Graphdatenbank neo4j vorgestellt. Dabei unterscheidet sich sowohl das gewählte Ausgangsformat der DTA als auch die gewählten Importtechniken.

## 8.1 Das deutsche Textarchiv

Das Deutsche Textarchiv (DTA) stellt einen Disziplinen übergreifenden Grundbestand deutscher Werke aus dem Zeitraum von ca. 1600 bis 1900 im Volltext und als digitale Faksimiles frei zur Verfügung und bereitet ihn so auf, dass er über das Internet in vielfältiger Weise nutzbar ist. Das DTA-Korpus soll in größtmöglicher Breite widerspiegeln, was an bedeutenden Werken in deutscher Sprache veröffentlicht wurde. Die ausgewählten Texte stehen repräsentativ für die Entwicklung der deutschen Sprache seit der Frühen Neuzeit. Alle DTA-Texte werden unter einer offenen Lizenz veröffentlicht (CC BY-NC). Das DTA fördert die Wiederverwendung seiner Texte in allen Bereichen der Digitalen Geisteswissenschaften.

## 8.2 Die Downloadformate des DTA

Das DTA bietet zu den bereitgestellten Texten verschiedene Formate zum Download an. Als Beispiel wird hier [Goethes Faust](#) in der ersten Auflage von 1808 importiert.

- [TEI-P5](#) bietet die textkritische Fassung des Faust
- [TCF](#) bietet die tokenisierte, serialisierte, lemmatisierte und normalisierte Fassung, textkritische Informationen fehlen jedoch.
- [Plain-Text](#) bietet einen einfachen Text mit Seiten- und Zeilenfall ohne weitere Zusatzinformationen

Für den Import in eine Graphdatenbanken bietet sich das TCF-Format an, da es den Text tokenisiert, serialisiert, lemmatisiert und normalisiert bietet. In diesem Format lässt er sich mit cypher-Befehlen in die Graphdatenbank importieren. Im Beispiel wird Goethes Faust in der TCF-Fassung in die Graphdatenbank importiert.

Hier wird ein Ausschnitt aus der TCF-Datei<sup>1</sup> gezeigt

---

<sup>1</sup>Vgl. zu diesem Beispiel [http://deutschestextarchiv.de/book/view/goethe\\_faust01\\_1808?p=11](http://deutschestextarchiv.de/book/view/goethe_faust01_1808?p=11).

```

<token ID="w5b">Ihr</token>
<token ID="w5c">bringt</token>
<token ID="w5d">mit</token>
<token ID="w5e">euch</token>
<token ID="w5f">die</token>
<token ID="w60">Bilder</token>
<token ID="w61">froher</token>
<token ID="w62">Tage</token>
<token ID="w63">,</token>
<token ID="w64">Und</token>
<token ID="w65">manche</token>
<token ID="w66">liebe</token>
<token ID="w67">Schatten</token>

```

und im Anschluss im Vergleich das Original (links) und der Lesetext (rechts).

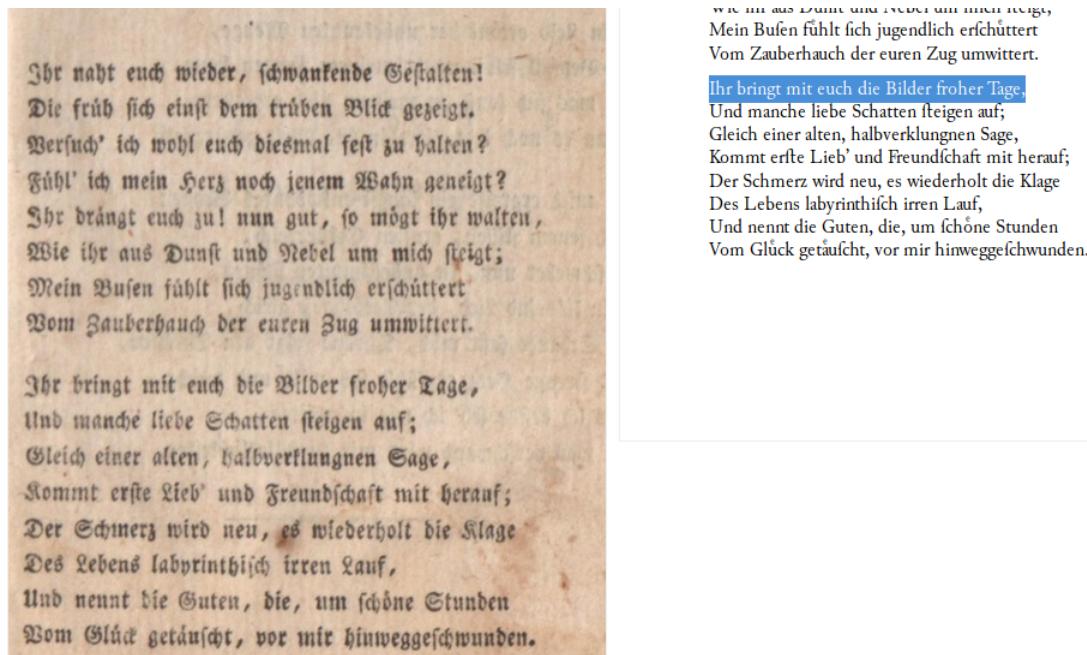


Abbildung 8.1: Eine Beispielzeile aus dem Faust

Vergleicht man das TCF-Xml mit der gleiche Stelle im TEIP5 ist zu erkennen, dass in letzterem der Zeilenfall annotiert ist.

```

<lb/>
<lg n="2">
  <1>Ihr bringt mit euch die Bilder froher Tage,</1><lb/>
  <1>Und manche liebe Schatten &#x017F;teigen auf;</1><lb/>

```

Die Downloadformate sind also für verschiedene Nutzungsszenarien optimiert. Für den

Import in eine Graphdatenbank bietet sich das TCF-Format an.

## 8.3 Vorbereitungen

Als Vorbereitung müssen einige Constraints eingerichtet werden.<sup>2</sup>

```
create constraint on (t:Token) assert t.id is unique;
create constraint on (s:Sentence) assert s.id is unique;
create constraint on (l:Lemma) assert l.text is unique;
```

Mit den Befehlen wird sichergestellt, dass die im nächsten Schritt importierten Knoten eindeutige IDs haben.

## 8.4 Import des TCF-Formats

### 8.4.1 Tokenimport

Nun folgt der Import-Befehl mit der apoc-procedure *apoc.load.xmlSimple*.

```
call apoc.load.xml('http://deutschestextarchiv.de/book/download_fulltcf/16181') yield val
unwind doc._TextCorpus._tokens._token as token
create (t:Token{id:token.ID, text:token._text})
with collect(t) as tokens
unwind apoc.coll.pairs(tokens)[0..-1] as value
with value[0] as a, value[1] as b
create (a)-[:NEXT_TOKEN]->(b);
```

In der ersten Zeile wird der apoc-Befehl *apoc.load.xmlSimple* aufgerufen, der als Argument die URL der TCF-Version von Goethes Faust im Deutschen Textarchiv erhält. Die weiteren cypher-Befehle parsen die XML-Datei und spielen die Token (also die einzelnen Wörter) als Wortknoten in die Graphdatenbank ein. Schließlich werden die NEXT\_TOKEN-Kanten zwischen den eingespielten Wörtern erstellt.

### 8.4.2 Satzstrukturen

Der nächste Befehl lädt wieder die gleiche XML-Datei und importiert die Satzstrukturen.

```
call apoc.load.xmlSimple("http://deutschestextarchiv.de/book/download_fulltcf/16181") yield val
unwind doc._TextCorpus._sentences._sentence as sentence
match (t1:Token{id:head(split(sentence.tokenIDs, " "))})
match (t2:Token{id:last(split(sentence.tokenIDs, " "))})
```

---

<sup>2</sup>Zu constraints vgl. <https://neo4j.com/docs/developer-manual/current/cypher/schema/constraints/>

```

create (s:Sentence{id:sentence.ID})
create (s)-[:SENTENCE_STARTS]->(t1)
create (s)-[:SENTENCE_ENDS]->(t2)
with collect(s) as sentences
unwind apoc.coll.pairs(sentences)[0..-1] as value
with value[0] as a, value[1] as b
create (a)-[:NEXT_SENTENCE]->(b);

```

### 8.4.3 Lemmimport

Im folgenden Befehl werden die Lemmata importiert und jedes Token mit dem zugehörigen Lemma verknüpft.

```

call apoc.load.xmlSimple('http://deutschesestextarchiv.de/book/download_fulltcf/16181') yie
unwind doc._TextCorpus._lemmas._lemma as lemma
match (t:Token{id:lemma.tokenIDs})
merge (l:Lemma{text:lemma._text})
create (t)-[:LEMMATISIERT]->(l);

```

Der letzte Befehl ergänzt bei jedem Token-Knoten noch die Lemma-Information als Proptery.

```

call apoc.load.xmlSimple('http://deutschesestextarchiv.de/book/download_fulltcf/16181') yie
unwind doc._TextCorpus._lemmas._lemma as lemma
match (t:Token{id:lemma.tokenIDs}) set t.Lemma = lemma._text;

```

Damit ist nun die Fassung von Goethes Faust aus dem Deutschen Textarchiv in die Graphdatenbank importiert worden und kann weiter untersucht werden (hier klicken, um den Code mit den cypher-Querys für den gesamten Artikel herunterzuladen).

### 8.4.4 Beispielabfrage

Bei Cypher-Abfragen können alle Eigenschaften von Knoten und Kanten miteinbezogen werden. Der Query fragt nach einem Token-Knoten mit dem Lemma **Bild**, gefolgt von einem Token-Knoten mit dem Lemma **froh** und dazu die drei vorhergehenden und die drei nachfolgenden Token-Knoten.

```

MATCH
w=()-[:NEXT_TOKEN*5]->(a:Token{Lemma:'Bild'})
-[:NEXT_TOKEN]->(b:Token{Lemma:'froh'})
-[:NEXT_TOKEN*5]->()
RETURN *;

```

Damit finden wir die am Anfang des Kapitels vorgestellte Stelle im Graphen

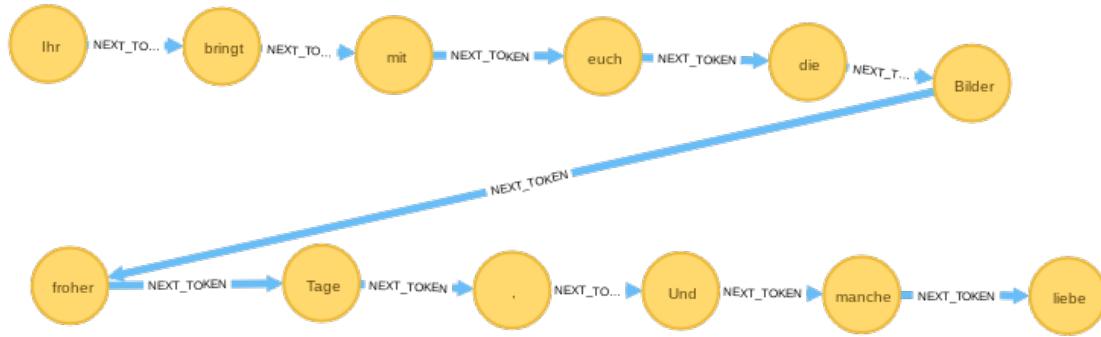


Abbildung 8.2: Eine Beispielzeile aus dem Faust

## 8.5 Import der TEIP5-Fassung

Im nächsten Schritt wird die TEIP5-Fassung von Goethes Faust importiert

```
call
apoc.xml.import("http://deutsches-textarchiv.de/book/download_xml/goethe_faust01_1808", {cr
true})
yield node
return node;
```

und mit dem folgenden Query auch jene **Bilder froher Tage**-Stelle im Text aufgerufen.

```
MATCH
w=(:XmlWord)-[:NEXT_WORD*3]->
(a:XmlWord {text:'Bilder'})-[:NEXT_WORD]->
(:XmlWord {text:'froher'})-[:NEXT_WORD*3]->(:XmlWord)
RETURN *;
```

Das Ergebnis zeigt die komplexere Struktur der gleichen Stelle im TEIP5-Graphen, da hier u.a. auch der Zeilenfall annotiert ist.

## 8.6 Zusammenfassung

Im vorliegenden Kapitel wurden die Schritte für den Import der DTA-TCF-Fassung von Goethes Faust in die Graphdatenbank neo4j vorgestellt. Die qualitativ hochwertigen Text-Quellen des Deutschen Textarchivs bieten in Verbindung mit Graphdatenbanken sehr interessante neue Möglichkeiten zur Auswertung der Texte. Durch Austausch des Links zur TCF-Fassung können auch andere Texte des DTA eingespielt werden. Am Ende wurde beispielhaft die TEI-P5-Fassung eingespielt um die gleiche Stelle in beiden Fassungen vergleichen zu können. Weitere Informationen zu XML im Graphen finden Sie im Kapitel zu [XML-Text im Graphen](#).

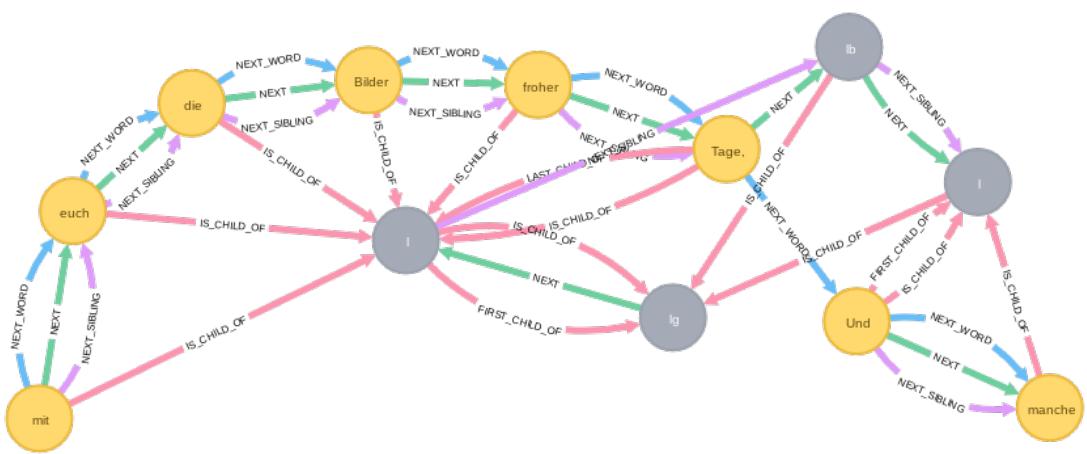


Abbildung 8.3: Die Beispielzeile aus der TEIP5-Fassung des Faust

# 9 Graph-Refactoring mit DTA-XML

## 9.1 Modellierungsüberlegungen

Die Diskussion über Modellierungsansätze von Text als Graph hält aktuell an.<sup>1</sup> Momentan ist XML als Technik für die Codierung von Text in digitalen Editionen sehr verbreitet und bildet einen Quasi-Standard. Da die technische Grundlage von XML normale Plain-Textdateien sind, handelt es sich bei XML um eine eindimensionale Kette von Tokens<sup>2</sup>. Prinzipiell können XML-Dateien ohne größere Probleme in einen Graphen importiert werden, da sie einen geerdeten, gerichteten azyklischen Graphen (der vielfache Elternbeziehungen verhindert) und damit ein Ordered Hierarchy of Content Objects (OHCO) darstellen. Es gibt vor allem im Bereich des Mixed-Content verschiedene Ansätze, XML-Strukturen im Graphen abzubilden<sup>3</sup>. Überlegungen zur Auslagerung von Annotationen aus XML in eine Graphdatenbank brachte schon Desmond Schmidt in die Diskussion ein:

*Embedded annotations can also be removed from TEI texts. The elements <note>, <interp>, and <interpGrp> describe content that, like metadata, is about the text, not the text itself. These are really annotations, and should ideally be represented via the established standards and practices of external annotation (Hunter and Gerber 2012). Annotations are stored in triple stores or graph databases like Neo4J,<sup>20</sup> which record the identifiers of each component of the annotation and its data<sup>4</sup>.*

Diesen Gedanken folgend werden in diesem Abschnitt Ansätze für eine Modellierung der Semantik des DTA-Basisformat im Property-Graphen nach dem Prinzip des **Texts als Kette von Wortknoten** (Text as a chain of wordnodes) mit angelagerter Annotation vorgestellt.

## 9.2 Granularität des Modells – Was ist ein Token ?

Dabei ist die Entscheidung, für ein Wort jeweils einen Knoten zu nehmen schon eine wichtige Vorentscheidung. Es wäre durchaus denkbar auch für jedes Zeichen einen Knoten

<sup>1</sup>Vgl. zuletzt @DekkerHaentjensItmorejust2017.

<sup>2</sup>Beispielsweise sieht @HuitfeldtMarkupTechnologyTextual2014, S. 161, digitale Dokumente prinzipiell als lineare Sequenz von Zeichen.

<sup>3</sup>Vgl. @DekkerHaentjensItmorejust2017.

