Bachelorarbeit

Einsatz von Graphdatenbanken zur Repräsentation kultureller Metadaten

Tim Jödden

Berlin, 22. April 2013

Fachbereich:	Informatik
Matrikelnummer:	906033
Erstgutachter:	Prof. Dr. Oliver Vornberger
Zweitgutachter:	Dr. Kai Stalmann

Inhaltsverzeichnis

1. Einleitung

Durch stetigen technischen Fortschritt werden die altbewährten Methoden der Datenhaltung mehr und mehr in den Hintergrund gedrängt. Der exponentiell wachsenden Masse an Daten steht mit Relationalen Datenbanksystemen eine Technologie gegenüber, die für eine Verarbeitung großer verknüpfter Datenmengen nicht mehr ausreichend ist, da die Veränderung in der Nutzung der Daten durch die tabellenbasierte Herangehensweise der Datenhaltung beschränkt wird.

Neue Technologien aus dem NoSQL[1]-Sektor sollen einen Lösungsansatz für diesen Fehlstand liefern. Durch das Abweichen von den üblichen festen Tabellenstrukturen und ihren rechenintensiven JOIN-Operationen, die Lastverteilung der Daten auf eine größere Anzahl von Systemen und eine neue Vorgehensweise bei der Strukturierung der Daten wie z.B. durch Graphdatenbanken werden neue Wege erforscht, wie mit einer großen Anzahl von Daten und der damit einhergehenden Menge an Schreib- und Leseanfragen umgegangen werden soll.

Im kulturellen Bereich sind insbesondere eine große Anzahl an Verknüpfungen zwischen Datensätzen ein gravierendes Problem. Die Metadaten eines kulturellen Gutes sind häufig mit anderen Objekten verbunden. Beispielsweise kann der Autor eines Buches noch ein zweites Buch verfasst haben, so dass die beiden Werke über den Autor miteinander in Verbindung stehen: ein Netzwerk entsteht. In der Deutschen Digitalen Bibliothek (DDB) [SKL11, S. 70], die am IAIS Fraunhofer[2] im Auftrag von Bund und Ländern seit 2010 entwickelt wird, wird eine Repräsentation einer derartigen Struktur implementiert. Ziel der DDB[3] ist es, kulturelle Güter und wissenschaftliche Forschungsergebnisse von über 30.000 kulturellen und wissenschaftlichen Einrichtungen in einem großen, gemeinsamen Portal der Öffentlichkeit zugänglich zu machen und diese Werke miteinander in Verbindung zu bringen, um so einen neuartigen Ansatz der Suchbarkeit zu ermöglichen. Ein Kernelement ist hierbei die Graphstruktur, die die Metadaten miteinander verknüpft. Die aufkommende Datenmenge überschreitet hierbei jedoch die technischen Möglichkeiten herkömmlicher Systeme, so dass mit Springdata Neo4j[4] zukünftig eine passendere Struktur verwendet werden soll.

[1] (engl.:"Not only SQL") Oberbegriff für nichtrelationale Datenbanken
[2] Fraunhofer Institut für Intelligente Analyse und Informationssysteme
[3] Deutsche Digitale Bibliothek
[4] Hochperformante Graphdatenbank in Java

Die vorliegende Arbeit soll darstellen, dass es möglich ist, Kulturdaten strukturiert zu speichern und die Suchbarkeit effizient zu gestalten. Dazu werden einige Fallbeispiele mit prototypischen Problemstellungen erklärt, deren Lösung zum Ergebnis der Recherche beitragen.

Aufbau der Arbeit

Die Ausarbeitung ist in 3 thematische Hauptabschnitte gegliedert.

Zunächst soll in Abschnitt 2 die Datenhaltung in Graphstrukturen, sowie die entsprechenden Algorithmen und Datenstrukturen allgemein dargestellt werden.

Im weiteren Verlauf von Abschnitt 3 dient die Deutsche Digitale Bibliothek als Anwendungsbeispiel für die Erfassung von Metadaten und die Verarbeitung zu einem Netzwerk. Analog zum in der DDB verwendeten CIDOC-CRM[1]-Modell zur Repräsentation von Kulturdaten wird das Konzept der Graphdatenbanken in einen kulturellen Kontext überführt und weitere Möglichkeiten zur Repräsentation des Modells besprochen, wie z.B. die derzeitige Implementation in Apache Solr und dessen Problemstellen.

In Abschnitt 4 soll eine Beispielimplementation das Konzept in einem einfachen Prototypen demonstrieren und eine Leistungsevaluation über die Möglichkeiten des Einsatzes von Neo4j liefern. Ein Ausblick und Fazit findet sich am Schluß der Ausarbeitung.

[1] CIDOC Conceptual Reference Model

2. Graphdatenbanken

Entwickelt mit graphtheoretischem Hintergrund sind Graphdatenbanken für Graphstrukturen optimierte Datenbanken, die sich besonders dazu eignen, Topologieanalysen auf den verbundenen Datensätzen durchzuführen. Dieses Kapitel soll das Konzept der Graphdatenbanken im Allgemeinen darstellen. Dazu wird zunächst die *Historische Entwicklung* (Abschnitt 2.1) von Datenbanken bis hin zu Graphdatenbanken dargestellt. In dem Abschnitt *Problemstellungen im Graphenbereich* (Abschnitt 2.5) werden einige typische Anwendungsfälle für Graphdatenbanken und ihre Problemstellungen erklärt. Einige der notwendigen Algorithmen sind in *Algorithmen und Datenstrukturen* (Abschnitt 2.3) beschrieben.

2.1. Historische Entwicklung

Anfang der 1960er Jahre kam nicht zuletzt durch die amerikanische Raumfahrtbehörde NASA die Frage auf, wie Computer zur Organisation von Daten genutzt werden können. Einfache Systeme bildeten bis dato lediglich Datenlisten ab, die keine optimalen Zugriffe ermöglichten. Durch Indizes und andere optimierte Algorithmen wurden diese Listen zwar erweitert, so dass die Datenabfrage kürzere Zugriffszeiten hatte, eine Abfrage verknüpfter Daten oder auch ein Vergleich der Daten war dennoch nicht effizient möglich. Verschiedene Unternehmen, u.a. IBM waren seitdem maßgeblich an der Erforschung einer Lösung zu dieser Fragestellung beteiligt, wobei der Schwerpunkt der Forschung auf der Verknüpfung der Daten lag, da die Verwaltung mit Hilfe von Registern einen großen personellen und logistischen Aufwand bedeutete. Seitdem wurden diverse Konzepte ausgearbeitet, die Datenhaltung effizienter zu gestalten.

Relationale Datenbanken

Im Juni 1970 legte Ted Codd, ein IBM Mitarbeiter, mit seinem ersten Artikel „A Relational Model of Data for Large Shared Data Banks“ [Cod70] über das Konzept der „Relationalen Datenbanken“ einen Grundstein für die heutigen Datenbanksysteme [WC05, Kapitel 1.1]. Die in dem Artikel beschrieben „Codd'schen Regeln“ [vgl. Sin11, Kapitel 5.2] zur Beschreibung des Konzeptes von Relationalen Datenbanken führten nach einigen

Dispositionen mit Konkurrenten schließlich zur Entwicklung der Structured Query Language (SQL), die seitdem als abstrakte Abfragesprache für alle relationalen Datenbanken in eigenen Dialekten implementiert wurde.

Selbst aktuelle Datenbanksysteme haben sich seit den ersten Entwicklungen mit SQL Unterstützung, wie die „Oracle Version 2“ in den 1980er Jahren im Kern kaum verändert. Diese Datenbankklasse hat sich durch eine ausgewogene Mischung aus Einfachheit, Robustheit, Flexibilität, Performanz, Skalierbarkeit[1] und Kompatibilität der Datenverwaltung bewährt. Besonders beim Speichern von strukturierten Massendaten in einzelnen ACID[2]-konformen Transaktionen, die Jim Gray erstmals 1986 definierte [GR93], werden Massendaten zuverlässig abgelegt und abgerufen. Die durch das große Einsatzfeld, die tabellenbasierte Funktionsweise und ACID-Konformität gewonnene Flexibilität und Sicherheit bringt jedoch gleichzeitig einen Verlust an Geschwindigkeit und Einfachheit mit sich, abhängig von der Anzahl der Datensätze. Standen diese Maßgaben bei der Entwicklung des relationalen Datenbankkonzeptes noch nicht im Vordergrund, erlangen sie derzeit aufgrund der Veränderung in der Nutzung der Daten oft eine höhere Priorität.

Datenbanken heute

Mit dem Internet wächst auch die Masse an Daten exponentiell, dabei sind die verfügbaren Hardwareressourcen jedoch beschränkt. Sowohl soziale Netzwerke und Mikroblogs wie Twitter und Facebook, als auch Suchmaschinen wie Google übertreffen sich regelmäßig mit Meldungen über das Überschreiten von Milliardengrenzen an Zugriffen. Die Vernetzung aller nur denkbaren Daten steht mit dem Ausbau der Internetinfrastruktur, der wachsenden Effizienz in der Datenerfassung und der immer besseren Verfügbarkeit der Systeme im Vordergrund: Eine Anforderung, für die relationale Datenbanken zu Beginn der 80er Jahre nicht ausgelegt waren und der sie bis heute nicht gewachsen sind.