<sup>4</sup>@SchmidtInteroperableDigitalScholarly2014, 4.1 Annotations.

anzulegen. Für den Bereich der historisch-kritischen und philologischen Editionen ist es in der Regel ausreichend, beim Import von XML-kodierten Texten in den Graphen jeweils ein Wort in einen Knoten zu importieren, da meist die historische Aussage der Quelle im Vordergrund steht. In anderen Bereichen der digitalen Geisteswissenschaften kann die Entscheidung, welche Einheit für den Import in einen Knoten gewählt wird, durchaus anders ausfallen. So ist für Philologien die Betrachtung auf Buchstabenebene interessant<sup>5</sup>.

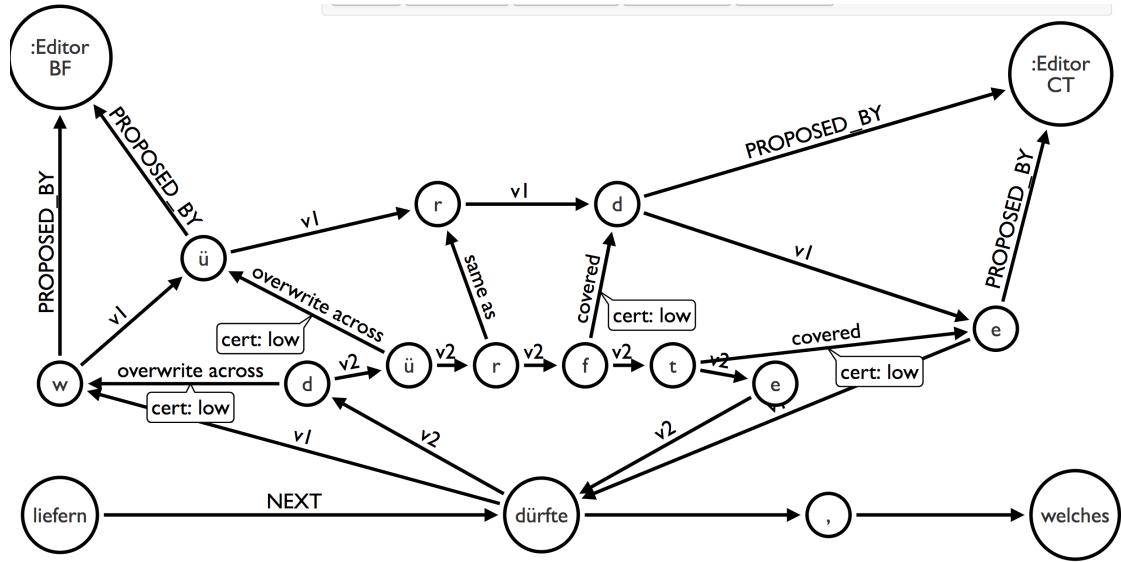


Abbildung 9.1: Granularität von Text im Graphen

Im Graphmodell ist man im Hinblick auf die Granularität des Datenmodells wesentlich flexibler als z.B. bei XML oder Standoff-Markup. So ist es beispielsweise denkbar, an einen Wortknoten eine weitere Kette von Knoten anzulagern, welche pro Knoten jeweils einen Buchstaben des Wortes und die zugehörigen Annotationen enthalten (vgl. Abb. [@fig:Granularitaet-im-Graphen.png]). Es handelt sich um einen buchstabenbasierten Sub-Graphen, dessen Anfang und Ende mit dem Wortknoten verbunden ist. Damit können verschiedene Granularitätsstufen in einem Modell und in einer Datenbank abgebildet werden.

<sup>5</sup>In FuD (<http://fud.uni-trier.de/>) werden Texte in Standoff-Markup auf Buchstabenebene ausgewertet, während beim DTA-Basisformat der Fokus auf der wortbasierten Auszeichnung liegt (vgl. <http://www.deutsches-textarchiv.de/doku/basisformat/eeAllg.html>).

### 9.3 Import der DTA-XML-Daten

Für den Import der Texte wird die Procedure apoc.xml.import aus der apoc-Bibliothek von neo4j verwendet<sup>6</sup>. Die Procedure nimmt XML-Dateien entgegen und importiert sie in die Graphdatenbank.

Importiert wird die Transkription von [Gotthilf Patzigs Mitschriften](#) von Humboldts Vorträgen über physische Geographie.<sup>7</sup>

Mit dem folgenden Befehl wird die Patzig-Mitschrift in die Graphdatenbank importiert<sup>8</sup>:

```
CALL apoc.xml.import('http://www.deutsches-textarchiv.de/book/download_xml/patzig_msgermf...  
  {createNextWordRelationships:true}  
  yield node return node;
```

Dabei werden die XML-Knoten in Graphknoten umgewandelt und verschiedene Arten von Kanten erstellt, die einerseits die Baum-Hierarchie des XMLs im Graphen abbilden. Mit der Option `createNextWordRelationships:true` wird darüber hinaus festgelegt, dass die im XML vorhandenen Textknoten über `NEXT_WORD`-Kanten miteinander verknüpft werden. Zu beachten ist hierbei, dass es in TEI-XML zwei verschiedene Klassen von Elementen gibt. Die eine dient der Klassifizierung von Text, die zweite bringt Varianten und zusätzlichen Text mit, der beim Import in seiner Serialität eingelesen und mit `NEXT_WORD`-Kanten verbunden wird. Dies kann dann zu Problemen bei der Lesbarkeit der Wortkette führen.

Das Wurzelement der importierten XML-Datei wird in einen Knoten vom Typ  `XmlDocument` importiert. Dieser erhält die Properties `_xmlEncoding` zur Darstellung des Encodings, `_xmlVersion` für die XML-Version und `url` für die URL des importierten XML-Dokuments.

Mit einem weiteren cypher-Query erhalten alle der importierten Knoten die Eigenschaft `url` mit der URL des importierten XML-Dokuments. Damit lassen sich Knoten in einer Graphdatenbank mit mehreren importierten XML-Dokumenten auseinanderhalten.

```
MATCH (d: XmlDocument)-[:NEXT_WORD*]->(w: XmlWord)  
SET w.url = d.url;
```

---

<sup>6</sup>Wie die apoc-Bibliothek installiert und die Funktionen und Procedures verwendet werden können wird im [Kapitel für Fortgeschrittene](#) erklärt.

<sup>7</sup>Gotthilf Patzig: Vorträge über physische Geographie des Freiherrn Alexander von Humboldt: gehalten im großen Hörsaal des Universitäts-Gebäudes zu Berlin im Wintersemester 1827/28 vom 3ten Novbr. 1827. bis 26 April 1828. Aus schriftlichen Notizen nach jedem Vortrage zusammengestellt vom Rechnungsrath Gotthilf Friedrich Patzig. Berlin 1827/28 (= Nachschrift der ‚Kosmos-Vorträge‘ Alexander von Humboldts in der Berliner Universität, 3.11.1827–26.4.1828), S. 9. In: Deutsches Textarchiv. Grundlage für ein Referenzkorporus der neuhochdeutschen Sprache. Herausgegeben von der Berlin-Brandenburgischen Akademie der Wissenschaften, Berlin 2007–2019. [http://www.deutsches-textarchiv.de/patzig\\_msgermfol841842\\_1828/13](http://www.deutsches-textarchiv.de/patzig_msgermfol841842_1828/13).

<sup>8</sup>Für die Vereinheitlichung des Druckbildes mussten an einigen Stellen Zeilenumbrüche in die Codebeispiele eingefügt werden, die deren direkte Ausführung behindern.

Mit dem nächsten cypher-Query werden die Knoten des importierten XML-Dokuments durchnummertiert und der jeweilige Wert in der Property `DtaID` abgelegt.

```

MATCH p = (start:XmlDocument)-[:NEXT*]->(end:XmlTag)
WHERE NOT (end)-[:NEXT]->()
AND start.url = 'http://www.deutschesetextarchiv.de/book/down'
WITH nodes(p) as nodes, range(0, size(nodes(p))) AS indexes
UNWIND indexes AS index
SET (nodes[index]).DtaID = index;

```

## 9.4 Erläuterung der entstandenen Graphstrukturen

Nach Abschluss des Imports werden jetzt die importierten Datenstrukturen erläutert. In der folgenden Tabelle werden die verschiedenen Typen von Knoten erläutert, die während des Imports erstellt wurden.

Tabelle zum Importvorgang der XML-Elemente und den entsprechenden Knoten

XML-Knoten	Graphknoten	Bemerkungen
XML-Wurzelement	XmlDocument	Gibt es nur einmal. Es enthält Angaben zur Encodierung, zur XML-Version und die URL der importierten XML-Datei
XML-Element-Knoten	XmlTag-Knoten	Die Attribute des XML-Elements werden in entsprechende Properties des XMLTag-Knotens in der Datenbank umgewandelt
XML-Text-Knoten	XmlWord	Jedes Wort des XML-Textknotens wird ein XmlWord-Knoten im Graphen

In der nächsten Tabelle werden die verschiedenen Kantentypen erläutert. Sie geben die Serialität des XMLs (`NEXT`-Kanten), die Hierarchie (`NEXT_SIBLING` und `IS_CHILD_OF`-Kanten) und auch die Abfolge der Inhalte der XML-Textelemente (`NEXT_WORD`) wieder.

Tabelle zu den erstellen Kantentypen

Kante	Bemerkungen
<code>:NEXT</code>	Zeigt die Serialität der XML-Datei im Graphen
<code>:NEXT_SIBLING</code>	Zeigt auf den nächsten Graphknoten auf der gleichen XML-Hierarchie-Stufe
<code>:NEXT_WORD</code>	Zeigt auf das nächste Wort in einem XML-Textknoten
<code>:IS_CHILD_OF</code>	Zeigt auf den in der XML-Hierarchie übergeordneten Knoten
<code>:FIRST_CHILD_OF</code>	Zeigt vom ersten untergeordneten auf den übergeordneten Knoten.

Kante	Bemerkungen
:LAST_CHILD_OF	Zeigt vom letzten untergeordneten auf den übergeordneten Knoten.

Die folgende Abbildung zeigt einen kleinen Ausschnitt aus der TEI-XML-Datei der Patzig-Vorlesungsmitschrift.

```

rendition="#aq">po&#x017F;thumi&#x017F;chen</hi>
Werke auf-<lb/>
gedeckt u. <subst><del rendition="#s">&#x017F;eine
Fehler</del><add
place="superlinear">die&#x017F;e</add></subst> zum
Theil erka<supplied reason="damage"
resp="#BF">n&#x0303;t,</supplied><lb/>
wen&#x0303; er redete von häßlichen u.

```

Abbildung 9.2: XML-Beispiel aus der TEI-XML-Datei der Patzig-Vorlesungsmitschrift.

Dieser Abschnitt wird in der folgenden Abbildung im Graphen gezeigt. In der Abbildung des XML-Ausschnittes sind jene Teile blau markiert, die sich auch in der Graphabbildung befinden.

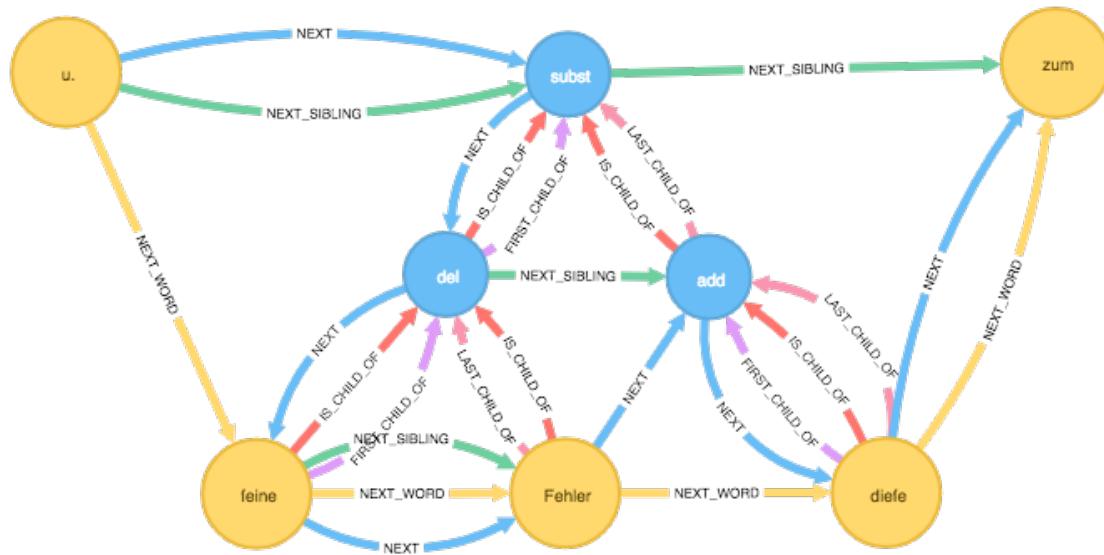


Abbildung 9.3: XML-Beispiel im Graphen.

So lassen sich beispielsweise die von einem `<add>`-Element umfassten Wörter abfragen, in dem man ausgehend vom `add`-Knoten der `FIRST_CHILD_OF`-Kante rückwärts

folgt, anschließend vom gefundenen Knoten den `NEXT_SIBLING`-Kanten so lange folgt, bis man über eine `LAST_CHILD_OF`-Kante wieder zum `add`-Knoten zurückgeführt. Der entsprechende cypher-Query sieht wie folgt aus:

```
MATCH
  (n:XmlTag {_name:'add'})
  -[:FIRST_CHILD_OF]-(w1:XmlWord)
  -[:NEXT_WORD*..5]->(w2:XmlWord)
  -[:LAST_CHILD_OF]->(n)
RETURN * LIMIT 25
```

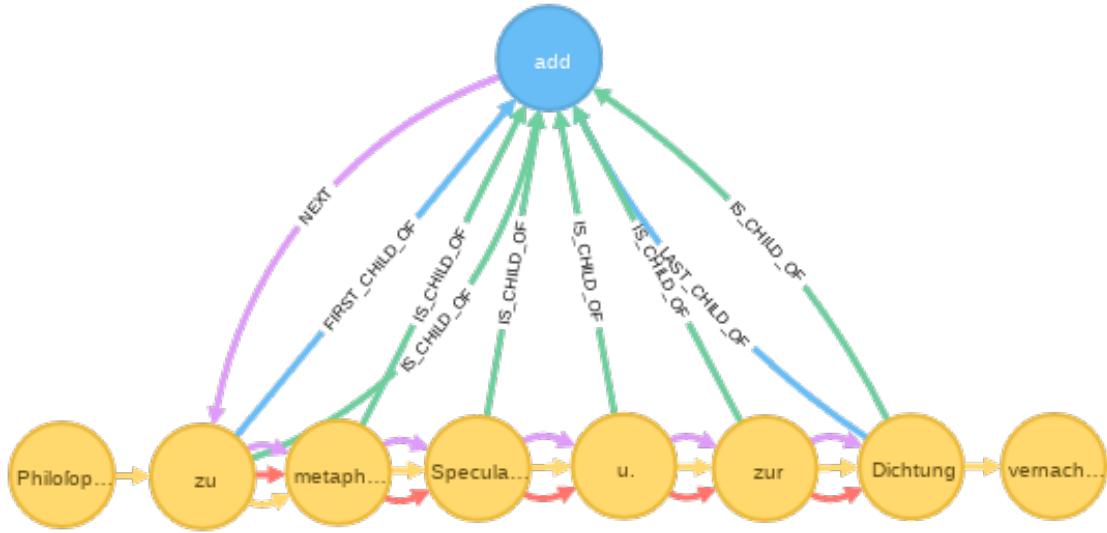


Abbildung 9.4: XML-Hierarchie eines `<add>`-Elements und der von ihm umfassten Wörter im Graphen.

In einem zweiten Schritt kann der so entstandene Graph mit Hilfe von cypher-Querys weiter bearbeitet werden. Die Graphdatenbank neo4j ist schemafrei und somit können nun über die importierten XML-Strukturen weitere Erschließungsstrukturen gelegt werden, ohne dass ein XML-Parser sich über das nicht mehr wohlgeformte XML beschwert. Zu beachten ist bei jedem Schritt, ob wieder der Schritt zurück nach XML getätigert werden soll. Sicherlich ist es kein größeres Problem, eine in eine Graphdatenbank importierte XML-Datei wieder als solche zu exportieren. Ist der Graph aber mit weiteren Informationen angereichert, so muss geklärt werden, ob, und wenn ja wie, diese zusätzlichen Informationen in wohlgeformtes XML transformiert werden können.

## 9.5 Das DTA-Basisformat im Graphen

Das DTA-Basisformat ist ein Subset der TEI und bietet für Textphänomene jeweils nur eine Annotationsmöglichkeit. Damit wird die in der TEI vorhandene Flexibilität

bei der Auszeichnung eingeschränkt, um damit einen höheren Grad an Interoperabilität zu erreichen. Das DTA-Basisformat folgt den P5-Richtlinien der TEI, trifft aber eine Tag-Auswahl der für die Auszeichnung historischer Texte notwendigen Elemente.

Im folgenden Abschnitt werden für ausgewählte Elemente des DTA-Basisformats mögliche Modellierungsformen im Graphen beschrieben. Zum äußeren Erscheinungsbild wird der Seitenfall sowie Spalten- und Zeilenumbrüche berücksichtigt. Bei den Textphänomenen werden Absätze, Schwer- und Unleserliches behandelt. Inhaltlich werden beschränkt es sich auf die Kapiteleinteilung und Inline-Auszeichnungen. Abschließend werden noch editorische Eingriffe im Graph modelliert. Für die Metadaten werden keine Modellierungsvorschläge formuliert, da diese sich sauber im XML-Baum darstellen lassen und keine Überlappungsprobleme etc. entstehen.

## 9.6 Layoutstrukturen des Dokuments

### 9.6.1 Graphenmodellierung von Zeilen

Nehmen wir als Beispiel Zeilenwechsel auf einer Seite des Patzig-Manuskripts ([http://www.deutsches-textarchiv.de/book/view/patzig\\_msgermfol841842\\_1828/?hl=Himalaja&p=39](http://www.deutsches-textarchiv.de/book/view/patzig_msgermfol841842_1828/?hl=Himalaja&p=39)).<sup>9</sup>

```
... Die<1b/>
Jnder hegen die große Verehrung vor<1b/>
dem Himalaja Gebirge. ...
```

Im Graphen sieht die Stelle wie folgt aus:

Das leere `<1b/>`-Element steht für die Markierung eines Zeilenanfangs (*line begins*). Der Graph soll nun so umgebaut werden, dass die Zeile durch einen `line`-Knoten gekennzeichnet wird, von dem aus eine `FIRST_CHILD_OF`-Kante mit dem ersten Wort der Zeile und eine `LAST_CHILD_OF`-Kante mit dem letzten Wort der Zeile verbunden ist.

Mit dem folgenden cypher-query kommt man den auf der Abbildung sichtbaren Subgraphen:

```
MATCH (n0:XmlWord)-[:NEXT_WORD]->
(n1:XmlWord {DtaID:10272})-[:NEXT_WORD*..8]->(n2:XmlWord) ,
(n1)<-[:NEXT]-(t1:XmlTag {_name:'1b'}) ,
(n3:XmlWord {DtaID:10277})-[:NEXT]->(t2:XmlTag {_name:'1b'}) 
RETURN * LIMIT 20;
```

Im folgenden Schritt wird ein `line`-Knoten erzeugt, der die Zeile darstellen soll. Mit diesem werden dann das erste und das letzte Wort der Zeile verbunden.

---

<sup>9</sup>URL des Beispieltexes: [http://www.deutsches-textarchiv.de/book/view/patzig\\_msgermfol841842\\_1828/?hl=Himalaja&p=39](http://www.deutsches-textarchiv.de/book/view/patzig_msgermfol841842_1828/?hl=Himalaja&p=39)  
abgerufen am 02.01.2018.

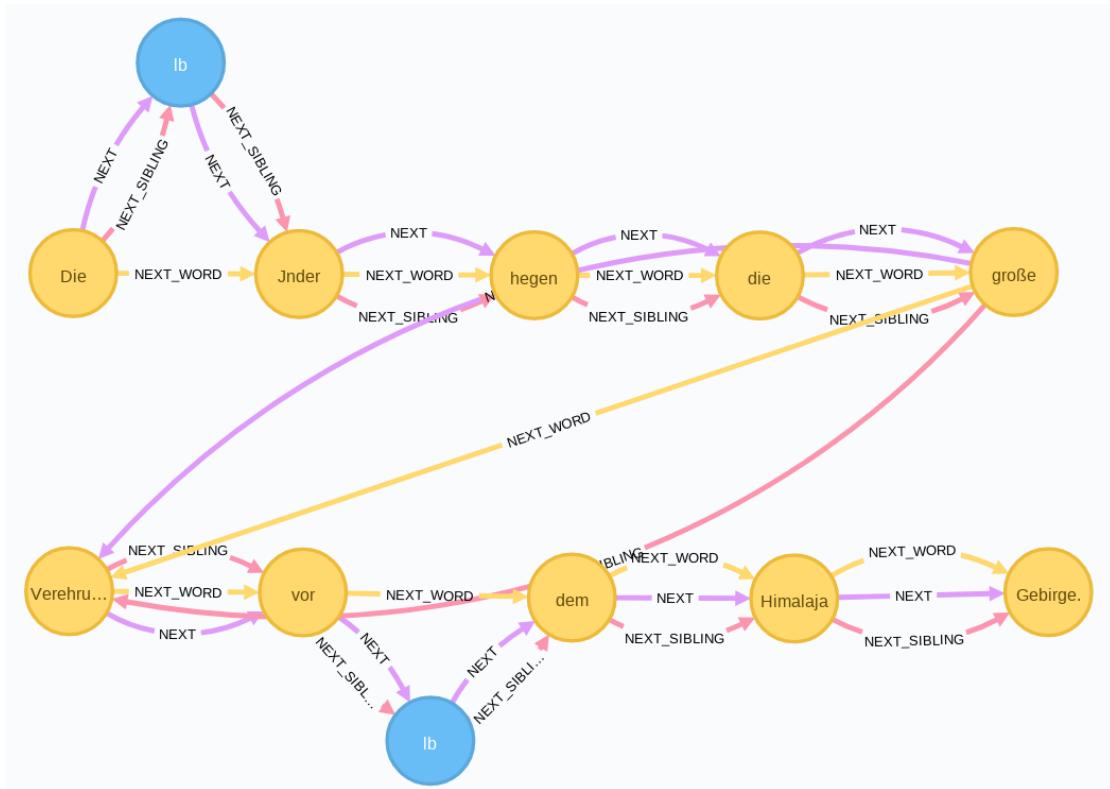


Abbildung 9.5: <lb/>-Element im Graphen

```

MATCH (n0:XmlWord)-[:NEXT_WORD]->
  (n1:XmlWord {DtaID:10272})-[:NEXT_WORD*..8]->(n2:XmlWord),
  (n1)<-[:NEXT]->(t1:XmlTag {_name:'lb'}),
  (n3:XmlWord {DtaID:10277})-[:NEXT]->(t2:XmlTag {_name:'lb'})
MERGE (n3)<-[:LAST_CHILD_OF]->(l:line {name:'line'})-[:FIRST_CHILD_OF]->(n1)
DETACH DELETE t1, t2
RETURN * LIMIT 20;

```

Im Graphen sieht das Ergebnis wie folgt aus:

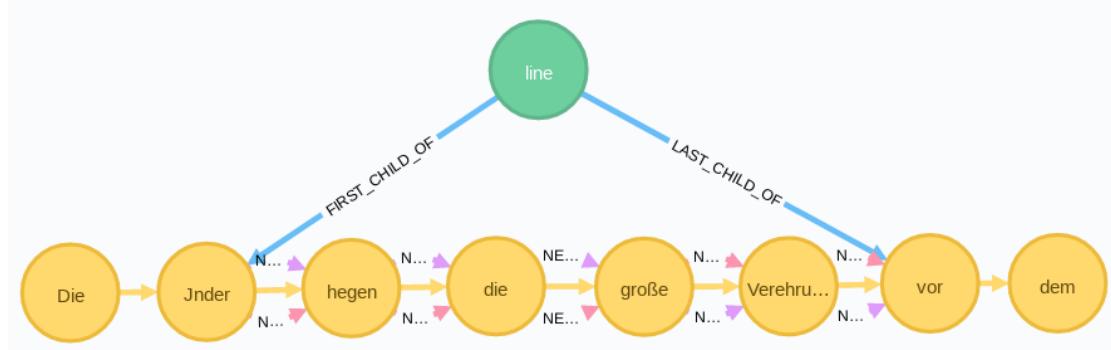


Abbildung 9.6: <lb/>-Element im Graphen

### 9.6.2 Zeilenwechsel mit Worttrennungen

Nun kommt es im Bereich der Zeilenwechsel sehr häufig zu Worttrennungen. Als Beispiel nehmen wir folgende Zeile, die sich auf der gleichen Seite wie das eben behandelte Beispiel befindet:

```

... Die Ken&#x0303;t-<lb/>
niß des Jahres durch nicht von einer Nation<lb/>
auf ...

```

Im Graphen sieht die Stelle wie folgt aus:

Mit dem folgenden cypher-query kommt man den auf der Abbildung sichtbaren Supergraphen:

```

MATCH (n0:XmlWord {DtaID:10197})-[:NEXT_WORD]->
  (n1:XmlWord)-[:NEXT_WORD*..9]->(n2:XmlWord),
  (n1)-[:NEXT]->(t1:XmlTag {_name:'lb'}),
  (n3:XmlWord {DtaID:10207})-[:NEXT]->(t2:XmlTag {_name:'lb'})
RETURN * LIMIT 20;

```

Das <lb/>-Element trennt das Wort Kenntniß.<sup>10</sup> Im nächsten Schritt werden nun die

<sup>10</sup>Zur einfacheren Lesbarkeit wurden im Wort *Kenntniß* die Sonderzeichen normalisiert.

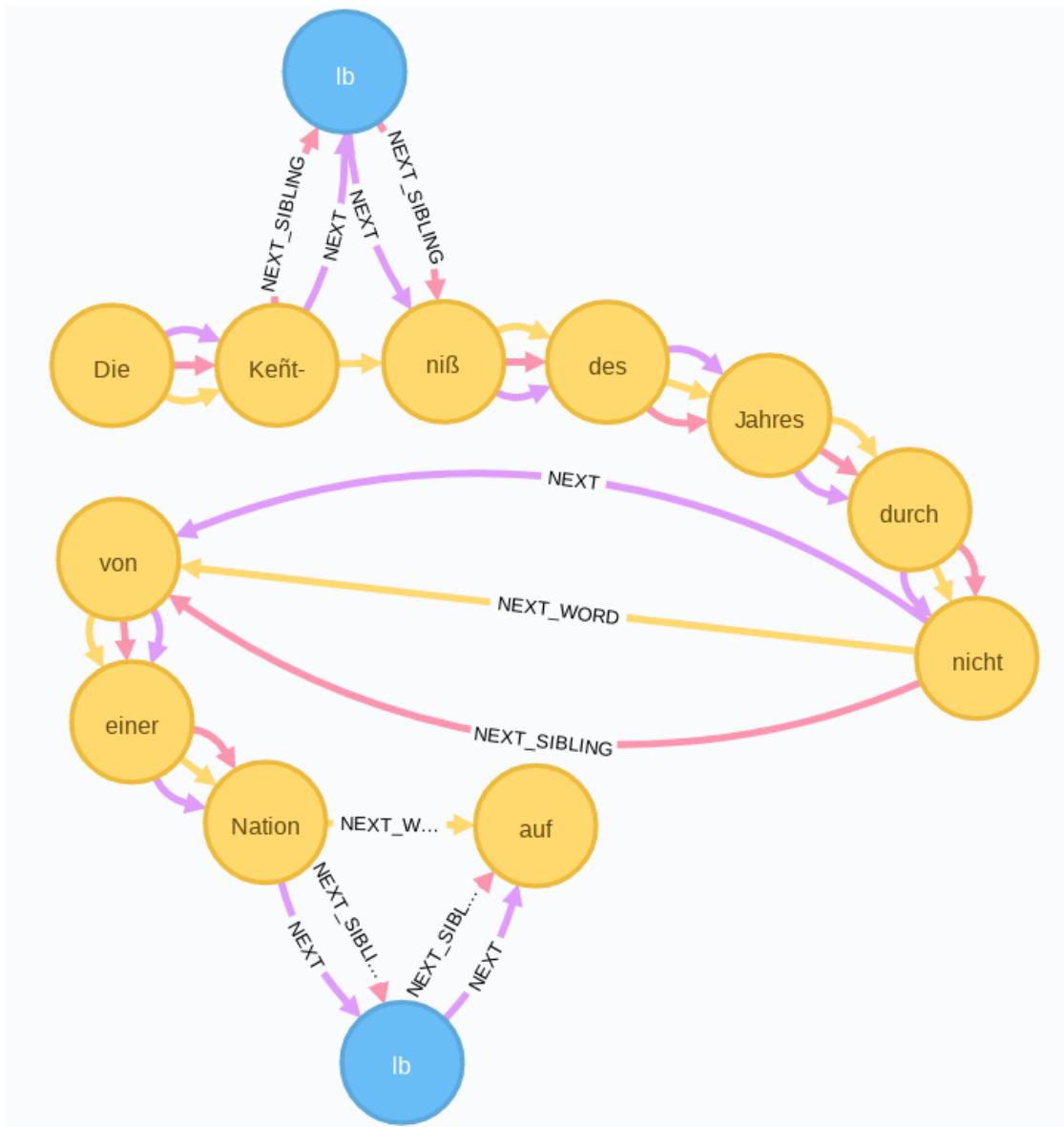


Abbildung 9.7: <lb>-Element im Graphen

beiden getrennten Wortknoten **Kennt-** und **niß** im zweiten Wortknoten **niß** zusammengefasst. Der erste Wortknoten **Kennt-** inkl. seiner Kanten wird gelöscht und eine neue **NEXT-** und **NEXT\_WORD**-Kante zwischen dem **niß**-Wortknoten und dem vorhergehenden **Die**-Wortknoten erstellt. Die Informationen, an welcher Stelle das Wort getrennt war, wird in den Eigenschaften des neuen **Kenntniß**-Wortknotens gespeichert. In der Eigenschaft **before** steht dann der Inhalt des ursprünglich ersten Wortknotens **Kennt-** und in der Eigenschaft **after** der Inhalt des ursprünglich zweiten Wortknotens **niß**.

Hier werden die notwendigen Cypher-Befehle angezeigt:

```

MATCH (n0:XmlWord {DtaID:10197})-[:NEXT_WORD]->
(n1:XmlWord {DtaID:10198})-[:NEXT_WORD]->
(n2:XmlWord {DtaID:10200})-[:NEXT_WORD*..8]->(n3:XmlWord {DtaID:10207}),
(n1)-[:NEXT]->(t1:XmlTag {_name:'lb'}),
(n4:XmlWord {DtaID:10207})-[:NEXT]->(t2:XmlTag {_name:'lb'})
SET n2.before = left(n1.text, size(n1.text)-1)
SET n2.after = n2.text
SET n2.text = left(n1.text, size(n1.text)-1)+n2.after
MERGE (n0)-[:NEXT_WORD]->(n2)
MERGE (n4)<-[:LAST_CHILD_OF]-(l:line {name:'line'})-[:FIRST_CHILD_OF]->(n2)
DETACH DELETE t1, t2, n1
RETURN * LIMIT 20;

```

Im Graphen ergibt sich anschließend folgendes Bild:

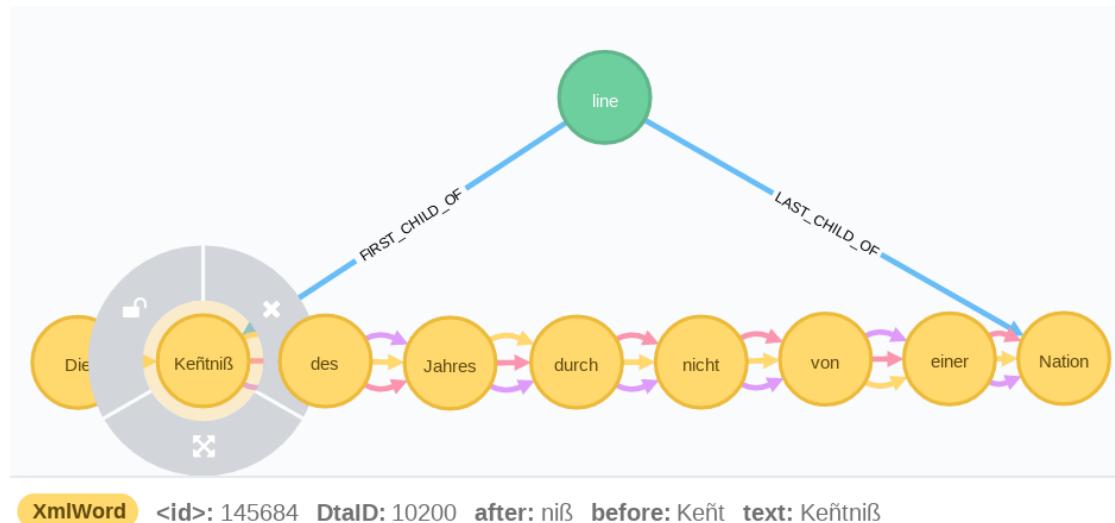


Abbildung 9.8: <1b/>-Element im Graphen herausgenommen, Wortknoten zusammengefasst

Im unteren Bereich der Abbildung sind in der Legende die Properties des Wortknotens **Kentniß** hervorgehoben. Dort erkennt man die vorher vorhandenen Wortbestandteile und

den neuen Wert der Property *text*.