Angepasste und für diese Problemstellungen optimierte Lösungen werden seitdem im „NoSQL“-Umfeld veröffentlicht. Diese Klasse von Datenbanken ist dadurch geprägt, dass sie von der üblichen Datenhaltung in Spalten und Tabellen Abstand nimmt und

[1] Skalierbarkeit ist die Fähigkeit eines Systems, Netzwerks oder Prozesses, eine wachsende Anzahl an Aufgaben durch Wachsen des Systems zu bewältigen

[2] (engl.: atomicity, consistency, isolation, durability) Voraussetzungen für die Verlässlichkeit von Datenbanksystemen

stattdessen eine neue Klasse von Datebanken eröffnet. Der Begriff NoSQL ist daher ein Oberbegriff für alle nicht-relationalen Datenbankkonzepte [PB13]. Statt durch die Abfragesprache SQL geschieht die Kommunikation mit diesen Datenbanktypen oft in nativen programmiersprachenabhängigen APIs oder durch neue Webservices wie REST. Die einfachste Art der Datenhaltung sind „Key-Value Stores“, also indexierte Listen von Einträgen. Diese sind für die Problemstellung geeignet, eine große Masse von Daten schnell verarbeiten zu müssen. Die einzelnen Einträge sind dabei jedoch nicht miteinander verknüpft. Technisch gesehen sind Key-Value Stores also im eigentlichen Sinne ein Rückschritt zu der Basis der relationalen Datenbanken, in der Listen zur Speicherung von Daten verwendet werden. Spätestens mit der Vorstellung des Artikels „MapReduce: simplified data processing on large clusters“ [DG08], der einen Algorithmus zum verteilten Berechnen von Massendaten beschreibt, wird klar, dass tabellenbasierte Datenbanken nicht ausreichen, um große Datenmengen zu verarbeiten: Google beschreibt in diesem Artikel im Jahr 2008 bereits einen Datenfluss von täglich 24 Petabyte. Suchanfragen per Google sind allerdings bis dato nur minimal miteinander vernetzt. Das Auffinden eines Elementes wird vielmehr durch den „Pagerank[1]“, einen proprietären, durch die Nutzer selbst gewichteten Volltextindex, ermöglicht.

Soziale Netzwerke wie Facebook veröffentlichten im Oktober 2012 Statistiken, aus denen ersichtlich ist, dass etwa 1 Milliarde Nutzer das Portal monatlich nutzen [Fac12]. Im Unterschied zu Datenquellen wie Twitter, die lediglich Listen von Objekten verarbeiten, beinhalten soziale Netze eine weitere Komplexitätsklasse, nämlich eine bidirektionale Verknüpfung der Datensätze und Traversierung[2] eines Clusters. In der Theorie kann dieses Netzwerk von Einträgen auch über relationale Datenbanken durch Fremdschlüsselrelationen abgebildet werden, in der Praxis sind die dafür nötigen Zugriffe jedoch zu zahlreich, um mitsamt der Transaktionenen über die in relationalen Datenbanken üblichen Tabellenvergleiche bearbeitet zu werden. Demselben Problem stehen auch die Key-Value Speicher gegenüber: Jede Kante zwischen zwei Datensätzen muss explizit durch einen eigenen Eintrag abgebildet werden und bedeutet, dass beim Zugriff dementsprechend eine Anfrage an den Index gestellt wird. Eine Traversierung über mehrere Kanten ist folglich mit einem Vielfachen des eigentlichen Aufwandes verbunden.

[1] Maß zur Gewichtung der Güte von Dokumenten anhand der Anzahl ihrer Verlinkungen
[2] Folgen eines Pfades zwischen zwei Knoten

Graphdatenbanken

Der neuen Anforderung an die Datenverarbeitung folgend, wurden *Graphdatenbanken* entwickelt, die einen Graphen in seiner eigentlichen Form abbilden und mit den für die Verarbeitung nötigen Algorithmen versorgen. Die dazu benötigten theoretischen Modelle und Algorithmen sind durch die Graphtheorie, einem Gebiet der Mathematik, bereits seit geraumer Zeit erforscht und mussten lediglich algorithmisch abgebildet werden. Graphdatenbanken konnten noch während der Entwicklungsphase schnell Erfolge verzeichnen, da sie besonders effizient in der Verarbeitung von großen Mengen verknüpfter Daten arbeiten können. Obwohl es bereits stabile Graphdatenbanken wie „Neo4j“ und „Sones“ im Produktiveinsatz gibt, stehen die Entwicklung und die Erforschung der Einsatzgebiete allerdings noch am Anfang, denn auch die peripheren Entwicklungen, wie beispielsweise eine einheitliche abstrakte Abfragesprache, müssen sich etablieren. Stetige Weiterentwicklungen in der Forschung der Algorithmen ermöglichen zukünftig ausgeklügeltere Abfragen und Techniken, um die Daten effizienter abzurufen und Korrelationen zwischen Daten zu erkennen [TD11].

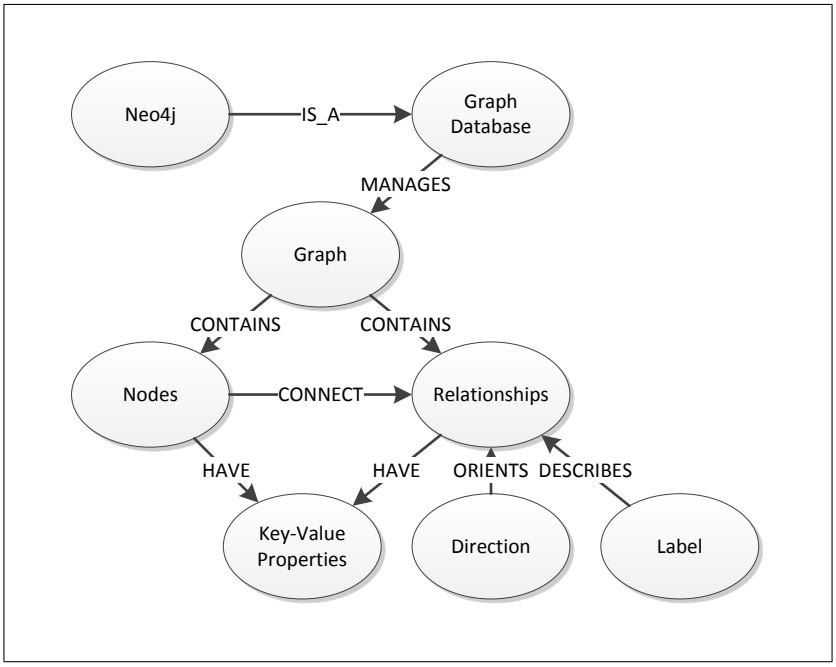


Abbildung 1: Neo4j Overview [PB13, S. 102]

2.2. Graphen

Ein Graph beschreibt eine mathematische Struktur, die durch *Kanten*[1] zwischen *Knoten*[2] gebildet wird, so dass eine verbundene Struktur entsteht. In der Graphtheorie lässt sich die abstrakte Struktur mathematisch als Tupel $G = (V, E)$ darstellen, wobei $V \neq \emptyset$ eine endliche Menge von Knoten und E eine Menge von Kanten ist. Eine Kante $e = \{v_1, v_2\}$ besteht dabei aus einer mindestens zweielementigen Teilmenge[3] von Knoten $v_1, v_2 \in V$. Dabei heißen v_1 und v_2 *adjazent* (benachbart) und alle Knoten, die zu v_1 benachbart sind heißen *Adjazenten* oder Nachbarn. Ein Knoten ohne Adjazenten heißt *isolierter Knoten*. Ein Graph mit ausschließlich isolierten Knoten heißt *Nullgraph*[4].

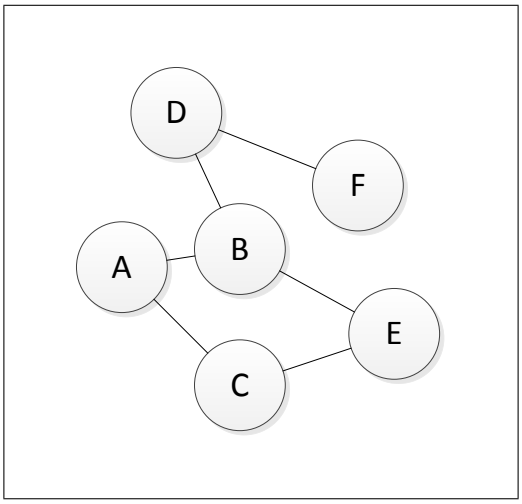


Abbildung 2: einfacher Graph

Typische mathematische Darstellungsformen von Graphen sind neben der bildlichen Darstellung Adjazenzmatrizen, Adjazenzlisten, Inzidenzmatrizen und Laplacematrizen. Abbildung 2 beschreibt einen einfachen Graphen als Ausgangsbasis. Weitere Methoden, wie z.B. Kompressionsalgorithmen optimieren die Darstellungsformen in den Implementierungen.

Ein Pfad[5] ist eine zusammenhängende Struktur mehrerer Kanten und Knoten, so dass diese durchgehend traversiert werden können. Die Pfadlänge[6] gibt eine Zahl von traver-

[1] synonym: Verknüpfungen, Relationen; engl.: node
[2] synonym: Objekte, Entitäten; engl.: edge
[3] Ausnahme: Hypergraphen, siehe Abbildung 7e
[4] frei nach Vorlesungsskript *Graphalgorithmen* von Sigrid Knust
[5] traversierbare, zusammenhängende Struktur mehrerer Kanten und Knoten
[6] Anzahl der traversierten Kanten zwischen zwei definierten Knoten

sierten Verbindungen an, die zwischen zwei Knoten liegen.