### 9.6.3 Seitenfall und Faksimilezählung

Im DTA-Bf wird jeweils der Anfang einer Seite mit dem leeren Element <pb> markiert<sup>11</sup>. Das leere Element kann noch die Attribute **fac**s für die Zählung der Faksimileseiten und **n** für die auf der Seite ggf. angegebene Seitenzahl enthalten.

```
<pb facs="#f[Bildnummer]" n="[Seitenzahl]" />
```

Ist eine Seitenzahl im Faksimile falsch wiedergegeben, so wird diese originalgetreu übernommen und die richtige Seitenzahl in eckigen hinzugefügt in das **n**-Attribut übernommen.

```
<pb facs="#f[Bildnummer]"  
n=" [fehlerhafte Seitenzahl [korrigierte Seitenzahl]]" />
```

Das <pb>-Element auf den Seiten 5 und 6 aus Patzig (<http://www.deutsches-textarchiv.de/book/view/patzig>)

```
... Abwe&#x017F;enheit vom heimi&#x017F;chen Boden ent-<lb/>  
<note place="left"><figure type="stamp"/><lb/>  
</note>fernt hielt, der &#x017F;ich viel mit einem Volke<lb/>  
<fw place="bottom" type="catch">befreun-</fw><lb/>  
<pb facs="#f0006" n="2." />  
befreundete, welches durch den  
...  
in einzelnen großen Zügen zu ent-<lb/>  
werfen</hi>.</p><lb/>  
<fw place="bottom" type="catch">Nachdem</fw><lb/>  
<pb facs="#f0007" n="3." />  
<p><note place="left"><hi rendition="#u">Neue&#x017F;te  
A&#x017F;tronomi&#x017F;che Ent-<lb/>  
deckungen.</hi><lb/> ...
```

Im Graphen findet man das <pb>-Element der Seite 6 mit folgendem Query[^a825]:

```
MATCH  
(n1:XmlWord {DtaID:869})-[:NEXT]->  
(lb1:XmlTag {_name:'lb'})-[:NEXT]->  
(t2:XmlTag {_name:'fw', place:'bottom', type:'catch'})-[:NEXT_SIBLING]->  
(lb2:XmlTag {_name:'lb'})-[:NEXT_SIBLING]->  
(pb:XmlTag {_name:'pb'}),  
(n1:XmlWord)-[nw1:NEXT_WORD]->
```

<sup>11</sup>Vgl. die Dokumentation des DTA-Basisformats unter <http://www.deutsches-textarchiv.de/doku/basisformat/seitenFacNr.html> abgerufen am 25.11.2017.

<sup>12</sup>Die Beispieleseite findet sich unter [http://www.deutsches-textarchiv.de/book/view/patzig\\_msgermfol841842\\_1828/?p=5](http://www.deutsches-textarchiv.de/book/view/patzig_msgermfol841842_1828/?p=5) abgerufen am 25.11.2017.

```

(n2:XmlWord)-[nw2:NEXT_WORD]->
(n3:XmlWord)-[nw3:NEXT_WORD]->
(n4:XmlWord),
(n2:XmlWord)-[:NEXT]->(t1:XmlTag {_name:'lb'})
RETURN * LIMIT 20;

```

Im Graphen ergibt sich folgendes Bild:

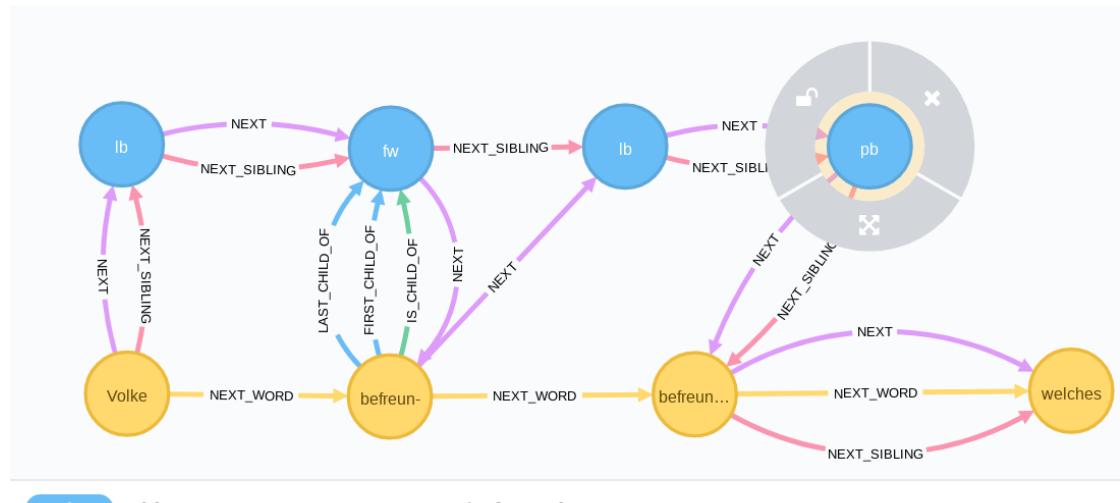


Abbildung 9.9: Der Pfad vom `<pb>`-Element zum ersten Wort der Seite *befreundet*.

Markiert ist das `<pb>`-Element der Seite 6. Im Fuß der Abbildung werden die Properties des Elements angezeigt. Der Textfluss wird durch den Wortknoten `befreun-` unterbrochen, der eine Kustode darstellt. Diese soll aus dem Textfluss herausgelöst und direkt mit dem letzten Wortknoten `Volke` über die neu eingeführte `catch_words`-Kante verbunden werden. Der `<fw>`, und der `<lb>`-Knoten werden gelöscht und der letzte Wortknoten der Seite über eine neue `NEXT`-Kante mit dem `<pb>`-Knoten verknüpft.

Hier der Query für den Umbau:

```

MATCH
(n1:XmlWord {DtaID:869})-[:NEXT]->
(lb1:XmlTag {_name:'lb'})-[:NEXT]->
(t2:XmlTag {_name:'fw', place:'bottom', type:'catch'})-[:NEXT_SIBLING]->
(lb2:XmlTag {_name:'lb'})-[:NEXT_SIBLING]->
(pb:XmlTag {_name:'pb'}),
(n1:XmlWord)-[nw1:NEXT_WORD]->
(n2:XmlWord)-[nw2:NEXT_WORD]->
(n3:XmlWord)-[nw3:NEXT_WORD]->
(n4:XmlWord),
(n2:XmlWord)-[:NEXT]->(t1:XmlTag {_name:'lb'})

```

```
DELETE nw1, nw2
DETACH DELETE t2
MERGE (n1)-[:NEXT_WORD]->(n3)
MERGE (n1)-[:CATCH_WORDS]->(n2)
MERGE (n1)-[:NEXT_WORD]->(n3)
MERGE (1b1)-[:NEXT]->(n2)
RETURN * LIMIT 20;
```

Im Graphen ergibt sich folgendes Bild:

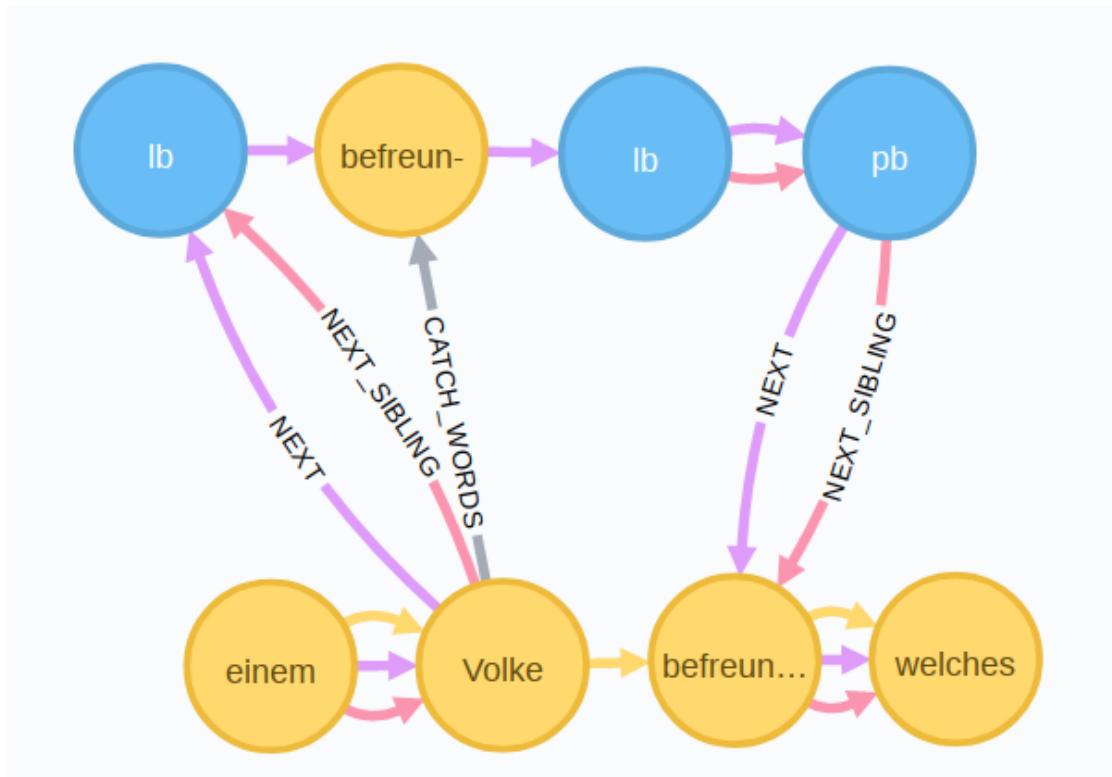


Abbildung 9.10: Die Kustode *befreun-* wird aus der NEXT\_WORD-Textkette herausgenommen und über eine CATCH\_WORDS-Kante mit dem Wortknoten *Volke* verknüpft.

Die Kustode ist nun nicht mehr über `NEXT_WORD`-Kanten mit dem Fließtext verknüpft, bleibt aber über die `CATCH WORDS`-Kante mit dem letzten Wort der Seite verbunden.

In einem zweiten Schritt müssen nun die beiden `<pb/>`-Elementknoten zu einem neu einzuführenden `page`-Knoten zusammengeführt werden. Hierfür lassen wir uns im nächsten cypher-Query alle `<pb/>`-Knoten mit einer DtaID kleiner als 875 anzeigen, da diese vor dem `<pb/>`-Knoten der Seite 6 mit der DtaID 874 liegen:

```
MATCH (n:XmlTag {_name:'pb'})
```

```
WHERE n.DtaID < 875
RETURN n;
```

"n"
{"_name": "pb", "fac": "#f0001", "DtaID": 444}
{"_name": "pb", "fac": "#f0002", "DtaID": 445}
{"_name": "pb", "fac": "#f0003", "DtaID": 455}
{"_name": "pb", "fac": "#f0004", "DtaID": 558}
{"_name": "pb", "fac": "#f0005", "DtaID": 562, "n": "1."}
{"_name": "pb", "fac": "#f0006", "DtaID": 874, "n": "2."}

Abbildung 9.11: Tabellenansicht aller <pb/>-Knoten mit einer DtaID kleiner als 875.

Aus der Tabellenansicht ist zu entnehmen, dass Seite 5 von den <pb/>-Elementen mit der DtaID 562 und 874 eingefasst wird.

Der cypher-Query zum Einfügen des page-Knoten sieht wie folgt aus:

```
MATCH
(pb1:XmlTag {DtaID:562, _name:'pb'})-[n1:NEXT*..5]->(w1:XmlWord {DtaID:565}),
(pb2:XmlTag {DtaID:874, _name:'pb'})-<-[n2:NEXT*..5]-(w2:XmlWord {DtaID:872})
MERGE
(w1)<-[:FIRST_CHILD_OF]-(page:page {fac: '#f0005', n:1})-[:LAST_CHILD_OF]->(w2)
RETURN pb1, w1, pb2, w2, page;
```

#### 9.6.4 Absätze

Absätze werden im DTA-Basisformat mit dem <p>-Element eingefasst. Im Manuskript von Patzig finden sich insgesamt 238 mit dem <p>-Element eingefasste Textabschnitte<sup>14</sup>.

<sup>13</sup>Die Darstellung der Wortkette ist zwischen den Wortknoten *der* und *einem* zu Gunsten der Übersichtlichkeit gekürzt.

<sup>14</sup>Die Anzahl der <p>-Elemente im Graph erhält man mit der Abfrage MATCH (n:XmlTag {\_name:'p'}) RETURN count(n);

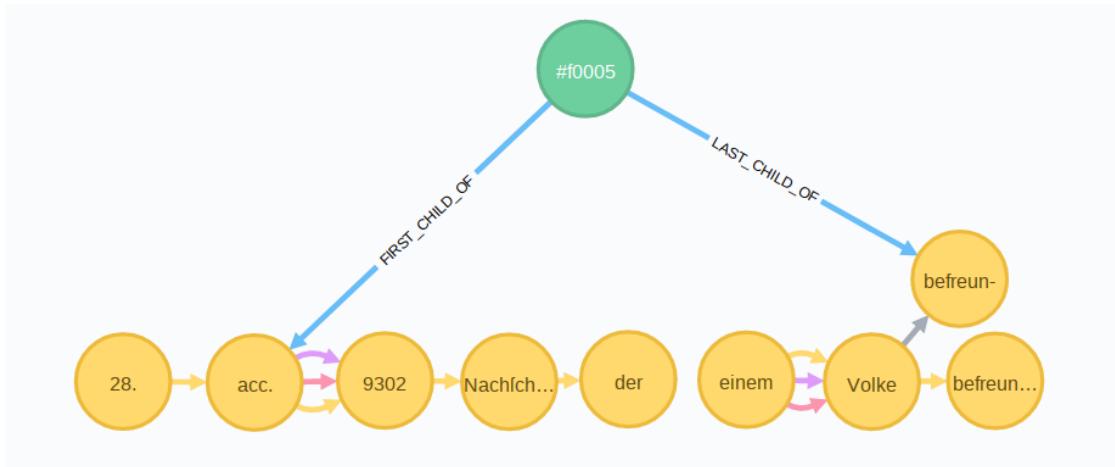


Abbildung 9.12: Die Seite wird modelliert mit dem page-Knoten #0005 der mit dem ersten Wort über eine FIRST\_CHILD\_OF- und mit dem letzten Wort der Seite über eine LAST\_CHILD\_OF-Kante verknüpft ist.<sup>13</sup>

```

<cp>Geogno&#x017F;ie, <choice><orig>Metereologie</orig><reg resp="#BF">Meteorologie</reg></choice> u. Phy&#x017F;iologie<lb/>
der <choice><abbr>Pflanz&#xFFFC;</abbr><expan resp="#BF">Pflanzen</expan></choice> haben mich den größten Theil<lb/>
meines Lebens be&#x017F;chäftigt u. &#x017F;o hoffe ich<lb/>
in die&#x017F;en Gegen&#x017F;tänden mich &#x017F;o deutlich<lb/>
zu mache, <choice><abbr>dß</abbr><expan resp="#BF">daß</expan></choice> auch die mit mindern <choice><abbr>Vorken&#x0303;t-<lb/>
ni&#x017F;&#x017F;&#xFFFC;</abbr><expan resp="#BF">Vorken&#x0303;t-<lb/>
ni&#x017F;&#x017F;en</expan></choice> meinen Vorträgen folgen kön&#x0303;en:</p><lb/>

```

Abbildung 9.13: XML-Auszug aus Patzig mit einem Absatz als Beispiel.

Da das `<p>`-Element im Unterschied zu den leeren Elementen wie `pb` oder `1b` ein öffnendes und schließendes Tag hat, wird beim Import der TEI-Xml-Datei durch den Importer schon ein `p`-Knoten erstellt, der mit einer `FIRST_CHILD_OF`-Kante mit dem ersten Wort des Absatzes und mit einer `LAST_CHILD_OF`-Kante mit dem letzten Wort des Absatzes verknüpft ist.

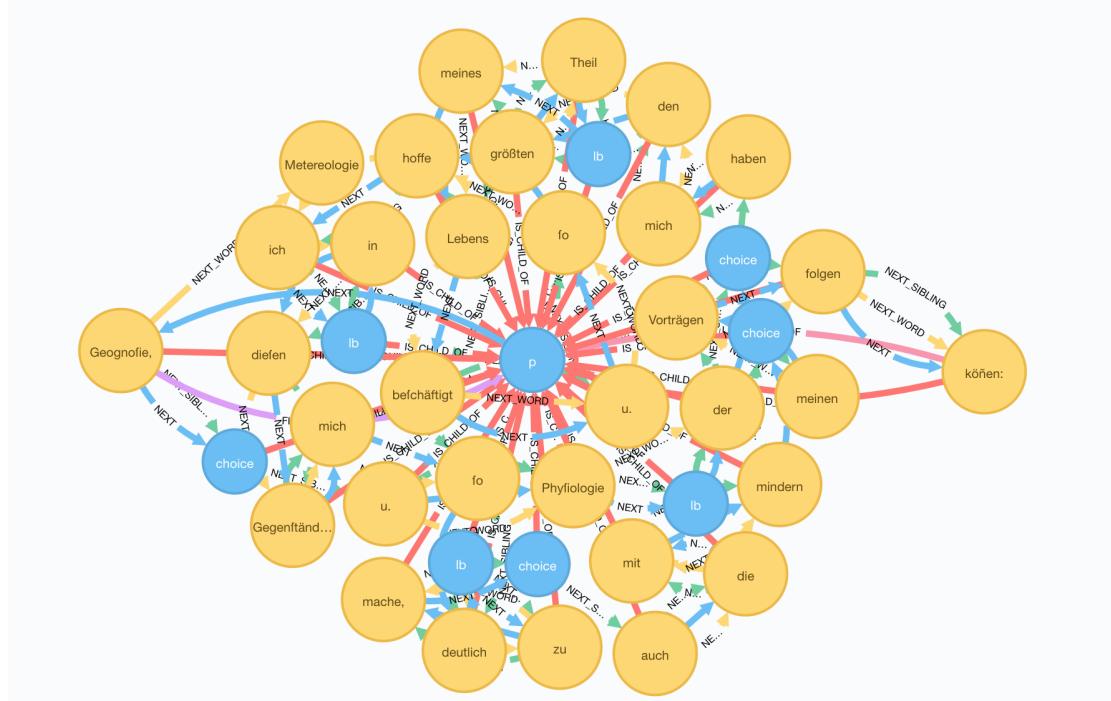


Abbildung 9.14: Ein Teil des gleichen Absatzes aus Patzig im Graphen.