Cluster[1] sind einzelne Häufigkeitspunkte von Gruppen zugehöriger Knoten, also Punkte mit hoher Zentralität, die ausserhalb einer unbestimmt großen Gruppe von Pfaden nur wenige Verbindungen zu anderen Kanten haben. Entsprechend weisen diese Knoten eine geringe Pfadlänge zueinander auf. Dieses Merkmal ist im Speziellen wichtig, wenn beispielsweise eine Graphdatenbank über mehrere Server verteilt werden soll, da bei einer Verteilung von Graphen möglichst wenige Verbindungen zwischen den Clustern existieren sollten um den Datenaustausch über langsame Datenverbindungen möglichst gering zu halten (siehe auch Abschnitt 2.5 zu Replikationsmethoden). Ratsam wäre also für eine Verteilung eine Trennung zwischen Clustern mit wenigen Verbindungen.

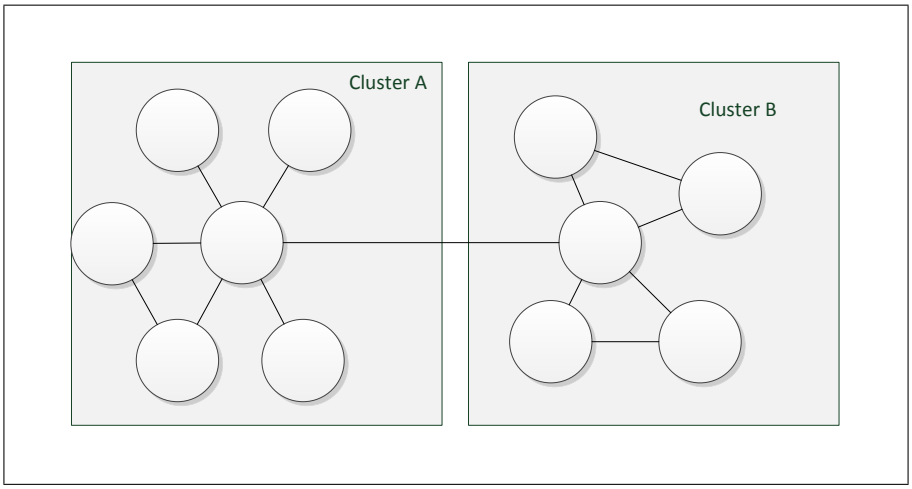


Abbildung 3: Clusterbildung in Graphen

Adjazenzmatrix

Eine Adjazenzmatrix ist eine Darstellungsform eines Graphen mit $n \times n$ Werten, wobei $n \in |V|$ ist. Die Indizes der Matrix beschreiben die jeweiligen Knoten des Graphen und die Werte, die angeben, ob eine Verknüpfung durch eine Kante e zwischen den Kanten besteht. Bei symmetrischen Graphen[2] ist die Matrix auf der Diagonalen gespiegelt. Gewichtete Graphen werden mit $n \in \mathbb{Z}$ dargestellt, $n \in \{0, 1\}$ wird dagegen für ungewichtete Graphen verwendet.

[1] Gruppe zugehöriger Knoten mit hoher Zentralität untereinander und wenigen Verbindungen zu anderen Knoten

[2] Graphen mit ausschließlich ungerichteten Kanten

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Abbildung 4: Adjazenzmatrix

Gerichtete Graphen (Abbildung 7c) können durch eine diagonal asymmetrische Matrix dargestellt werden. Bei gewichteten Graphen (Abbildung 7c) wird statt den binären Werten $0, 1$ ein anderer Zahlenraum eingesetzt, wie z.B. $\mathbb{R}^+$. Eine Kante von einem Knoten zu sich selbst wird auch *Schleife* genannt und lässt sich in der Adjazenzmatrix durch einen Wert auf der Diagonalen darstellen.

Adjazenzmatrizen sind durch ihre direkte Schreibweise algorithmisch effizient beim Hinzufügen und Entfernen von Verknüpfungen oder bei der Prüfung der Existenz einer Kante zwischen zwei Knoten. Unabhängig von der Anzahl der Kanten beträgt die Größe einer Adjazenzmatrix $|E|^2$. Algorithmisch ineffizient ist das Hinzufügen von Kanten, da in diesem Fall eine neue Dimension zur Matrix angefügt werden muss. Die Suche nach Adjazenzen ist durch linearen Zeitaufwand abhängig von der Anzahl der Kanten im Graph. [SP12, Seite 2]

Adjazenzliste

Adjazenzlisten sind eine Menge von direkten Abbildungen von Knoten eines Graphen (siehe Abbildung 5). Dabei wird jedem Knoten eine Menge von Knoten zugewiesen, zu denen er adjazent ist. Diese komprimierte Darstellungsform ist besonders effizient, wenn Nachbarn zu einem Knoten hinzugefügt werden. Ist die Menge der Knoten nicht indexiert, ist es sehr aufwändig, die Nachbarn eines Knoten zu identifizieren. Im Gegenzug ist bei der Indexierung der Knoten das Hinzufügen von logarithmischer Komplexität. [SP12, Seite 4]

Inzidenzmatrix

Eine Inzidenzmatrix ist eine bidirektionale boolesche $n \times e$ Matrix mit $n = |V|$ und $e = |E|$, wobei die Spalten die Knoten repräsentieren und die Zeilen die Kanten.

$$
\begin{aligned}
A &\rightarrow \{B, C\} \\
B &\rightarrow \{A, E\} \\
C &\rightarrow \{A, E\} \\
D &\rightarrow \{B, F\} \\
E &\rightarrow \{B, C\} \\
F &\rightarrow \{D\}
\end{aligned}
$$

Abbildung 5: Adjazenzliste

Jedes Auftreffen einer Kante auf einen Knoten wird dann durch eine entsprechende Stelle in der Matrix markiert. Abbildung 6 stellt ein Beispiel für eine Inzidenzmatrix dar.

$$
I = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

Abbildung 6: Inzidenzmatrix

In einfachen Graphen werden die Kanten durch spaltenweise jeweils zwei positive Werte dargestellt. Beispielsweise kann die bidirektionale Kante zwischen Knoten C und E durch Setzen der Werte $I_{5,3}$ und $I_{5,5}$ beschrieben werden. Eine Schleife kann in der Matrix durch eine Spalte mit einem einzigen Wert $x \neq 0$ dargestellt werden.

Inzidenzmatrizen eignen sich besonders für die Darstellung von *Hypergraphen* (siehe 7e), in denen eine Kante mehrere Knoten verbindet, da sie spaltenweise mehrere Werte positiv deklarieren kann. Unidirektionale Kanten können dagegen nicht dargestellt werden.

Eine Inzidenzmatrix ist aufgrund des Speicherbedarfs von $n*e$ Bits im Allgemeinen nicht effizient für Graphen mit vielen Kanten, da dies zu einem hohen Speicherverbrauch führt. [SP12, Seite 5]

	einfach	gerichtet	gewichtet	Multigraphen	Hypergraphen
Adjazenzmatrix	ja	ja	ja	nein	nein
Adjazenzliste	ja	nein	nein	ja	nein
Inzidenzmatrix	ja	nein	ja	ja	ja

Tabelle 1: Mathematische Darstellbarkeit der Graphklassen [RN04]

2.3. Algorithmen und Datenstrukturen

2.3.1. Klassifikation von Graphen

Die Topologie von Graphen kann durch verschiedene Klassen kategorisiert werden. Die Klassifikation richtet sich dabei hauptsächlich nach Art der Verknüpfungen zwischen den Kanten. Einige Beispiele für Klassen von Graphen sind im Folgenden dargestellt.

Einfache Graphen

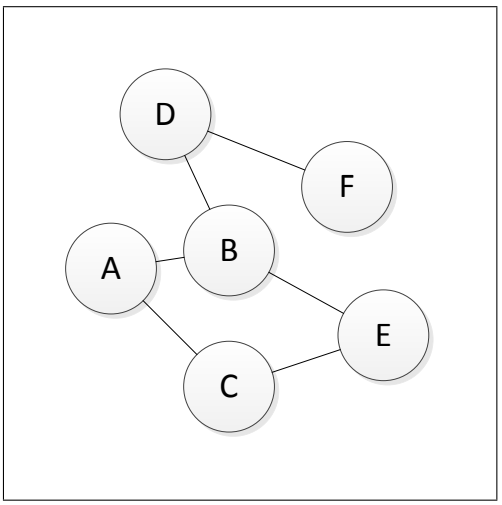
Ein einfacher Graph beschreibt Knoten, die durch Kanten verbunden sind. Die Kanten sind hierbei ungerichtet, also bidirektional traversierbar (siehe Abbildung 7a).

Gewichtete Graphen

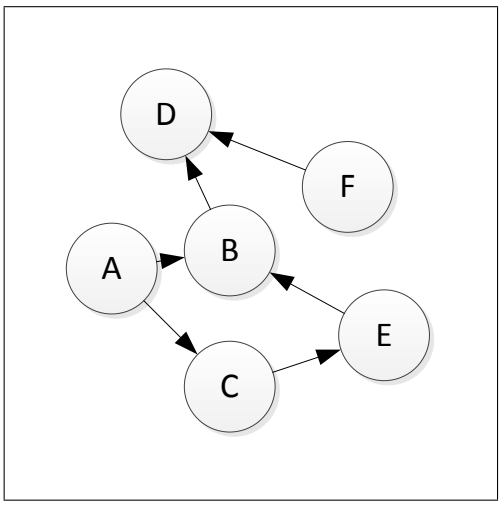
Die Haupteigenschaft gewichteter Graphen ist, dass den Kanten ein Zahlenwert zugewiesen wird (vgl. Abbildung 7c). Dieser Wert kann abhängig vom angewendeten Algorithmus Präferenzen für die Traversierung dieser Kante bedeuten. Beispielsweise würde ein Algorithmus für die Traversierung des Graphen von Knoten `C` zu Knoten `D` in Abbildung 7c mit der Präferenz „niedrigste Summe der Kantenwerte“ den Pfad `<C, A, B, D>` dem Pfad `<C, E, B, D>` vorziehen, da die Summe der Werte der Kanten niedriger wäre.

Multigraphen

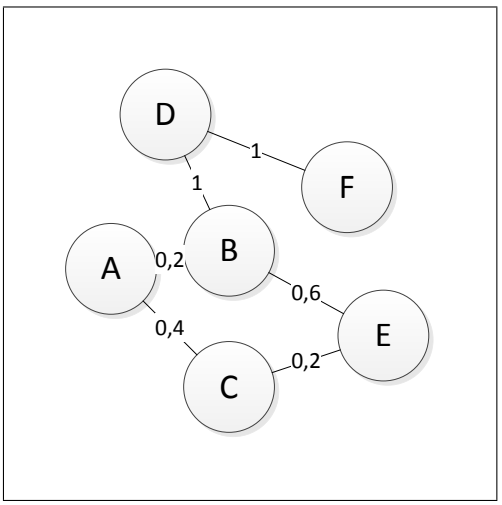
Multigraphen sind Graphen mit Mehrfachverbindungen zwischen einzelnen Knoten. Die Kanten können demnach auf verschiedenen Pfaden über dieselben Kanten traversiert



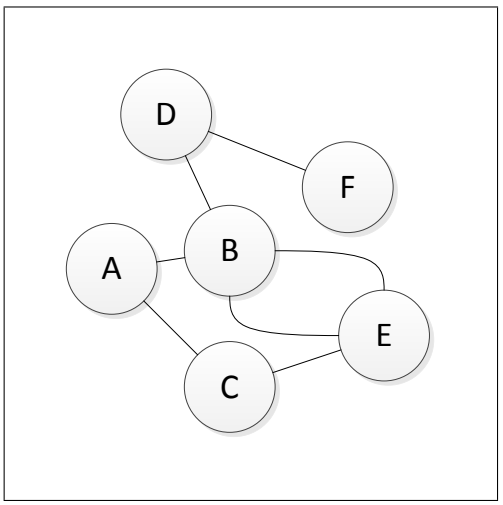
(a) einfach



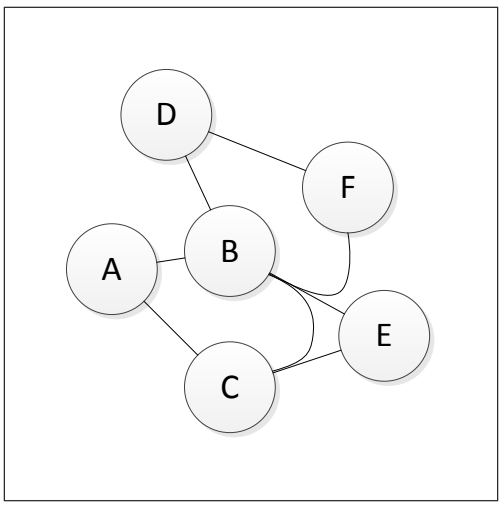
(b) gerichtet



(c) gewichtet



(d) Multigraph



(e) Hypergraph

Abbildung 7: Graph Klassifikationen

werden (Siehe Abbildung 7d). Ein Graph, der ausschließlich gerichtete Kanten beinhaltet, wird auch *Digraph*[1] genannt. Oftmals lassen sich die Kanten durch ihre Benennung unterscheiden. Ein soziales Netzwerk könnte also verschieden benannte Kanten wie „kennt persönlich“ und „mag“ zwischen zwei Personen einführen und somit mehrere Kanten zwischen den Personen zulassen.

Gerichtete Graphen

Gerichtete Graphen zeichnen sich durch die Eigenschaft aus, dass die Kanten nur jeweils unidirektional traversiert werden können. Im Beispiel (Abbildung 7b) gäbe es demnach einen Pfad von Knoten `<A, B, F>`, jedoch nicht in die Rückrichtung. Jede gerichtete Kante hat einen Start und einen Endknoten. Ein Graph, der ausschließlich aus gerichteten Kanten besteht, heißt *gerichteter Graph* oder *Digraph*. Schematisch werden gerichtete Kanten in Graphen wie im Beispiel ersichtlich oft durch Pfeile dargestellt. Ein Straßennetz würde die Darstellbarkeit der Richtung des Verkehrsflusses ermöglichen.

Hypergraphen

Ein Hypergraph ist ein Graph, bei dem Kanten mehrfache Verbindungen zwischen verschiedenen Knoten zulassen. Abbildung 7e zeigt eine solche Mehrfachverbindung zwischen den Knoten `<B, C, F ,E>`. Diese Darstellungsweise kann jedoch auch durch die „Auflösung“ der Mehrfachkanten in einzelne, separate Kanten dargestellt werden.

Kombinationen von Strukturen

Grundsätzlich sind Kombinationen dieser Klassen möglich, so dass gerichtete, gewichtete Multigraphen wie im CIDOC-CRM Modell in der Praxis keine Seltenheit sind. Ein Multigraph kann so beispielseise gerichtete und gewichtete Kanten enthalten, muss jedoch nicht darauf beschränkt sein.

2.3.2. Traversierung und Suche

Um eine sinnvolle Anwendung für Graphen zu erstellen, ist es notwendig, zunächst die Struktur der einzubringenden Daten in einen Graph zu überführen und dann Suchalgo-

[1] abgeleitet von engl.: „Directed Graph“

rithmen und Traversierungen für diesen Graph zu definieren, die die Ergebnisse für diese Anwendung liefern. Besondere Anpassungen der Algorithmen für andere Klassen als den einfachen Graphen sind notwendig, um Probleme wie z.B. eine unendliche Iterationstiefe von Ringpfaden zu umgehen. Da in der Prototypimplementation der Graphdatenbank für Kulturdaten keine zielgerichtete Suche durchgeführt wird und aufgrund der verfügbaren Daten keine informierte Suche möglich ist, werden in diesem Abschnitt die relevanten uninformierten[1] Suchalgorithmen dargestellt. [RN04]

Breitensuche

Bei einer Breitensuche[2] werden die direkten Nachbarn eines Ausgangsknotens traversiert. Kann der gesuchte Knoten nicht gefunden werden, wird automatisch eine weitere rekursive Breitensuche mit den entsprechenden Adjazenten des Ausgangsknoten als neuer Ausgangsknoten gestartet. Dieses Suchverfahren eignet sich besonders bei flachen Strukturen geringer Tiefe, da der Speicheraufwand von $O(|V| + |E|$ für große Graphen sehr hoch ist. [SP12, Seite 30ff]

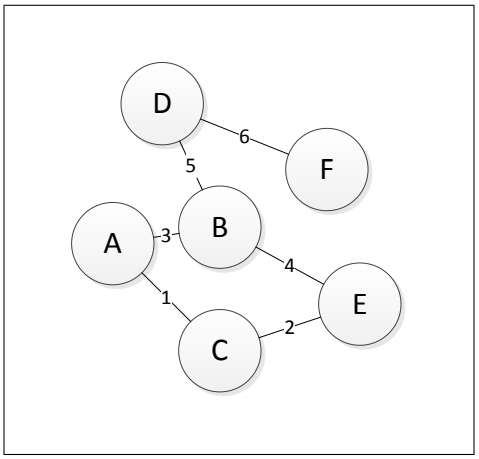


Abbildung 8: Breitensuche

Das Ergebnis ist für ungewichtete Graphen mit einem möglichen Ergebnis optimal, da immer das Ergebnis mit den wenigsten Traversierungen gefunden wird. Jedoch werden für Ergebnisse mit einer Tiefe $t \geq 2$ mehr Kanten als notwendig traversiert. Abbildung 8 demonstriert die Breitensuche ausgehend von `A` zu `F`. Dabei zeigen die Kantenbenennungen die Reihenfolge der Traversierung. [RN04]

[1] „blinde Suche“, ohne Einsatz von Heuristiken
[2] engl.: breadth-first search

Beschränkte Tiefensuche

Die (beschränkte) Tiefensuche[1] folgt rekursiv der nächstgelegenen Kante des Ausgangsknotens zum nächsten Endknoten bis zu einer Tiefe von n. Der Speicherverbrauch ist dabei gleich dem der Breitensuche, also $O(|V| + |E|)$. Das Ergebnis ist auch bei der Tiefensuche optimal wenn ein Ergebnis existiert, da diese ebenfalls das erste gefundene Ergebnis liefert. Weitere mögliche Ergebnisse werden jedoch nicht erfasst. Das führt bei mehreren möglichen Ergebnissen zu einem nicht optimalen Ergebnis. Die beschränkte Tiefensuche zeichnet sich vielmehr durch eine geringe Speicheranforderung aus. [RN04]