Alle Wörter eines Absatzes sind darüber hinaus über `NEXT_SIBLING`-Kanten in der Textreihenfolge verknüpft.

## 9.6.5 Kapiteleinteilung

Im DTA-Basisformat wird bei der Transkription von Büchern die Kapitaleinteilung mit verschachtelten `div`-Element vorgenommen. Das im `div`-Element erlaubte `@n`-Attribut gibt die Strukturebene an. Über das `@type`-Attribut kann der Typ des Kapitels näher spezifiziert werden. Eine Liste der möglichen Werte für das Attribut findet sich unter <http://deutschestextarchiv.de/doku/basisformat/div.html>.

Für Manuskripte, wie der hier behandelten Vorlesungsmitschrift von Patzig gibt es unter <http://deutschestextarchiv.de/doku/basisformat/msKapitel.html> noch zwei zusätzliche Werte für das @type-Attribut, nämlich *session* für Vorlesungsmitschriften und *letter* für Briefe.

Mit folgendem cypher-Query erhalten wir die in Patzig verwendeten Werte für das @type-Attribut des div-Elements.

```
MATCH (n:XmlTag {_name:'div'})
RETURN n.type, count(n.type) AS Anzahl ORDER BY Anzahl DESC;
```

n.type	Anzahl
session	62
null	0

Es sind also insgesamt 62 Kapitel vom Typ *session* (Vorlesungsmitschrift) enthalten. Mit folgendem cypher-Query wird die Kapitelstruktur der ersten Kapitel und der darunter liegenden Ebenen bis zum jeweils ersten und letzten Wort des Kapitels angezeigt.

```
MATCH
p1 = shortestPath(
    (div:XmlTag {_name:'div'})->[:FIRST_CHILD_OF*..20]-(w1:XmlWord)),
p2 = shortestPath(
    (div:XmlTag {_name:'div'})->[:LAST_CHILD_OF*..20]-(w2:XmlWord))
RETURN p1,p2 LIMIT 20;
```

Mit dem folgenden cypher-Query wird das erste Wort des Kapitels über eine FIRST\_CHILD\_OF-Kante und das letzte Wort des Absatzes über eine LAST\_CHILD\_OF-Kante mit dem div-Knoten verbunden. Um die neu erstellen Kanten von den vom Importer erstellen zu unterscheiden erhalten diese die Proptery *type* mit dem Wert *graph*. Um die div-Knoten von den anderen XmlTag-Knoten unterscheiden zu können erhalten sie das zusätzliche Label *Session*.

```
MATCH
p1 = shortestPath(
    (div:XmlTag {_name:'div'})->[:FIRST_CHILD_OF*..20]-(w1:XmlWord)
    ),
p2 = shortestPath(
    (div:XmlTag {_name:'div'})->[:LAST_CHILD_OF*..20]-(w2:XmlWord)
    )
MERGE (w1)-[:FIRST_CHILD_OF {type:'graph'}]->
    (div)-[:LAST_CHILD_OF {type:'graph'}]->(w2)
SET div:Session
RETURN * LIMIT 20;
```

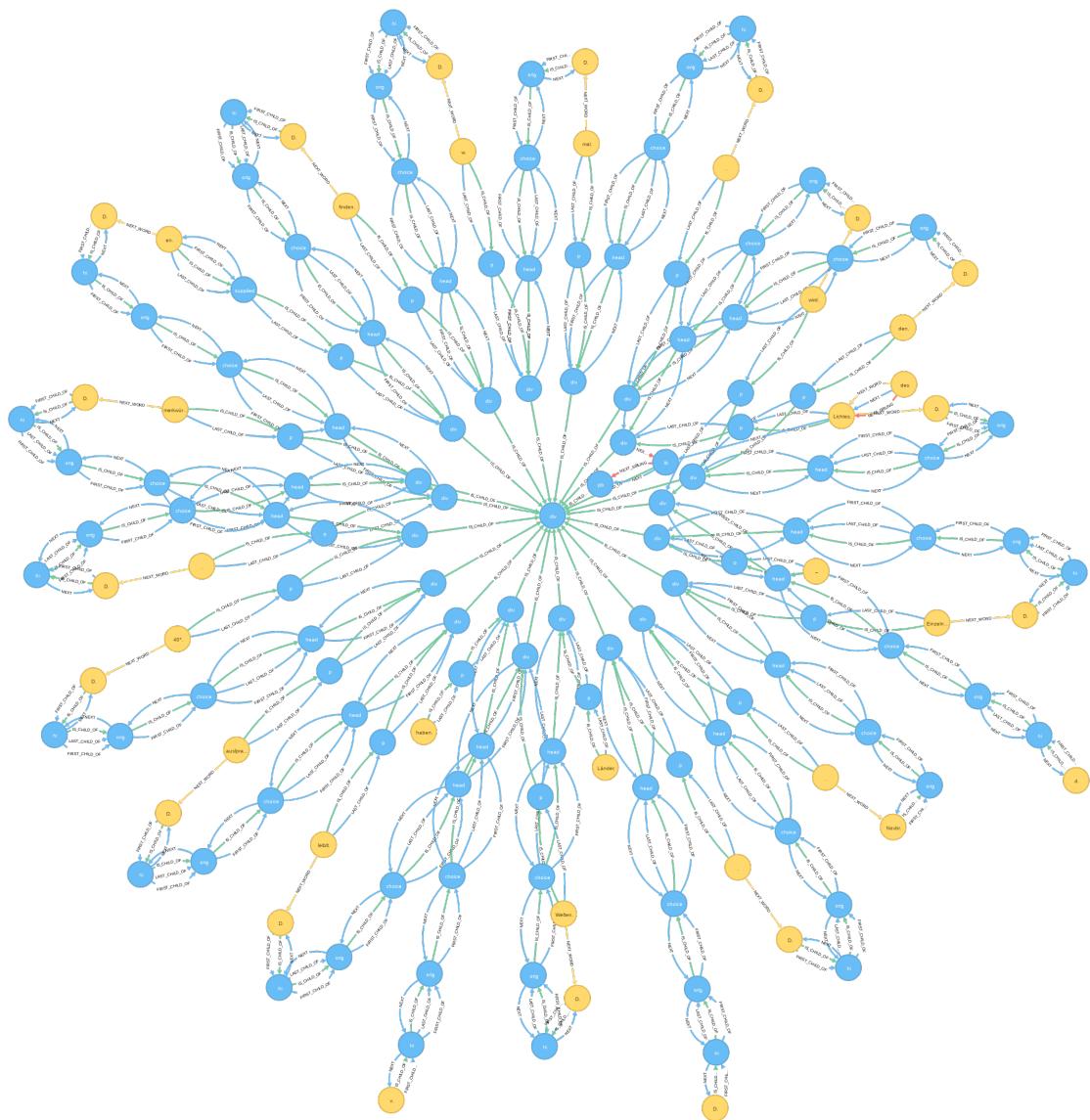


Abbildung 9.15: Struktur der ersten Kapitel mit dem jeweils ersten und letzten Wort.

## 9.7 Editorische Eingriffe

### 9.7.1 Hinzufügungen und Tilgungen

Die Elemente `<add>` und `<del>` werden für Kennzeichnung von Tilgungen und Hinzufügungen des Autors oder von späteren Bearbeitern verwendet.

### 9.7.2 `<add>`-Element

Dabei können die Umstände der Änderungen beim `<add>`-Element mit dem `@place`-Attribut näher beschrieben, welches die in der folgenden Tabelle angegebenen Werte annehmen darf<sup>15</sup>:

Element	<code>@place</code> -Wert	Bedeutung
<code>&lt;add&gt;</code>	superlinear	über der Zeile eingetragen
<code>&lt;add&gt;</code>	sublinear	unter der Zeile eingetragen
<code>&lt;add&gt;</code>	intralinear	innerhalb der Zeile eingetragen
<code>&lt;add&gt;</code>	across	über den ursprünglichen Text geschrieben
<code>&lt;add&gt;</code>	left	am linken Rand eingetragen
<code>&lt;add&gt;</code>	right	am rechten Rand eingetragen

Mit folgenden cypher-Query erhält man die Verteilung der Attributwerte.

```
MATCH (n:XmlTag {_name:'add'})  
RETURN n.place, count(n.place) AS Anzahl ORDER BY Anzahl DESC;
```

n.place	Anzahl
across	436
superlinear	268
intralinear	60
left	16
sublinear	2

#### 9.7.2.1 `<del>`-Element

Die mit dem `<del>`-Element gekennzeichneten Tilgungen können mit dem `@rendition`-Attribut näher beschrieben werden, dessen mögliche Werte in der folgenden Tabelle

---

<sup>15</sup>Vgl. hierzu <http://deutschestextarchiv.de/doku/basisformat/msAddDel.html>.

angegeben sind<sup>16</sup>.

Element	@rendition-Wert	Bedeutung
<del>	#ow	Tilgung durch Überschreibung des ursprünglichen Textes
<del>	#s	Tilgung durch Streichung
<del>	#erased	Tilgung durch Radieren, Auskratzen

Mit folgenden cypher-Query erhält man die Verteilung der Attributwerte.

```
MATCH (n:XmlTag {_name:'add'})
RETURN n.rendition, count(n.rendition) AS Anzahl
ORDER BY Anzahl DESC;
```

n.rendition	Anzahl
#ow	436
#s	268
#erased	60

### 9.7.3 Umbau von <add>- und <del>-Elementen in einer <subst>-Umgebung

Der Umbau wird an einem Beispieltext der Seite 32 des Patzig-Manuskripts durchgeführt<sup>17</sup>.

```
rendition="#aq">po&#x017F;thumi&#x017F;chen</hi>
Werke auf-<lb/>
gedeckt u. <subst><del rendition="#s">&#x017F;eine
Fehler</del><add
place="superlinear">die&#x017F;e</add></subst> zum
Theil erka<supplied reason="damage"
resp="#BF">n&#x0303;t,</supplied><lb/>
wen&#x0303; er redete von häßlichen u.
```

Abbildung 9.16: -Beispiel in der XML-Ansicht.

Im Graphen findet man die entsprechende Stelle mit folgendem cypher-Query.

```
MATCH
(w1:XmlWord)-[r1:NEXT_WORD]->
(w2:XmlWord)-[r2:FIRST_CHILD_OF]->
```

<sup>16</sup>Vgl. hierzu <http://deutschestextarchiv.de/doku/basisformat/msAddDel.html>.

<sup>17</sup>Vgl. [http://www.deutschestextarchiv.de/book/view/patzig\\_msgermfol841842\\_1828/?hl=zum&p=32](http://www.deutschestextarchiv.de/book/view/patzig_msgermfol841842_1828/?hl=zum&p=32).

```

(t1)-[r3:FIRST_CHILD_OF]->
(s:XmlTag {_name:'subst', DtaID:8248})
<-[r4:LAST_CHILD_OF]-(t2)
<-[r5:LAST_CHILD_OF]-(w4:XmlWord)
-[r6:NEXT_WORD]->(w5:XmlWord),
(w2)-[r7:NEXT_WORD]->(w3)-[r8:NEXT_WORD]->(w4)
RETURN *;

```

Der Query gruppiert sich um den **s**-Knoten, der das **subst**-Element darstellt und es über die DtaID identifiziert. Vom **s**-Knoten ausgehend, folgt der Pfad einerseits über **FIRST\_CHILD\_OF**-Kanten zum **n3**-Knoten (add-Element) und zum **n2**-Knoten, der schließlich das Wort *seine* darstellt. Über die **LAST\_CHILD\_OF**-Kante geht es zum **n4**-Knoten (del-Element) zum **n5**-Wortknoten, der das Wort *diese* darstellt. Im zweiten Teil des MATCH-Befehls wird der Pfad zwischen dem Wort *seine* und *diese* ermittelt und schließlich alles ausgegeben.

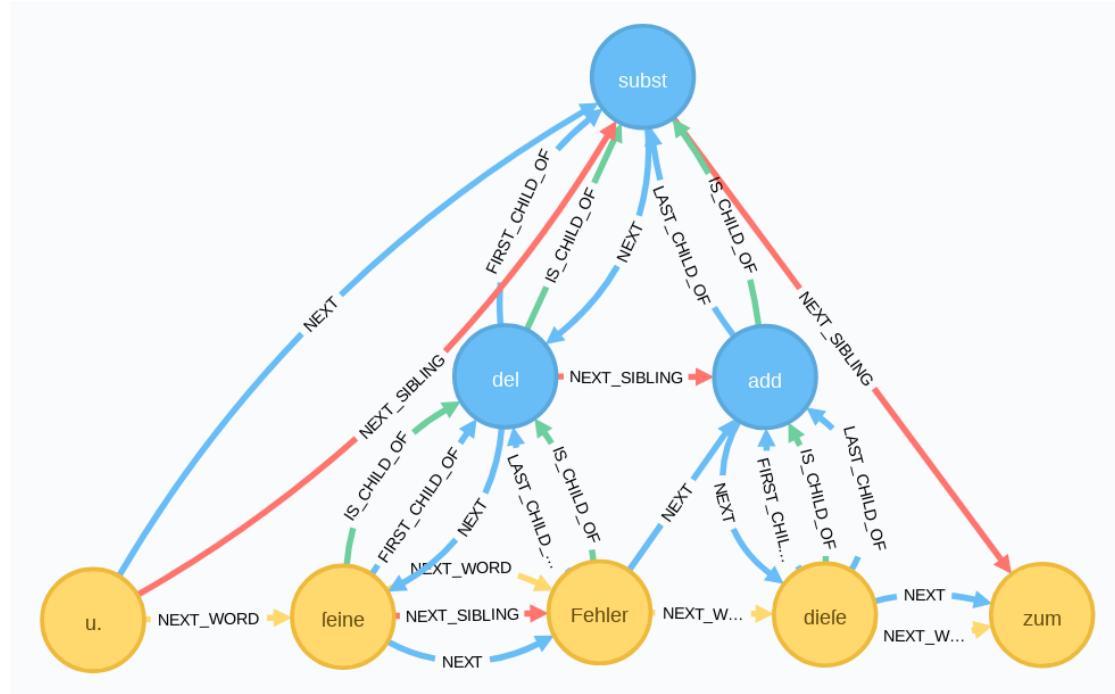


Abbildung 9.17: -Beispiel in der Graph-Ansicht.

cyper-Query für den umgebaut

```

MATCH
(w1:XmlWord)-[r1:NEXT_WORD]->
(w2:XmlWord)-[r2:FIRST_CHILD_OF]->
(t1)-[r3:FIRST_CHILD_OF]->
(s:XmlTag {_name:'subst', DtaID:8248})

```

```

<-[r4:LAST_CHILD_OF]-(t2)
<-[r5:LAST_CHILD_OF]-(w4:XmlWord)
-[r6:NEXT_WORD]->(w5:XmlWord),
(w2)-[r7:NEXT_WORD]->(w3)-[r8:NEXT_WORD]->(w4)
DELETE r1, r8
SET r8.variant_type='add'
CREATE (w1)-[:NEXT_WORD{variant_type:'add'}]->(w4)
CREATE (w1)-[:NEXT_WORD{variant_type:'del'}]->(w2)
CREATE (w3)-[:NEXT_WORD{variant_type:'del'}]->(w5)
SET r7.variant_type='del'
RETURN *;

```

Das Ergebnis erhält man über den folgenden Query.

```

MATCH
(w1:XmlWord)-[r1:NEXT_WORD]->
(w2:XmlWord)-[r2:FIRST_CHILD_OF]->
(t1)-[r3:FIRST_CHILD_OF]->
(s:XmlTag {_name:'subst', DtaID:8248})
<-[r4:LAST_CHILD_OF]-(t2)
<-[r5:LAST_CHILD_OF]-(w4:XmlWord)
-[r6:NEXT_WORD]->(w5:XmlWord),
(w2)-[r7:NEXT_WORD]->(w3)-[r8:NEXT_WORD]->(w4)

```

## 9.8 Zusammenfassung

In diesem Kapitel wurden exemplarisch die XML-Strukturen für Layout (Zeilen (1b), Seiten (pb), Absätze (p)), Struktur (Kapitel (div)) und editorische Eingriffe (**subst**, **add** und **del**) in Graphstrukturen überführt. Die entsprechenden Tags wurden in einen Annotationsknoten zusammengeführt. Mit diesem Knoten wird jeweils der erste und der letzte betroffene Wortknoten mit einer **FIRST\_CHILD\_OF**- und einer **LAST\_CHILD\_OF**-Kante verknüpft. Damit entstehen klare Annotationsstrukturen, die aber offensichtlich überlappen. Dies stellt im Graphen aber kein Problem dar.