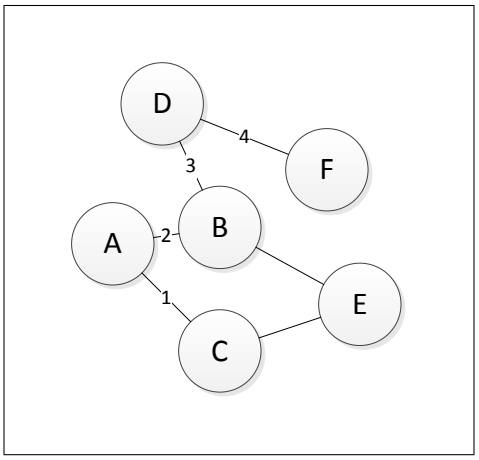


Abbildung 9: Tiefensuche

Iterative Tiefensuche

Die iterative Tiefensuche (ID DFS)[2] kombiniert die Optimalität der Lösung der Breitensuche mit dem Speicherverbrauch der Tiefensuche. Bei der iterativen Tiefensuche wird rekursiv eine beschränkte Tiefensuche mit Tiefe $t = 1$ für den Ausgangsknoten durchgeführt, welche in den jeweiligen Iterationen jeweils um 1 erhöht wird. Der Speicherverbrauch ist ähnlich zu dem der beschränkten Tiefensuche, da intern auf diese zurückgegriffen wird. Die Laufzeit steigt jedoch mit der Dimension der Tiefe. Das Ergebnis ist optimal.[RN04]

[1] engl.: depth-first search
[2] engl.: iterative deepening depth-first search

Kriterium	Breitensuche	Einheitliche Kosten	Tiefensuche	Beschränkte Tiefensuche	Iterative Vertiefung	Bidirektional (falls möglich)
Vollständig?	Ja^{a}	Ja^{a,b}	Nein	Nein	Ja^{a}	J^{a,d}
Zeit	$O(b^{d+1})$	$O(b^{\lceil c^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Speicher	$O(b^{d+1})$	$O(b^{\lceil c^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Ja^{c}	Ja	Nein	Nein	Ja^{c}	Ja^{c,d}

Legende: b ist der Verzweigungsfaktor, d die Tiefe der flachsten Lösung, m die maximale Tiefe des Suchbaums, l die Tiefenbegrenzung. ^{a}: vollständig, wenn b endlich ist, ^{b}: vollständig, wenn die Schrittkosten $\geq \epsilon$ für jedes positive ϵ sind, ^{c}: optimal, wenn alle Schrittkosten identisch sind; ^{d}: wenn beide Richtungen eine Breitensuche verwenden.

Tabelle 2: Bewertung der Suchstrategien nach [RN04, S. 115]

2.4. Anwendungsszenarien

Graphen sind besonders zur Topologieanalyse von vernetzten Daten geeignet. Fragestellungen wie: „Wie sind die Daten verbunden?“ oder „Wie traversiert man den Graphen optimal bis zu einem bestimmten Knoten?“ werden speziell hervorgehoben. Die Einsatzgebiete von Graphstrukturen umfassen oft auch np-vollständige Problemstellungen und machen sie so zur präferierten Darstellungsform für die typischen Anwendungsfelder wie z.B. Navigationssysteme. Weniger komplizierte Problemstellungen wie beispielsweise einfache verkettete Listen sind durchaus durch Graphstrukturen abzubilden, wenn auch durch ihre einfache Struktur nicht besonders herausfordernd.

Objektorientierte Programmiersprachen

Objektorientierte Programmiersprachen sind hauptsächlich definiert durch das Konzept der Vererbung von Objekten. Ein Objekt kann Eigenschaften und Methoden von Oberklassen „erben“, indem sie als Subklasse ihrer Oberklasse definiert wird. Die entsprechenden Eigenschaften werden dann von der Oberklasse wahlweise übernommen oder neu definiert, füllen also den Bezeichner der Methode der Oberklasse mit einem neuen Algorithmus, der dann als eigene Implementation angesehen wird. Damit eine Methode einer Oberklasse ausgeführt werden kann, wird für ein Objekt in aufsteigender Reihenfolge die Oberklassen nach der passenden Methodensignatur durchsucht bis diese gefunden

wurde. Probleme sind dabei u.A. Mehrfachvererbung oder das Finden der optimalen Klasse. [CTS06]

Soziale Netze

Graphen werden in sozialen Netzen dazu benutzt, Beziehungen zwischen Menschen darzustellen und diese auszuwerten. Beispielsweise könnten Knoten einer Graphdatenbank für einzelne Nutzer stehen und benannte Kanten (z.B. „Kennt“, „Mag“) für eine Art der Beziehung zwischen den Nutzern. So entstehen gerichtete oder ungerichtete Netze von Beziehungen die mit statistischen Mitteln beispielsweise in Cliquen[1] unterteilt werden können. Die „Hypothese der kleinen Welt“ besagt, wenn alle Personen der Welt miteinander in Beziehung stünden, würden sich zwei Personen über einen Pfad von höchstens 6 anderen Personen kennen. Es wäre hier also eine Traversierung über alle Kanten, ausgehend von Ausgangsperson A mit der Tiefe von maximal $c = 6$ notwendig, um zur Zielperson B zu gelangen.

Verkehrsnetze

Ein typischer Anwendungsfall von Graphstrukturen sind Verkehrsnetze, also Straßenkarten, die Wege zwischen einzelnen Punkten wie Städten, Plätzen etc. beschreiben. Derartige Netze werden durch gerichtete, gewichtete Graphen modelliert. Algorithmen wie Dijkstra[2] oder A*[3] zeigen dann für diese Netze optimale Traversierungen auf, um von Punkt A zu Punkt B zu gelangen. Gleichermaßen werden diese Algorithmen für dynamische Routingstrukturen z.B. im Internet verwendet.

2.5. Problemstellungen im Graphenbereich

Graphen können besonders bei analytischen Problemstellungen effizienter sein als relationale Modelle. Die typischen Problemstellungen umfassen beispielsweise die Traversierung von Graphen, das Finden von Nachbarn eines Knotens und Topologieanalysen.

[1] Gruppen von Personen, die sich untereinander gegenseitig kennen
[2] Algorithmus zur Berechnung des kürzesten Pfades zwischen zwei Knoten
[3] Erweitert Dijkstra Algorithmus um eine Abschätzfunktion

Hinzufügen von Daten

Daten können zu Graphen hinzugefügt werden, indem eine Graphstruktur mit Eigenschaften versehen wird. An einen bestehenden Knoten werden also Felder mit Nutzdaten angehängt, wie z.B. „ID“ oder „Value“, so dass dieser Knoten ein einmaliges Objekt ergibt. Es wird mindestens eine Kante angehängt, die wiederum auf einen weiteren datenhaltenden Knoten verweist. Zunächst muss also der Startknoten identifiziert werden, wozu sich in der Praxis ein einfacher Index[1] eignet.

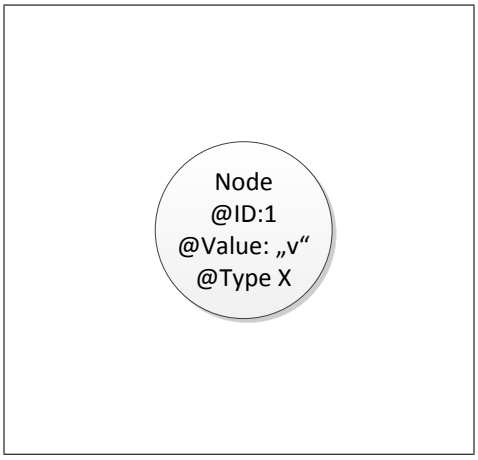


Abbildung 10: Datenhaltung im Knoten

Adjazenzen

Um die Nachbarn (Adjazenten) eines Knoten zu identifizieren, reicht es aus, sämtliche Knoten der Kanten des Ausgangsknotens mit Tiefe 1 zu betrachten. Im Beispiel 11 wären also `<A, D, E>` die Adjazenten von Knoten `B`.

Mit Hilfe der Adjazenzmatrix (vgl. Abbildung 4) lassen sich diese verknüpften Nachbarn vergleichsweise einfach ablesen. Beispielsweise ist $M_{1,2} = 1$, damit ist B als Nachbar von A identifiziert. Adjazenzlisten lassen sich dagegen direkt ablesen. In einer Graphdatenbank hätte Knoten B eine Liste von Referenzen auf seine Nachbarn, die nacheinander traversiert werden können. In einer Inzidenzmatrix wird zunächst der gewünschte Knoten gesucht. Für jede Spalte, die für den Knoten einen positiven Wert definiert hat, wird zeilenweise der Endpunkt der Verbindung gesucht.

[1] Register zur Beschleunigung der Suche von Datensätzen innerhalb einer Datenbank

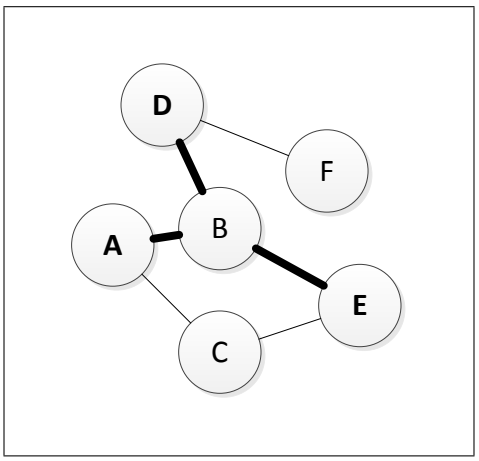


Abbildung 11: Nachbarschaften von B

Löschen von Kanten und Knoten

Für isolierte Knoten, die nur eine Kante haben, ist das Löschen unproblematisch. Derartige Knoten können mitsamt der verbindenden Kante entfernt werden. Jedoch kann es abhängig von domänenspezifischen Maßgaben unmöglich sein, Knoten mit mehreren Adjazenten zu entfernen. Derartige Knoten können als Verbindungspunkt zwischen mehreren unabhängigen Clustern fungieren. Der Knoten `<B>` aus dem Beispielgraphen ist ein Knoten, der Teilnetze miteinander verbindet. Dieser Knoten kann nicht gelöscht werden ohne dass eine implizite Verbindung der Teilnetze dabei entfernt wird (Abbildung 12). Eine Neuverbindung dieser Strukturen ist in bestimmten Fällen nicht zulässig, da die möglichen Verbindungen zwischen den Kanten abhängig vom Domänenmodell bzw. der Semantik der Daten beschränkt ist.

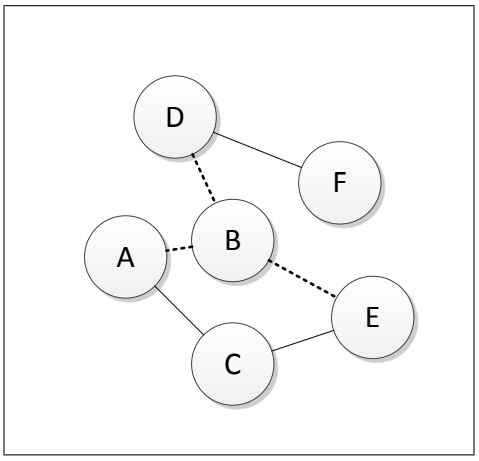


Abbildung 12: Löschen von Knoten

Entsprechend ergibt sich daraus eine nichttriviale Problemstellung, die für die Einzelfälle der Anwendung gelöst werden muss. Eine allgemeingültige Lösung ist nicht anwendbar.

Nachladen von Werten

Implementationen von Graphdatenbanken versprechen Traversierungen in Echtzeit[1] bei mehreren Milliarden Kanten. Bei einer Datenbank dieser Größe ist der Speicheraufwand für die Nutzdaten jedoch enorm. Es existieren daher Mechanismen wie „lazy fetching“, die die Daten von Knoten nur auf explizite Anfrage abrufen, so dass lediglich die Signaturen der Kanten im Speicher gehalten werden. Bis zum expliziten Abruf werden die Nutzdaten des Knoten z.B. auf einem Festspeicher oder in einer externen Datenbank persistiert und der komplette Datensatz per indexiertem Abfragemechanismus abgerufen.

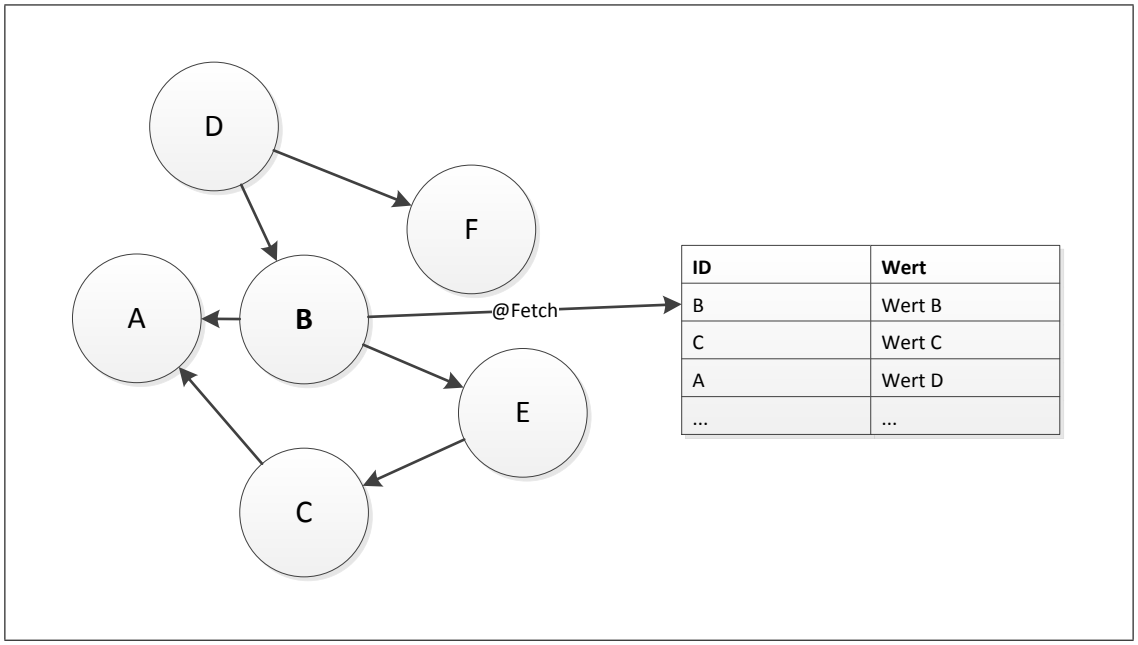


Abbildung 13: Nachladen von Werten

Topologieanalyse

Anforderungsabhängig können Graphen einer statistischen Topologieanalyse unterzogen werden. Dabei werden die Daten auf herausragende Merkmale wie beispielsweise Häufigkeiten bei Pfadkombinationen oder Clusterbildung, untersucht. Pfadkombinationen

[1] Die Berechnung wird garantiert innerhalb einer gegebenen zeitlichen Beschränkung durchgeführt

sind Pfade unbestimmter Länge zwischen Knoten, die unabhängig voneinander in der Struktur oft anzutreffen sind. Im CIDOC-CRM Modell könnte dies für die Auswertung von Facettenanfragen genutzt werden (Siehe Abschnitt 3.2.3).

Replikationsmethoden

In einem zentralisierten Replikationsumfeld übernimmt ein Steuerungsserver die Aufgabe, Daten auf mehrere Server zu verteilen, die dann jeder für sich einen Teil der Daten enthalten. Dezentrale Umfelder sind dagegen selbstorganisierend, kommen demnach ohne Steuerungsserver aus indem sie sicherstellen, dass jeder Datensatz mindestens einmal in der Gesamtserverstruktur vorhanden ist, in sicherheitskritischen Umfeldern auch öfter. Eine replizierbare Datenbank umfasst weitere Optimierungen wie Replikationssalgorithmen, die Teile der Datenbank in einer größeren Gesamtdatenstruktur verteilen um entweder Verfügbarkeit durch Spiegelung als auch Performanz durch Verteilung zu erhöhen. Die Knoten und Kanten werden dabei dezentral auf mehreren Servern gehalten und auf Anfrage abgerufen [SP12, S. 13].

Da diese Server üblicherweise durch weniger durchsatzstarke Techniken wie z.B. einem Netzwerk oder dem Internet angebunden sind, ist insbesondere die Verteilung der Daten, als auch die Formulierung der Anfragen dementsprechend zu optimieren, dass möglichst wenige Kantenverknüpfungen zwischen den einzelnen Servern traversiert werden. Ein soziales Netz, das eine geringe Clusterbildung aufweist, dient als Beispiel für eine schwer verteilbare Struktur, da die Daten kaum Clusterbildung, dafür jedoch gleichzeitig ein sehr hohes Aufkommen an Anfragen für Kantentraversierungen aufweisen. Üblicherweise ist die Topologie eines sozialen Netzes abhängig von der Anzahl der Nutzer hochgradig dynamisch, da Beziehungen werden regelmäßig hinzugefügt, geändert oder gelöscht werden. Insofern sind übliche Algorithmen für das Verteilen von Graphstrukturen hier nicht anzuwenden.

Die Erfassung von Metadaten ist dagegen ein besonders geeignetes Beispiel für die Verteilbarkeit von Graphdatenbanken, da die Daten eine sehr hohe Clusterbildung aufweisen und die Änderungsrate der Daten gleichzeitig sehr niedrig ist.

Traversierung

Eine Traversierung bezeichnet das unidirektionale Verfolgen eines Pfades von einer oder mehr miteinander verbundenen Knoten und Kanten innerhalb eines Graphen. Der Algo-

rithmus ist auch hier von der Zielsetzung abhängig. Um eine Abfrage zum Finden eines einzelnen Knoten am Ende eines Pfades zu erstellen, wird ausgehend vom Startknoten eine Pfadtraversierung definiert, die durch bestimmte Kriterien wie dem Kantentyp beschränkt ist. In einem sozialen Netz ist dies beispielsweise die Anfrage: „Wieviele Friends of Friends hat Person A in Tiefe 3?“ (Anwendungsfall der Performanzanalyse in Abschnitt 3.5.3).