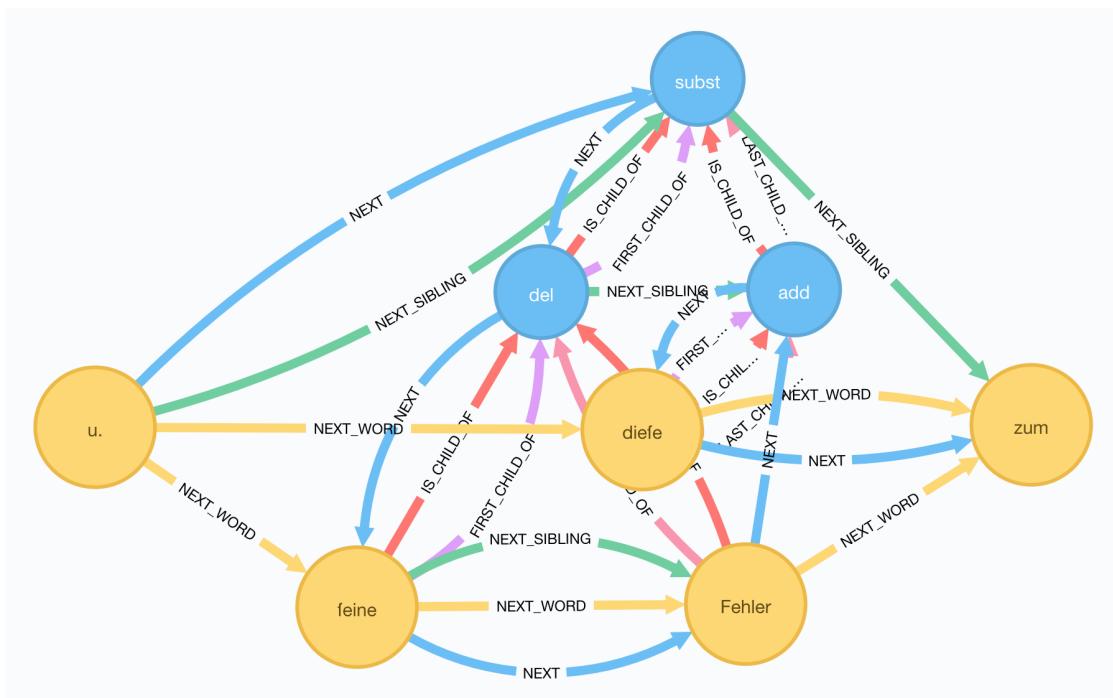


Abbildung 9.18: `<subst>`-Beispiel nach dem Graph-Umbau.

## 10 JSON Import mit den Daten der Germania Sacra

(Dieser Abschnitt befindet sich in Bearbeitung)

## 10.1 Das Projekt Germania Sacra

Das Projekt Germania Sacra erschließt die Quellen der Kirche des alten Reiches.<sup>[^eea]</sup> Dabei werden die Kirche und ihre Institutionen von den Anfängen im 3./4. Jahrhundert bis zu deren Auflösung am Beginn des 19. Jahrhunderts dar. Im Rahmen des Projekts werden die überlieferten Quellen nach einheitlichen Kriterien aufgearbeitet und so strukturierte Daten für Kirchengeschichte im alten Reich bereitgestellt. So bildet das Projekt die Grundlage für weiterführende Forschungen.

Neben den Bänden bietet das Projekt auch ein digitales Personenregister mit Angaben u.a. zu Personen, Bischöfen, Klöstern und Stiften. Die Daten werden über die Schnittstellen des Projekts als JSON-Daten bereitgestellt.<sup>1</sup>

## 10.2 Germania Sacra JSON

### 10.2.1 Personen

Die Daten des Projekts umfassen z.B. Angaben zu Namen, Namensalternativen, Daten zu den Personen und der institutionelle Anbindung. Der folgende Ausschnitt aus einer JSON-Datei umfasst beispielhaft zwei Personeneinträge.

```
{  
  "persons": [  
    {"  
      "person_vorname": "Ludold",  
      "person_name": "Ludold von Escherde",  
      "person_namensalternativen": "von Goltern (?)",  
      "person_gso": "060-02673-001",  
      "person_gesamt": "Ludold von Escherde (von Goltern (?))"  
    }  
  ]  
}
```

<sup>1</sup>Der in diesem Kapitel verwendete json-Dump wurde freundlicherweise direkt vom Projekt Germania Sacra zur Verfügung gestellt und die Verwendung in dieser Veröffentlichung sowie die weitere Verwendung genehmigt.

```

"person_gnd": "",
"person_bezeichnung": "Abt",
"person_bezeichnung_plural": "\u00c4bte",
"person_anmerkung": "",
"person_von_verbal": "1234",
"person_von": 1234,
"person_bis_verbal": "1263",
"person_bis": 1263,
"person_office_id": "282620"
}, {
    "person_vorname": "Hermann",
    "person_name": "Hermann",
    "person_namensalternativen": "",
    "person_gso": "033-02024-001",
    "person_gnd": "",
    "person_bezeichnung": "Abt",
    "person_bezeichnung_plural": "\u00c4bte",
    "person_anmerkung": "",
    "person_von_verbal": "1262",
    "person_von": 1262,
    "person_bis_verbal": "1265",
    "person_bis": 1265,
    "person_office_id": "311579"
}

```

Die folgende Abbildung zeigt beispielhaft den ersten Personeneintrag des obigen json-Beispiels von Ludold von Escherde als importierten Personenknoten im Graphen.



Abbildung 10.1: Personenknoten von Ludold von Escherde in der Graphdatenbank

Unter dem Personenknoten werden die aus dem json importierten Properties aufgelistet.

## 10.2.2 Klöster

Die folgende Abbildung zeigt die ersten drei Einträge der JSON-Datei mit den Angaben zu den Klöstern.

```
{"kloster": [ { "bezeichnung": "Adeliges Damenstift Neuburg", "ort": "Heidelberg", "bistum": "Worms", "klosterid": 20595, "Wikipedia": "#http://de.wikipedia.org/wiki/Abtei_Neuburg#", "GND": "#http://d-nb.info/gnd/4316849-8#", "GeonameID_Ortsname": 2907911, "Datum_von": 1671, "Datum_bis": 1681 }, { "bezeichnung": "Adeliges weltliches Chorfrauenstift St. Fridolin, Säckingen", "ort": "Bad Säckingen", "bistum": "Konstanz", "klosterid": 20381, "Wikipedia": "#http://de.wikipedia.org/wiki/Damenstift_S%C3%A4ckingen#", "GND": "#http://d-nb.info/gnd/4343770-9#", "GeonameID_Ortsname": 2953363, "Datum_von": 501, "Datum_bis": 1806 }, { "bezeichnung": "Adliges Damenstift Frauenalb, zuvor Benediktinerinnenkloster", "ort": "Marxzell", "bistum": "Speyer", "klosterid": 20195, "Wikipedia": "", "GND": "#http://d-nb.info/gnd/4446800-3#", "GeonameID_Ortsname": null, "Datum_von": 1180, "Datum_bis": 1803 } ]}
```

Im folgenden die cypher-queries für den Import der json-Dateien. Die json-Dateien selbst werden über Seafile mit einem Download-Link bereitgestellt.

Die folgende Abbildung zeigt beispielhaft den ersten Klostereintrag des obigen json-

Beispiele zum Adeligen Damenstift Neuburg als importierten Klosterknoten im Graphen.



Abbildung 10.2: Klosterknoten des Adeligen Damenstifts Neuburg in der Graphdatenbank

Im folgenden die cypher-queries für den Import der json-Dateien. Die json-Dateien selbst werden über Seafile mit einem Download-Link bereitgestellt.

Im ersten Abschnitt des Codebeispiels werden Indexe z.B. für die Property gnd von Personenknoten und die Property Bistum von Klosterknoten erstellt. Anschließend werden Constraints für die IDs von Kloster- und Personenknoten eingerichtet, mit denen sichergestellt wird, dass die IDs der Kloster- und Personenknosten jeweils nur einmal vorkommen können.

```
CREATE INDEX ON :Person(gnd);
CREATE INDEX ON :Person(bezeichnung);
CREATE INDEX ON :Kloster(Bistum);
CREATE CONSTRAINT ON (p:Person) ASSERT p.id IS UNIQUE;
CREATE CONSTRAINT ON (k:Kloster) ASSERT k.id IS UNIQUE;
```

Der nächste Befehl importiert aus der Personen-json-Datei die Personen in die Graphdatenbank. Die Zusatzinformationen zu den einzelnen Personeneinträgen werden jeweils als Properties des Personenknoten in der Graphdatenbank angelegt.

```
// Personenknoten erstellen
call apoc.load.json("https://seafile.rlp.net/f/456adda2cffc475ab755/?dl=1")
yield value as all
unwind all.persons as p
CREATE (p1:Person {personBezeichnungPlural:p.person_bezeichnung_plural,
  gso:p.person_gso, personOfficeId:p.person_office_id, name:p.person_name,
  gnd:p.person_gnd, anmerkung:p.person_anmerkung,
  personVonVerbal:p.person_von_verbal, bezeichnung:p.person_bezeichnung,
  personVon:p.person_von, personBisVerbal:p.person_bis_verbal,
  personBis:p.person_bis, personNamensalternativen:p.person_namensalternativen,
  vorname:p.person_vorname})
RETURN count(p1);
```

In diesem Query werden analog zu den Personen die Klöster mit den zugehörigen Informationen in die Graphdatenbank importiert.

```
// Klosterknoten erstellen
call apoc.load.json("https://seafile.rlp.net/f/91c3600003d54cc9ac83/?dl=1") yield value as v
unwind all.kloster as k
CREATE (kl:Kloster {ort:k.ort,
GeonameIDOrtsname:k.GeonameID_Ortsname,
datum:k.Datum_von, bezeichnung:k.bezeichnung, bistum:k.bistum,
wikipedia:k.Wikipedia, datumBis:k.Datum_bis, kid:k.klosterid, gnd:k.GND})
RETURN count(kl);
```

Die Zugehörigkeit eines Klosters zu einem Bistum ist in der Eigenschaft Bistum bei den jeweiligen Klosterknoten gespeichert. Aus dieser Information werden in diesem Query die Bistumsknoten erstellt und die Klosterknoten den jeweiligen Bistumsknoten zugeordnet.

```
// Bistumsknoten erstellen
MATCH (k:Kloster)
MERGE (b:Bistum {name:k.bistum})
MERGE (b)-[bi:BISTUM]-(k)
RETURN count(bi);
```

Analog zu den Bistumern werden in diesem Query die Professionen erstellt und den einzelnen Personenknoten zugeordnet.

```
//Professionsknoten erstellen
MATCH (p:Person)
MERGE (pro:Profession {name:p.bezeichnung})
MERGE (pro)-[pr:PROFESSION]-(p)
RETURN count(pr);
```

## 10.3 Zusammenfassung

In diesem Abschnitt wurden die Prinzipien für den Import von json-Dateien am Beispiel der Daten des Projekts Germania Sacra vorgestellt.

Im geplanten Folgekapitel werden die Analyse von json-Daten und der Import komplexerer json-Strukturen erläutert werden.

[^eeaa]: Zu diesem Abschnitt vgl. <http://www.germania-sacra.de/> (zuletzt abgerufen am 07.03.2019).

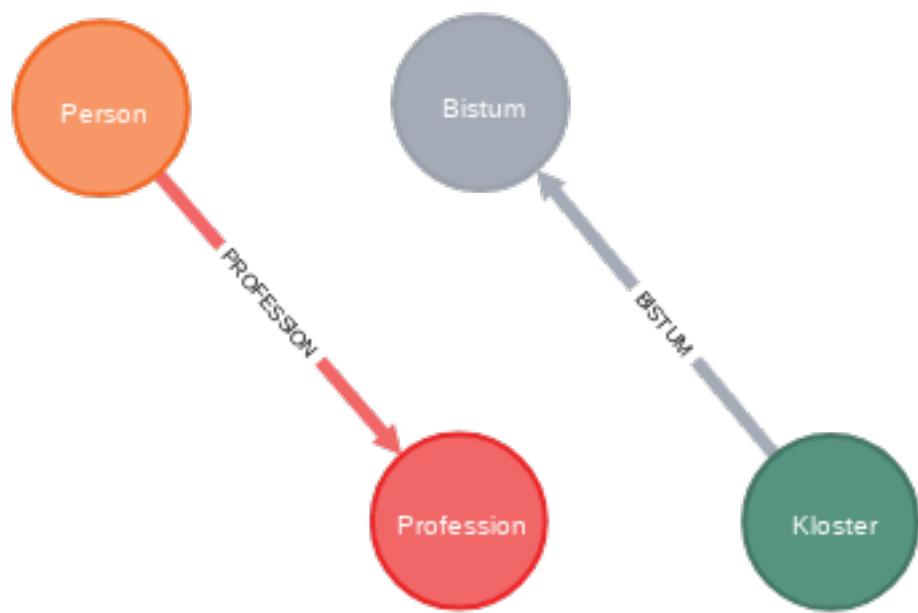


Abbildung 10.3: Schema der importierten json-Daten (Quelle:Kuczera).)

# 11 Netzwerkanalyse mit neo4j

## 11.1 Grundlagen zur Netzwerkanalyse

### 11.1.1 Vorbemerkungen

Bei der Netzwerkanalyse handelt es sich um einen relationalen Forschungsansatz, bei dem Methoden aus der Graphen- und Netzwerktheorie sowie der Statistik zur Anwendung kommen.<sup>1</sup> Aus den bisherigen Abschnitten sind Beziehungen (**Kanten**) von Entitäten (**Knoten**) in einem Netzwerk (**Graph**) bereits bekannt. Weitere wichtige Begriffe sind die **Dyade** als kleinste mögliche Analyseeinheit eines Netzwerkes. Sie umfasst alle möglichen Beziehungen zwischen zwei Knoten. Eine **Triade** bezieht alle möglichen Beziehungen zwischen drei Knoten mit ein. Bei einer **Clique** sind mindestens drei Knoten vollständig miteinander verbunden. Als **Geodätischer Abstand** (in neo4j *shortest path*) wird die kürzeste Verbindung zwischen zwei Knoten bezeichnet.

### 11.1.2 Überblick zu den Netzwerkmaßen

Grundlegende Maße in der Netzwerkanalyse sind

- **Dichte** als Quotient zwischen tatsächlichen Anzahl der Kanten und der maximal möglichen Anzahl der Kanten.
- **Entfernungsmaße**, z.B. geodätische Abstände, Durchmesser des Netzwerks etc.
- **Reziprozität** als Quotient zwischen der Anzahl bidirektonaler und einseitigen Beziehungen.
- **Clustering** beispielsweise als Anzahl und Art von Triaden.

Eine wichtige Rolle spielen auch die Zentralitätsmaße (Beispiele):

- Bei der **Degree Centrality** werden pro Knoten die Anzahl der Verbindungen zu anderen Knoten betrachtet.
- Bei der **Betweenes Centrality** wird die Anzahl der über einen Knoten laufenden möglichen Verbindungen zwischen zwei anderen Knoten erhoben.
- Die **Closeness Centrality** erhebt die Nähe zu allen anderen Knoten.
- Die **Eigenvektor Centrality** misst die Verbindung zu “einflussreichen” Knoten.

---

<sup>1</sup>Die Informationen und Abbildungen in diesem Abschnitt stammen aus dem Kurs [Historisch-archäologische Netzwerkanalyse](#) von Aline Deicke und Marjam Trautmann, der im Rahmen der International Summer School in Mainz stattfand (abgerufen am 07.02.2019).