Da Neo4j bereits sämtliche dieser elementaren Techniken unterstützt, ist das Framework ein potenziell ideales Werkzeug für die Verwaltung eines großen verknüpften Netzes kultureller Metadaten.

3. Kulturdaten als Showcase für Graphdatenbanken

Kulturdaten sind ein Oberbegriff für kulturelle Metadaten[1], welche Daten über Werke wie Bücher, Bilder, Statuen oder Musik umfassen. Dabei wird nicht der Inhalt der Werke selbst als Metainformation bezeichnet, sondern lediglich dessen Beschreibung. Die Werke werden von den einzelnen Einrichtungen weitestgehend unabhängig voneinander verwaltet, so dass sie oft unterschiedliche Formate zur Metadatenerfassung verwenden.

3.1. Die Deutsche Digitale Bibliothek

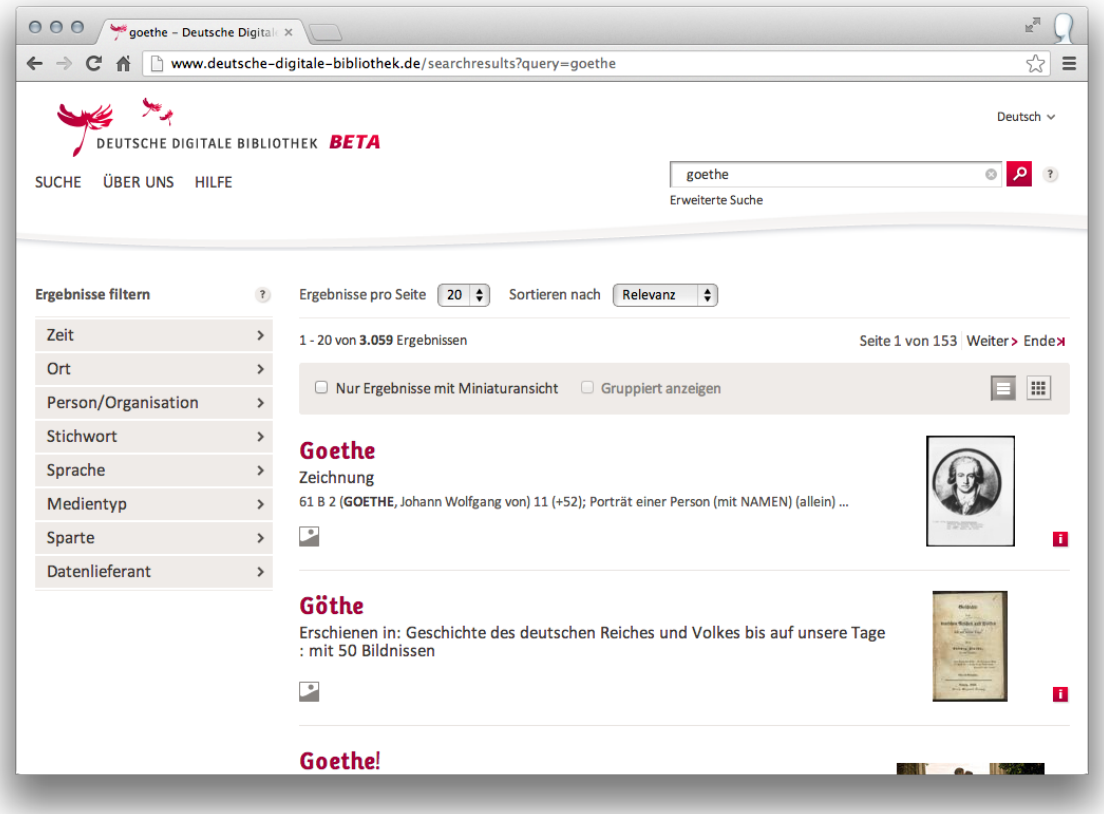


Abbildung 14: Die Deutsche Digitale Bibliothek

[1]Metainformationen: Daten über Merkmale anderer Datensätze, nicht aber die Datensätze selbst

Das Projekt „DDB“[1] stellt als Gemeinschaftsprojekt initiiert von Bund, Ländern und Kommunen einen Ansatz dar, Metadaten[2] deutscher kultureller Objekte automatisiert semantisch[3] miteinander zu vernetzen und dadurch eine kontextbezogene Methode zur Kulturrecherche zu ermöglichen. Das Ergebnis dieses langfristigen Projekts soll über ein Webportal einem breiten Publikum zur Verfügung gestellt werden. Auftragnehmer für die Planung, Entwicklung und technischen Entwicklung ist das IAIS Fraunhofer[4], welches für die erste Ausbaustufe seitens des Bundesbeauftragten für Kultur und Medien BKM mit der technischen Gesamtkoordination und -realisierung in der Laufzeit von März 2010 bis Dezember 2011 betraut war. Beginn der technischen Entwicklung hat im 3. Quartal 2010 stattgefunden. Seit dem 28.11.2012 ist die Deutsche Digitale Bibliothek im Internet für die Allgemeinheit zugänglich. Datenlieferanten für den bisherigen Realtestdatenbestand in Form von Texten, Videoclips, Bildern oder digitalen Tonformaten von etwa 6.2 Mio. Datensätzen sind verschiedene deutsche kulturelle und wissenschaftliche Einrichtungen (KWE).

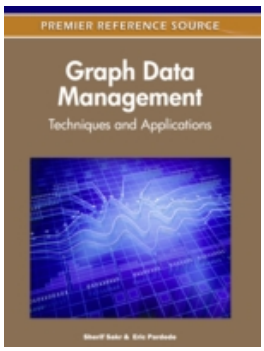
3.2. Problembeschreibungen

Der Abschnitt stellt einige der Probleme dar, die in der Deutschen Digitalen Bibliothek gelöst werden. Die Problemstellungen sind dabei speziell auf die Anforderungen ausgerichtet, die sich durch die Verwendung von verknüpften Metadaten ergeben. Der Folgeabschnitt 3.3 beschreibt konkrete Lösungsansätze zu den hier beschriebenen Problemstellungen.

3.2.1. Metadatenformat

Die gespeicherten Metadateninformationen liegen in den KWE in verschiedenen, oft zueinander inkompatiblen Metadatenformaten vor. Eine Seitenzahl könnte in einem Beispielformat A direkt einem Objekt, wie z.B. einem Buch als Eigenschaft zugeordnet sein, in Format B, das auf musikalische Kompositionen ausgelegt sein könnte, kann

[1] http://www.iais.fraunhofer.de/ddb.html
[2] eine Ressource beschreibende Daten, nicht jedoch die Ressource selbst
[3] Ein Semantik versetzt ein technisches System in die Lage, eine Verbindung aufgrund der eigentlichen Bedeutung herzustellen
[4] Fraunhofer Institut für Intelligente Analyse und Informationssysteme



Graph Data Management

By:	Sherif Sakr; Eric Pardede
Publisher:	IGI Global
Pub. Date:	August 31, 2011
Print Isbn-10:	1-61350-053-X
Web Isbn-10:	1-61350-054-8
Print Isbn-13:	978-1-61350-053-8
Web Isbn-13:	978-1-61350-054-5
Pages in Print Edition:	346

Abbildung 15: Metadaten zu Buch „Graph Data Management“

diese bereits irrelevant sein. Um eine Einheitlichkeit zwischen den einzelnen Metadatensätzen herzustellen, ist es notwendig, ein Metadatenformat[1] zu nutzen, das Metadaten aus sämtlichen unterschiedlichen Standardformaten und Quasi-Standards ohne Verlust von Information abbilden kann.

Das weit verbreitete Metadatenformat Dublin Core, welches von der DMCI[2] erarbeitet wurde, definiert beispielsweise 15 Kernfelder zum Speichern von Metadaten. Diese Felder liefern eine Grundmenge an Metadaten, bilden jedoch keine ausreichende Menge an Information ab. Zwar ist Dublin Core frei erweiterbar, jedoch müssen sich durch den präzisionsverlust durch die freie Definierbarkeit der Felder bei Austausch über Dublin Core sowohl die Quelle als auch das Ziel für die Daten über die Syntax und die Abgrenzung einig sein um eine hinreichende Güte der Daten zu gewährleisten. Strukturierte Formate wie MARC-21[3] ermöglichen eine größere Präzision, sind wiederum nur für einen eine kleine Menge von Ressourcentypen ausgelegt. Es muss also ein Metadatenformat erschaffen werden, welches typübergreifend Metadaten präzise erfassen kann.

Das Beispiel in Abbildung 15 zeigt einige der Metadaten des Buches „Graph Data Management“ [SP12]. Dies sind jedoch nur die für einen potenziellen Leser verwertbaren Informationen. Die eigentlich möglichen Metadateninformationen sind zahlreicher, da Daten wie Restauration oder sogar geographischer Wandel durch Grenzverschiebung erfasst werden[4].

[1] technisches Austauschformat für Metadaten
[2] Dublin Core Metadata Initiative
[3] Machine-Readable Cataloging Standards zur Beschreibung von Metadaten in Bibliotheken
[4] Eine Restauration oder eine Neuauflage könnten ebenso erfasst werden wie unpräzisere Metadaten.