### 11.1.3 Beispiel: Zentralitätsmaße

Die folgende Abbildung zeigt einen kleinen Beispielgraphen, an dem einige Zentralitätsmaße erklärt werden sollen.<sup>2</sup>

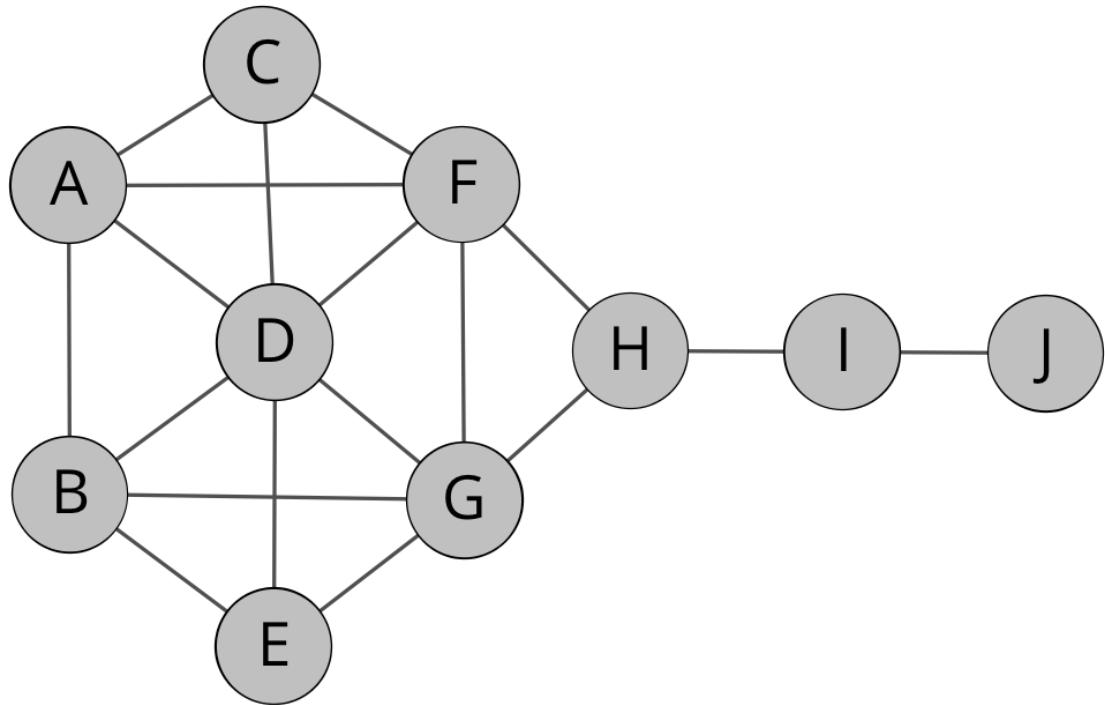


Abbildung 11.1: Beispielgraph für Zentralitätsmaße aus D. Krackhardt, Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Administrative Science Quarterly 35, 2, 1990, S. 351, <http://dx.doi.org/10.2307/2393394>.

Beispielgraph für Zentralitätsmaße aus D. Krackhardt, Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Administrative Science Quarterly 35, 2, 1990, S. 351.

- Bei der Berechnung der **Dichte** wird der Quotient zwischen der tatsächlichen Anzahl der Kanten und der maximal möglichen Anzahl der Kanten berechnet. Die Formel hierfür lautet

$$\text{Dichte des Netzwerks} = 2m / (n * (n - 1))$$

wobei m die Anzahl der vorhandenen Kanten ist und n die Anzahl der Knoten. Für

---

<sup>2</sup>Vgl. zu diesem Abschnitt D. Krackhardt, Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Administrative Science Quarterly 35, 2, 1990, S. 342-369 (abgerufen am 07.02.2019).

unser Beispiel ergibt sich damit:

$$2 * 18 / (10 * (10 - 1)) = 0,4$$

womit die Dichte des Netzwerks 0,4 beträgt.

- Die höchste **Degreenes Centrality** hat mit 6 Kanten der Knoten D, was mehr als bei allen anderen ist. Den ersten Platz bei der **Betweenes Centrality** teilen sich die Knoten D, F, G I und H.
- Die höchste **Closeness** besteht zwischen den Knoten F und G.
- Der größte **Eigenvektor** besteht zwischen den Knoten D, F und G.
- Die Knoten F und G sind **strukturell äquivalent**.

Mit diesen Vorbemerkungen als Hintergrund werden in den folgenden Abschnitten Netzwerkanalyse-Algorithmen auf eine aufbereitete Graphdatenbank der Regesten Kaiser Heinrichs IV. angewendet.

## 11.2 Die Register der Regesta Imperii

Grundlage der hier verwendeten Netzwerksdaten sind die Nennungen von Personeneinträgen im Register in den Regesten. Da die hier vorgestellten Netzwerkanalyse-Algorithmen nur mit unimodalen Netzwerken arbeiten können, also Netzwerken, die nur Knoten eines Typs enthalten, muss die Regestendaten entsprechend erweitert werden. Dabei wird davon ausgegangen, dass zwei Personen, die gemeinsam in einem Regest genannt sind, etwas miteinander zu tun haben. Diese Verbindung wird durch eine **APPEARS\_WITH**-Kante dargestellt, da die gemeinsame Rolle im Regest nicht näher qualifiziert werden kann.

Für diesen Abschnitt werden als Datenbeispiel die Regesten Heinrichs IV. in einer Graphdatenbank verwendet. Die Erstellung der Graphdatenbank wird im Kapitel [Regestenmodellierung im Graphen](#) beschrieben. Auf dieser Datengrundlage werden dann noch zusätzliche Kantentypen erstellt. Mit dem folgenden cypher-Query werden zwischen Personeneinträgen des Registers, die gemeinsam in einem Regest genannt sind, **APPEARS\_WITH**-Kanten erstellt.

```
MATCH (n1:IndexPerson)-[r1:PERSON_IN]->(:Regesta)<-[r2:PERSON_IN]-(n2:IndexPerson)
WHERE id(n1) <> id(n2)
WITH n1, count(r1) AS c, n2
CREATE (n1)-[k:APPEARS_WITH]->(n2)
SET k.count = c;
```

## 11.3 Explorative Datenanalyse oder was ist in der Datenbank

Mit den in diesem Abschnitt vorstellten Queries lassen sich Graphen explorativ erfassen. Mit dem folgenden Query findet man alle im Graph vorkommenden Typen von Knoten und die jeweiligen Häufigkeiten.

```
CALL db.labels()
YIELD label
CALL apoc.cypher.run("MATCH (:"+label+")")
RETURN count(*) as count", null)
YIELD value
RETURN label, value.count as count
ORDER BY label;
```

Der nächste Query führt die gleiche Untersuchung für die in der Graphdatenbank vorhanden Kantentypen durch.

```
CALL db.relationshipTypes()
YIELD relationshipType
CALL apoc.cypher.run("MATCH ()-[: " + `relationshipType` + "]->()")
RETURN count(*) as count", null)
YIELD value
RETURN relationshipType, value.count AS count
ORDER BY relationshipType;
```

Mit diesen Queries lässt sich bei unbekannten Graphdatenbanken ein erster Überblick zur statistischen Verteilung von Knoten- und Kantentypen erstellen.

## 11.4 Zentralitätsalgorithmen in der historischen Netzwerkanalyse

Im folgenden Abschnitt werden verschiedene Zentralitätsalgorithmen zur Analyse der Personennetzwerke der Regesten Kaiser Heinrichs IV. verwendet. Im Zentrum steht hier zunächst die technische Anwendung. Die inhaltliche Analyse wird Gegenstand eines geplanten nächsten Kapitels sein.

### 11.4.1 PageRank

Der PageRank-Algorithmus bewertet und gewichtet eine Menge verknüpfter Knoten anhand ihrer Struktur.<sup>3</sup> Auf Grundlage der Verlinkungsstruktur wird dabei jedem Knoten ein Gewicht, der sog. PageRank zugeordnet.

---

<sup>3</sup>Zu PageRank vgl. <https://de.wikipedia.org/wiki/PageRank>.

```

CALL algo.pageRank.stream("IndexPerson", "APPEARS_WITH",
{iterations:20})
YIELD nodeId, score
MATCH (node) WHERE id(node) = nodeId
RETURN node.name1 AS Person,
apoc.math.round(score,3)
ORDER BY score DESC;

```

#### 11.4.2 Degree Centrality

```

MATCH (u:IndexPerson)
RETURN u.name1 AS name,
size((u)-[:APPEARS_WITH]->()) AS follows,
size((u)<-[:APPEARS_WITH]-()) AS followers
ORDER by followers DESC;

```

#### 11.4.3 Betweenes Centrality

```

// betweenness centrality
CALL algo.betweenness.stream("IndexPerson",
"APPEARS_WITH", {direction: "OUTGOING", iterations: 10}) YIELD nodeId, centrality
MATCH (p:IndexPerson) WHERE id(p) = nodeId
RETURN p.name1 AS Name, centrality
ORDER by centrality DESC;

```

#### 11.4.4 Closeness Centrality

```

// betweenness centrality
CALL algo.closeness.stream("IndexPerson", "APPEARS_WITH")
YIELD nodeId, centrality
MATCH (n:IndexPerson) WHERE id(n) = nodeId
RETURN n.name1 AS node, apoc.math.round(centrality,2)
ORDER BY centrality DESC
LIMIT 30;

```

### 11.5 Community Detection Algorithmen

#### 11.5.1 Strongly Connected Components

```
CALL algo.scc.stream("IndexPerson", "APPEARS_WITH")
```

```

YIELD nodeId, partition
MATCH (u:IndexPerson) WHERE id(u) = nodeId
RETURN u.name1 AS name, partition
ORDER BY partition DESC;

```

### 11.5.2 Weakly Connected Components

```

CALL algo.unionFind.stream("IndexPerson", "APPEARS_WITH")
YIELD nodeId, setId
MATCH (u:IndexPerson) WHERE id(u) = nodeId
RETURN u.name1 AS name, setId;

```

### 11.5.3 Label Propagation

```

CALL algo.labelPropagation.stream("IndexPerson",
"APPEARS_WITH", {direction: "OUTGOING", iterations: 10})
YIELD nodeId, label
MATCH (p:IndexPerson) WHERE id(p) = nodeId
RETURN p.name1 AS Name, label ORDER BY label DESC;

```

### 11.5.4 Louvain Modularity

```

CALL algo.louvain.stream("IndexPerson",
"APPEARS_WITH", {})
YIELD nodeId, community
MATCH (user:IndexPerson) WHERE id(user) = nodeId
RETURN user.name1 AS user, community
ORDER BY community;

```

### 11.5.5 Triangle count and Clustering Coefficient

```

CALL algo.triangle.stream("IndexPerson", "APPEARS_WITH")
YIELD nodeA, nodeB, nodeC
MATCH (a:IndexPerson) WHERE id(a) = nodeA
MATCH (b:IndexPerson) WHERE id(b) = nodeB
MATCH (c:IndexPerson) WHERE id(c) = nodeC
RETURN a.name1 AS nodeA, b.name1 AS nodeB, c.name1 AS node;

```

direkt aus dem Beispiel übernommen

```

CALL algo.triangleCount.stream('IndexPerson', 'APPEARS_WITH')
YIELD nodeId, triangles, coefficient

```

```

MATCH (p:IndexPerson) WHERE id(p) = nodeId
RETURN p.name1 AS name, triangles, coefficient
ORDER BY coefficient DESC;

```

nach der Anzahl der Dreiecksbeziehungen sortiert

```

CALL algo.triangleCount.stream('IndexPerson', 'APPEARS_WITH')
YIELD nodeId, triangles, coefficient
MATCH (p:IndexPerson) WHERE id(p) = nodeId
RETURN p.name1 AS name, triangles, coefficient
ORDER BY triangles DESC;

```

```

CALL algo.labelPropagation.stream("IndexPerson", "APPEARS_WITH", {direction: "OUTGOING"})
MATCH (p:IndexPerson) WHERE id(p) = nodeId
RETURN p.name1 AS Name, label ORDER BY label DESC;

```

```

CALL algo.triangle.stream("IndexPerson", "APPEARS_WITH")
YIELD nodeA, nodeB, nodeC
MATCH (a:IndexPerson) WHERE id(a) = nodeA
MATCH (b:IndexPerson) WHERE id(b) = nodeB
MATCH (c:IndexPerson) WHERE id(c) = nodeC
RETURN a.shortName AS nodeA, b.shortName AS nodeB, c.shortName AS node;

```

## 11.6 Zusammenfassung

In diesem Kapitel wurde die im Kapitel zur Graphmodellierung eingerichtete Graphdatenbank mit den Regesten Kaiser Heinrichs IV. für die Anwendung von Netzwerkanalyse-Algorithmen vorbereitet. Im zweiten Abschnitt wurden dann cypher-Queries für verschiedenen Netzwerkalgorithmen aufgelistet. In einem weiteren geplanten Kapitel werden die Ergebnisse dieser Algorithmen qualitativ ausgewertet.

## 12 Zusammenfassung

An Hand der verschiedenen Beispiel konnte gezeigt werden, dass Graphentechnologien hervorragend für die Modellierung, Speicherung und Analyse hochvernetzter Daten geeignet ist. Und die Beispiele konnten auch zeigen, dass die digitalen Geisteswissenschaften reich an hochvernetzten Daten sind. Gleichzeitig lassen sich mit dem einer Mind-Map sehr ähnlichen Modell Forschungsdaten und Forschungsfragestellungen tatsächlich in einer Weise modellieren, die dem menschlichen Denken sehr nahe kommt. Damit können Graphentechnologien gleichsam als Brücke zwischen den geisteswissenschaftlichen Fachdisziplinen und den informationstechnologischen Herausforderungen und Perspektiven des digitalen Zeitalters dienen.

So gelingt es in den digitalen Geisteswissenschaften mit dem Graphenmodell bei der Modellierung und Strukturierung von Forschungsdaten und Forschungsfragestellung die Kluft zwischen Informatiker und Geisteswissenschaftler zu schließen, da der Graph eine für beide Seiten verständliche Plattform bietet. Für den Informatiker ist er hinreichend genau und berechenbar und für den Geisteswissenschaftler wegen seiner Schema- und Hierarchiefreiheit ausreichend flexibel. Gerade diese Eigenschaften, mit denen sich die beiden zentralen Zweige der Digitalen Geisteswissenschaften vereinen lassen, machen Graphen zu einem Schlüsselkonzept der Geisteswissenschaften des 21. Jahrhunderts.

# 13 Anhang

In diesem Kapitel werden Tipps und Tricks rund um typische Herausforderungen bei der Verwendung von Graphdatenbanken in den digitalen Geisteswissenschaften vorgestellt. Die Hinweise stammen oft von meinem Kollegen Stefan Armbruster von neo4j, dem an dieser Stelle nochmal herzlich für seine Unterstützung gedankt sei.

## 13.1 cypher-Dokumentation

Die Dokumentation von cypher findet sich auf den Seiten von neo4j: <https://neo4j.com/docs/developer-manual/current/>

## 13.2 Analyse der Graphdaten

### 13.2.1 Welche und jeweils wieviele Knoten enthält die Datenbank

Mit dem folgenden Query werden alle Typen von Knoten und deren jeweilige Häufigkeit aufgelistet.

```
CALL db.labels()  
YIELD label  
CALL apoc.cypher.run("MATCH (:`"+label+"`)  
RETURN count(*) as count", null)  
YIELD value  
RETURN label, value.count as count  
ORDER BY label
```

### 13.2.2 Welche Verknüpfungen gibt es in der Datenbank und wie häufig sind sie

```
CALL db.relationshipTypes()  
YIELD relationshipType  
CALL apoc.cypher.run("MATCH ()-[: " + `relationshipType` + "]->()  
RETURN count(*) as count", null)  
YIELD value
```

```
RETURN relationshipType, value.count AS count
ORDER BY relationshipType
```

### 13.2.3 Welche Knoten haben keine Kanten

```
MATCH (n)
WHERE size((n)--())=0
RETURN labels(n), count(labels(n)) AS Anzahl ORDER BY Anzahl DESC;
```

## 13.3 Weitere Labels für einen Knoten

Gegeben sind Knoten vom Typ IndexEntry, die in der Property type noch näher spezifiziert sind (z.B. Ort, Person, Sache etc.). Mit dem folgenden Query wird der Wert der Property type als zusätzliches Label angelegt.

```
MATCH (e:IndexEntry)
WHERE e.type IS NOT NULL
WITH e, e.type AS label
CALL apoc.create.addLabels(id(e), [label]) YIELD node
RETURN node;
```

Die Namen der Labels können auch selbst bestimmt werden.

```
MATCH (e:IndexEntry)
WHERE e.type = 'person'
WITH e
CALL apoc.create.addLabels(id(e), ['IndexPerson']) YIELD node
RETURN node;
```

```
MATCH (e:IndexEntry)
WHERE e.type = 'ereignis'
WITH e
CALL apoc.create.addLabels(id(e), ['IndexEvent']) YIELD node
RETURN node;
```

```
MATCH (e:IndexEntry)
WHERE e.type = 'sache'
WITH e
CALL apoc.create.addLabels(id(e), ['IndexThing']) YIELD node
RETURN node;
```

```
MATCH (e:IndexEntry)
WHERE e.type = 'ort'
```

```
WITH e
CALL apoc.create.addLabels(id(e), ['IndexPlace']) YIELD node
RETURN node;
```

## 13.4 CSV-Feld enthält mehrere Werte

Beim Import von Daten im CSV-Format in die Graphdatenbank kann es vorkommen, dass in einem CSV-Feld mehrere Werte zusammen stehen. In diesem Abschnitt wird erklärt, wie man diese Werte auseinandernehmen, einzeln im Rahmen des Imports nutzen kann.

In der Regel ist es von Vorteil, zunächst das CSV-Feld als eine Property zu importieren und in einem zweiten Schritt auseinanderzunehmen.

Angenommen wir haben Personen importiert, die in der Property `abschluss` eine kommasseparierte Liste von verschiedenen beruflichen Abschlüssen haben, wie z.B. Lehre, BA-Abschluss, MA-Abschluss, Promotion.

In der Property `abschluss` steht zum Beispiel drin:

```
lic. theol., mag. art., dr. theol., bacc. art., bacc. bibl. theol.
```

Für die Aufteilung der Einzelwerte kann die `split`-Funktion verwendet werden, die einen String jeweils an einem anzugebenden Schlüsselzeichen (hier das Komma) auftrennt. Der Befehl hierzu sieht wie folgt auch:

```
MATCH (p:Person)
FOREACH ( j in split(p.abschluss, ", ") |
MERGE (t:Titel {name:j})
MERGE (t)-[:ABSCHLUSS]-(p)
);
```

Der Query nimmt die Liste von Abschlüssen jeweils beim Komma auseinander, erstellt mit dem `MERGE`-Befehl einen Knoten für den Abschluss (falls noch nicht vorhanden) und verlinkt diesen Knoten dann mit dem Personenknoten. Zu beachten ist, dass die im CSV-Feld gemeinsam genannten Begriffe konsistent benannt sein müssen.

## 13.5 Reguläre Ausdrücke

Mit Apoc ist es möglich, reguläre Ausdrücke zum Auffinden und Ändern von Property-Werten zu nutzen.

Mit dem folgenden Query werden in den Überlieferungsteilen der Regesten Kaiser Heinrichs IV. die Verlinkungen der Literatur rausgesucht und für jeden Link per `MERGE`

ein Knoten erzeugt. Anschließend werden die neu erstellen Knoten mit den jeweiligen Regesten über eine REFERENCES-Kante verbunden.

```
MATCH (reg:Regesta)
WHERE reg.archivalHistory CONTAINS "link"
UNWIND apoc.text.regexGroups(reg.archivalHistory, "<link (\\S+)>(\\S+)</link>") as link
MERGE (ref:Reference {url:link[1]}) ON CREATE SET ref.title=link[2]
MERGE (reg)-[:REFERENCES]->(ref);
```

## 13.6 Vorkommende Wörter in einer Textproperty zählen

Werden Texte in der Property source eines Knotens l gespeichert, kann man sich mit folgendem Query die Häufigkeit der einzelnen Wörter anzeigen lassen.

```
match (l:Letter)
return apoc.coll.frequencies(
  apoc.coll.flatten(
    collect(
      split(
        apoc.text.regreplace(l.source, "[^a-zA-Z0-9ÄÖÜäöüß ]", "")
      ),
      " ")
    )
  );
);
```

In der folgenden Fassung wird die Liste noch nach Häufigkeit sortiert.

```
match (l:Letter)
with apoc.coll.frequencies(
  apoc.coll.flatten(collect(
    split(
      apoc.text.regreplace(l.source, "[^a-zA-Z0-9ÄÖÜäöüß ]", "")
      , " ")
    )
  )
) as freq
unwind freq as x
return x.item, x.count order by x.count desc
```

## 13.7 MERGE schlägt fehl da eine Property NULL ist

Der MERGE-Befehl entspricht in der Syntax dem CREATE-Befehl, überprüft aber bei jedem Aufruf, ob der zu erstellende Knoten bereits in der Datenbank existiert. Bei dieser

Überprüfung werden alle Propertys des Knoten überprüft. Falls also ein vorhandener Knoten eine Property nicht enthält, wird ein weiterer Knoten erstellt. Umgekehrt endet der **MERGE**-Befehl mit einer Fehlermeldung, wenn eine der zu prüfenden Propertys NULL ist.

Gerade beim Import von CSV-Daten leistet der **MERGE**-Befehl in der Regel sehr gute Dienste, da man mit ihm bereits beim Import einer Tabelle weitere Knotentypen anlegen und verlinken kann. Oft kommt es aber vor, dass man sich nicht sicher ist, ob eine entsprechende Property in allen Fällen existiert. Hier bietet es sich an, vor dem **MERGE**-Befehl mit einer **WHERE**-Clause die Existenz der Property zu überprüfen.

Im folgenden Beispiel importierten wir Personen aus einer CSV-Liste, bei denen pro Person jeweils eine ID, ein Name und manchmal ein Herkunftsstadt angegeben ist. Im ersten Schritt werden im **CREATE**-Statement die Personen erstellt und auch der Herkunftsstadt als Property angelegt, der aber auch NULL sein kann.

```
LOAD CSV WITH HEADERS FROM "file:///import.csv" AS line
CREATE (p:Person {pid:line.ID_Person, name:line.Name, herkunft:line.Herkunft});
```

Im zweiten Schritt wird nun der **LOAD CSV**-Befehl nochmals ausgeführt und über die **WHERE**-Clause nur jene Fälle weiter bearbeitet, in denen die Property Herkunft nicht NULL ist. Nach der **WHERE**-Clause wird über den **MATCH**-Befehl zunächst der passende Personenknoten aufgerufen, anschließend per **MERGE**-Befehl der Ortsknoten erstellt (falls noch nicht vorhanden) und schließlich mit **MERGE** beide verknüpft.

```
LOAD CSV WITH HEADERS FROM "file:///import.csv" AS line
WHERE line.Herkunft IS NOT NULL
MATCH (p:Person {pid:line.ID_Person})
MERGE (o:Ort {ortsname:line.Herkunft})
MERGE (p)-[:HERKUNFT]->(o);
```

## 13.8 Knoten hat bestimmte Kante nicht

Am Beispiel der **Regesta-Imperii-Graphdatenbank** der Regesten Kaiser Friedrichs III. werden mit dem folgenden Cypher-Query alle Regestenknoten ausgegeben, die keine **PLACE\_OF\_ISSUE**-Kante zu einem Place-Knoten haben:

```
MATCH (reg:Regesta)
WHERE NOT
(reg)-[:PLACE_OF_ISSUE]->(:Place)
RETURN reg;
```

## 13.9 Häufigkeit von Wortketten

Am Beispiel des [DTA-Imports](#) von [Berg Ostasien](#) wird mit dem folgenden Query die Häufigkeit von Wortketten im Text ausgegeben:

```
MATCH p=(n1:Token)-[:NEXT_TOKEN]->(n2:Token)-[:NEXT_TOKEN]->(n3:Token)
WITH n1.text as text1, n2.text as text2, n3.text as text3, count(*) as count
WHERE count > 1 // evtl höherer Wert hier
RETURN text1, text2, text3, count ORDER BY count DESC LIMIT 10
```

## 13.10 Der WITH-Befehl

Da cypher eine deklarative und keine imperative Sprache ist gibt es bei der Formulierung der Querys Einschränkungen.<sup>1</sup> Hier hilft oft der WITH-Befehl weiter, mit dem sich die o.a. beiden Befehle auch in einem Query vereinen lassen:

```
LOAD CSV WITH HEADERS FROM "file:///import.csv" AS line
CREATE (p:Person {pid:line.ID_Person, name:line.Name, herkunft:line.Herkunft})
WITH line, p
WHERE line.Herkunft IS NOT NULL
MERGE (o:Ort {ortsname:line.Herkunft})
MERGE (p)-[:HERKUNFT]->(o);
```

Der `LOAD CSV`-Befehl lädt die CSV-Datei und gibt sie zeilenweise an den `CREATE`-Befehl weiter. Dieser erstellt den Personenknoten. Der folgende `WITH`-Befehl stellt quasi alles wieder auf Anfang und gibt an die nach ihm kommenden Befehle nur die Variablen `line` und `p` weiter.

## 13.11 Die Apoc-Bibliothek

Die Funktionalitäten sind bei neo4j in verschiedene Bereiche aufgeteilt. Die Datenbank selbst bringt Grundfunktionalitäten mit. Um Industriestandards zu genügen haben diese Funktionen umfangreiche Tests und Prüfungen durchlaufen. Weiteregehende Funktionen sind in die sogenannte [apoc-Bibliothek](#) ausgelagert, die zusätzlich installiert werden muss. Diese sogenannten *user defined procedures* sind benutzerdefinierte Implementierungen bestimmter Funktionen, die in cypher selbst nicht so leicht ausgedrückt werden können. Diese Prozeduren sind in Java implementiert und können einfach in Ihre Neo4j-Instanz implementiert und dann direkt von Cypher aus aufgerufen werden.<sup>2</sup>

---

<sup>1</sup>Hierzu vgl. [https://de.wikipedia.org/wiki/Deklarative\\_Programmierung](https://de.wikipedia.org/wiki/Deklarative_Programmierung) zuletzt abgerufen am 12.6.2018.

<sup>2</sup>Vgl. <https://guides.neo4j.com/apoc> (zuletzt aufgerufen am 11.04.2018).

Die APOC-Bibliothek besteht aus vielen Prozeduren, die bei verschiedenen Aufgaben in Bereichen wie Datenintegration, Graphenalgorithmen oder Datenkonvertierung helfen.

### 13.11.1 Installation in neo4j

Die Apoc-Bibliothek lässt sich unter <http://github.com/neo4j-contrib/neo4j-apoc-procedures/releases/%7Bapoc-release%7D> herunterladen und muss in den plugin-Ordner der neo4j-Datenbank kopiert werden.

### 13.11.2 Installation unter neo4j-Desktop

In *neo4j-Desktop* kann die Apoc-Bibliothek jeweils pro Datenbank im Management-Bereich über den Reiter **plugins** per Mausklick installiert werden.

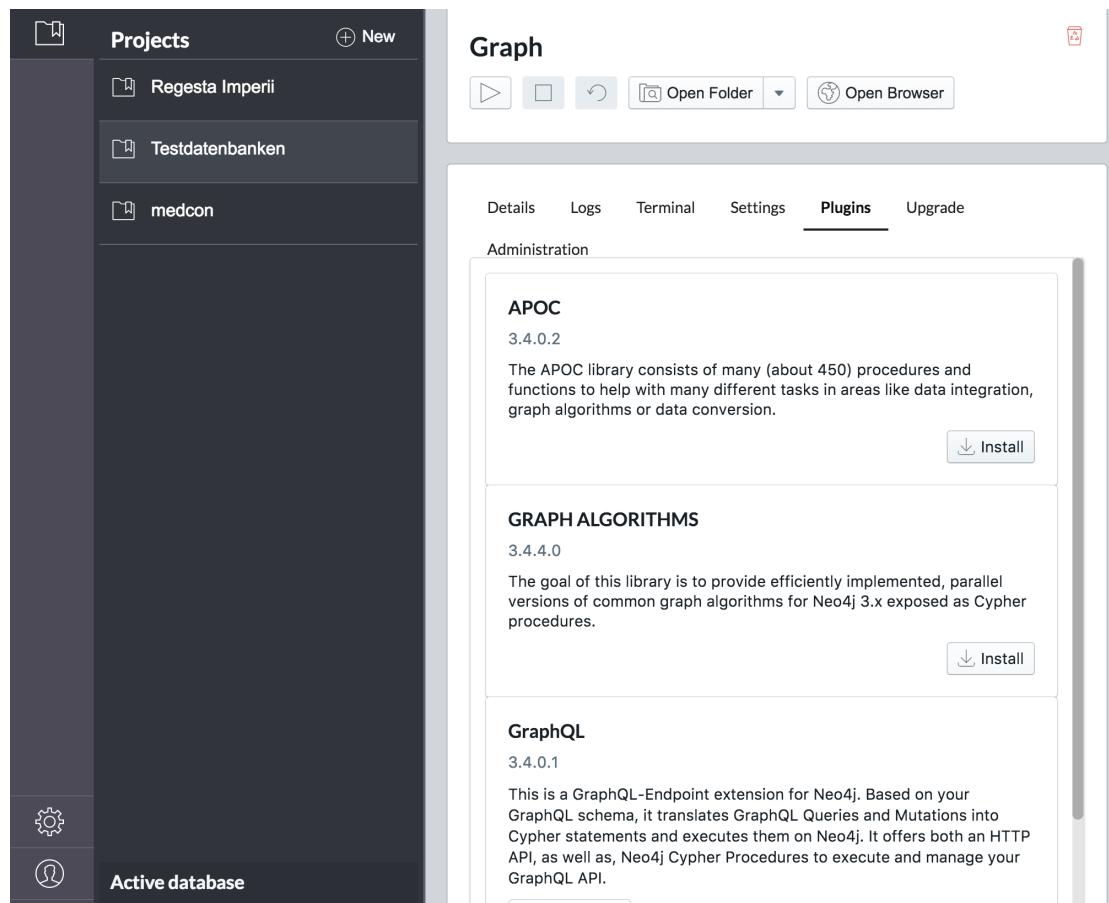


Abbildung 13.1: Installation der apoc-Bibliothek in neo4j-Desktop

### 13.11.3 Liste aller Funktionen

Nach dem Neustart der Datenbank stehen die zusätzlichen Funktionen zur Verfügung. Mit folgendem Befehl kann überprüft werden, ob die Apoc-Bibliotheken installiert sind:

```
CALL dbms.functions()
```

Wenn eine Liste mit Funktionen ausgegeben wird, war die Installation erfolgreich. Falls nicht, sollte die Datenbank nochmals neu gestartet werden.

### 13.11.4 Dokumentation aller Funktionen

In der [Dokumentation](#) der apoc-Bibliothek sind die einzelnen Funktionen genauer beschrieben.

## 13.12 apoc.xml.import

Mit dem Befehl apoc.xml.import ist es möglich, einen xml-Baum 1:1 in die Graphdatenbank einzuspielen. Die [Dokumentation](#) findet sich [hier](#).

Beispielbefehl: `call apoc.xml.import("URL", {createNextWordRelationships: true}) yield node return node;`

Kantentyp	Beschreibung
:IS_CHILD	Verweis auf eingeschachteltes Xml-Element
:FIRST_CHILD	Verweis auf das erste untergeordnete Element
:NEXT_SIBLING	Verweis auf das nächste Xml-Element auf der gleichen Ebene

Kantentyp	Beschreibung
:NEXT	Erzeugt eine lineare Kette durch das gesamte XML-Dokument und gibt so die Serialität des XMLs wieder
:NEXT_WORD	Verbindet Wortknoten zu einer Kette von Wortknoten. Wird nur erzeugt, wenn createNextWordRelationships:true gesetzt wird

### 13.13 (apoc.load.json)

(Dieser Abschnitt befindet sich gerade in Bearbeitung)

```

create constraint on (p:Person) assert p.id is unique;
create constraint on (p:AristWork) assert p.id is unique;
create constraint on (p:Manuscript) assert p.id is unique;

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind keys(value.persons) as personId
merge (personNode:Person{id:personId})
set personNode = value.persons[personId];

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind keys(value.aristWorks) as aristWorksId
merge (aristWorksNode:AristWork{id:aristWorksId})
set aristWorksNode = value.aristWorks[aristWorksId];

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind keys(value.mss) as msId
merge (msNode:Manuscript{id:msId})

```

```

set msNode = value.mss[msId];

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind value.`ms-person-rel` as rel
match (start:Person{id:rel.person})
match (end:Manuscript{id:rel.ms})
with start, end, rel
call apoc.merge.relationship(start, toUpper(rel.rel), {}, {}, end) yield rel as dummy
return count(*);

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind value.`ms-ms-rel` as rel
merge (start:Manuscript{id:rel.ms})
merge (end:Manuscript{id:rel.`other-ms`})
with start, end, rel
call apoc.merge.relationship(start, toUpper(rel.rel), {}, {}, end) yield rel as dummy
return count(*);

call apoc.load.json("file:///var/lib/neo4j/import/cagb-graph-test-v1.json") yield value
unwind value.`ms-aristWork-rel` as rel
match (start:Manuscript{id:rel.ms})
match (end:AristWork{id:rel.aristWork})
with start, end, rel
call apoc.merge.relationship(start, toUpper(rel.rel), {}, {}, end) yield rel as dummy
return count(*);

json-example
{
  "mspersonrel": [
    {
      "ms": "69686",
      "person": "d3f1",
      "rel": "author-contained"
    },
    {
      "ms": "69686",
      "person": "p3366450e-0387-43d4-9f04-7f0f1c08dff8",
      "rel": "scribe"
    },
    {
      "ms": "69686",
      "person": "p8c827441-77b4-4e12-8209-7ce8f06060f1",
      "rel": "scribe"
    }
  ],
  "persons": {
    "d19f17": {

```

```
    "label": "Castro, Juan Pàez de",
    "id": "d19f17"
},
"d10f30": {
    "label": "Augustinus (Aurelius Augustinus)",
    "id": "d10f30"
},
"d22f20": {
    "label": "Manouel\n Chrysoloras",
    "id": "d22f20"
}
},
"aristWorks": {
    "EE": {
        "label": "Ethica ad Eudemum (EE)",
        "id": "EE"
    },
    "Parva-Naturalia": {
        "label": "Parva naturalia (Parva Naturalia)",
        "id": "Parva-Naturalia"
    },
    "Hist.-An.": {
        "label": "Historia animalium (Hist. An.)",
        "id": "Hist.-An."
    }
}
```