3.2.2. Finden von Adjazenzen über Metadaten

Mit der Erstellung eines vollständig definierten Domänenmodells ergibt sich ein großes Netz von einzelnen Entitäten mit ihren beschreibenden Metadaten. Diese Objekte können über die benannten und typisierten Pfade miteinander verknüpft werden. Die Darstellung in Abbildung 16 verdeutlicht das Prinzip eines solchen Netzwerks von Metadaten, ist jedoch mit drei verbundenden Datensätzen und nur einigen gericheten (benannten) Kanten lediglich ein simples Beispiel zur Demonstration des Prinzips.

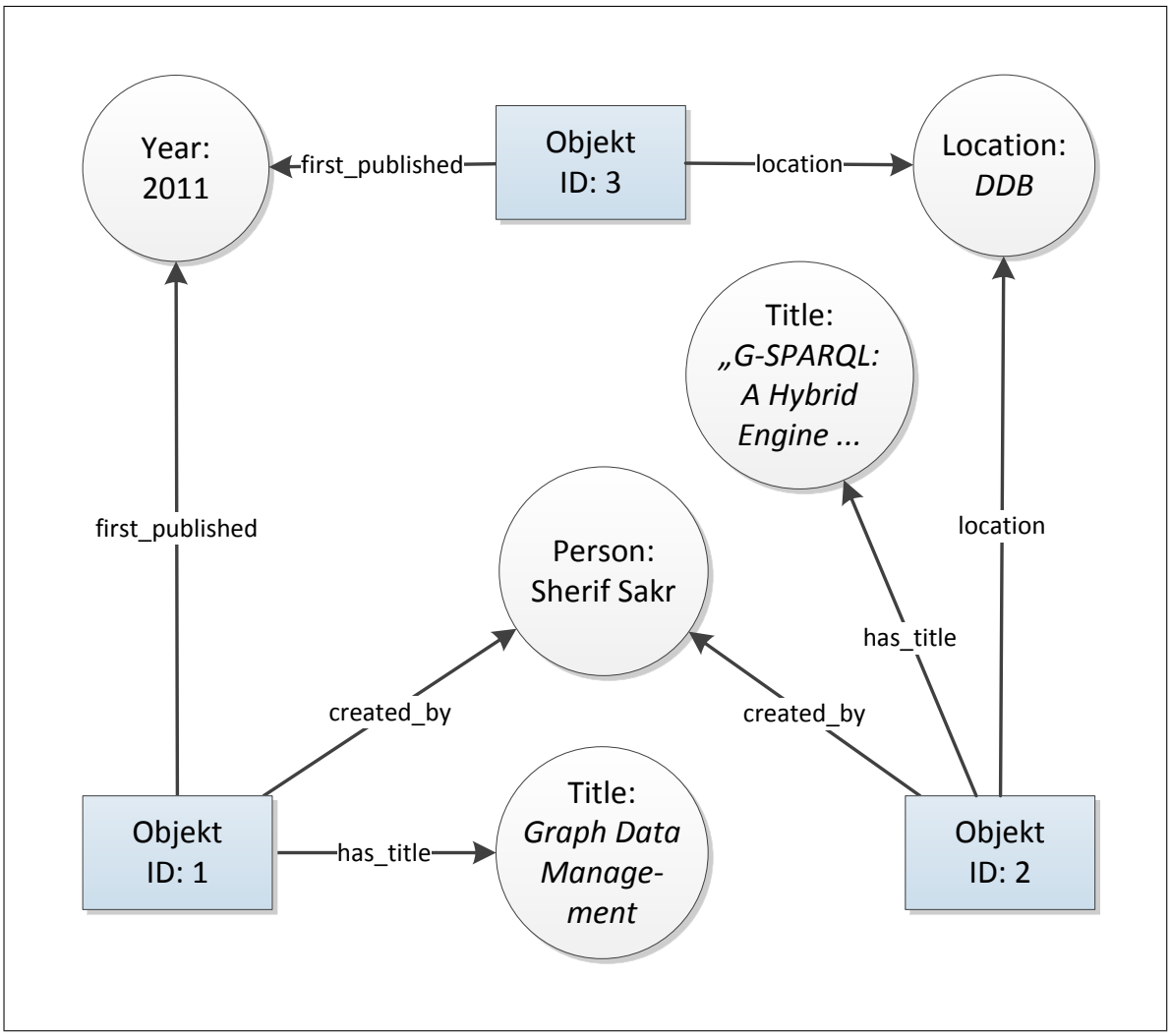


Abbildung 16: Verbundene Metadaten im Nodestore

In der Abbildung wird ersichtlich, dass ein Pfad, sprich eine Liste von Kanten existiert,

Ein Werk könnte sich nach einer Grenzverschiebung z.B. im Weltkrieg in einem anderen Land befinden

der alle drei Objekte implizit miteinander verbindet. Die Objekte 1 und 2 haben beispielsweise denselben Autor. Entsprechend ergibt sich über diesen traversierten Pfad eine neue implizite Relation. Gleichermaßen ist Objekt 3 im gleichen Jahr erschienen wie Objekt 1 und befindet sich am selben Standort wie Objekt 2. Dieses Netzwerk von verbundenen Metainformationen ermöglicht erweiterte Abfragen von Datensätzen ausgehend von einem Objekt. Damit ein solcher Graph von verbundenen Daten erstellt werden kann, müssen die Daten zunächst in ein einheitliches Metadatenformat überführt werden. In einem Prozess der Überführung in eine Graphstruktur werden einzelne Objekte manuell oder durch intelligente Vergleichsalgorithmen an möglicherweise vorhandene Strukturen angebunden.

3.2.3. Facetten

Die facettierte Suche ist wie in der DDB indexbasiert auf vielen derzeitigen Plattformen mit digitalem Inhalt zu finden. Diese ermöglicht einzelne Ergebnisse innerhalb einer Liste, ähnlich einem Kombinationsfilter, abhängig von bestimmten Kriterien auszuwählen. Ein so definierter Kombinationsfilter verfolgt das Prinzip, dass mehrere mehrere Kriterien definiert werden können, so dass sukzessiv Daten aus der Ergebnisliste ausblendet werden. Die facettierte Suche hat somit einen großen Nutzen für die detaillierte Recherche einzelner Objekte bei einer großen Trefferanzahl (Beispielabbildung 17).

Möchte ein Nutzer beispielsweise alle Bücher finden, die zur Zeit der industriellen Revolution in Bremen geschrieben wurden, würde er in einer Webmaske exklusive Filterkriterien erstellen, die folgende Kriterien enthalten:

- Zeit: „1870-1940“
- Ort: „Bremen“
- Medium: „Monographie“

Eine Abfrage der Graphdatenbank würde zusammenhängende Teilstrukturen von Objekten suchen, die diese Kriterien erfüllen und die entsprechenden Objekte als Ergebnis zurückliefern. Dies impliziert für relationale Datenbanken einen erheblichen Rechenaufwand, da alle Objekte durch JOIN-Vergleiche auf passende Teilbäume überprüft werden müssen.

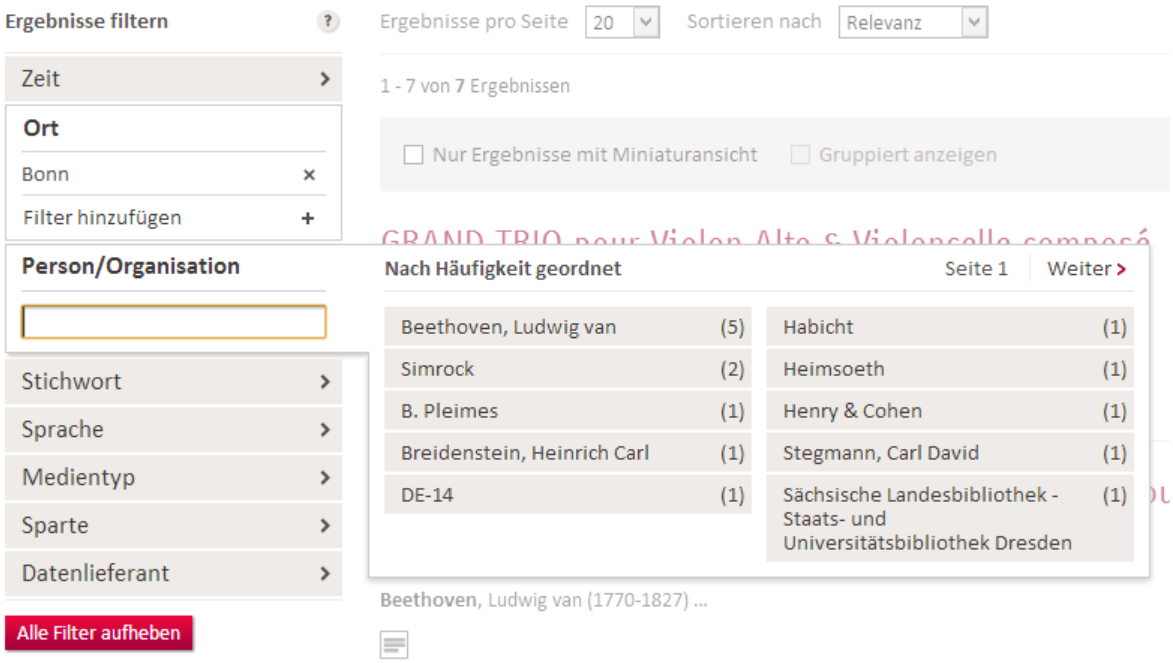


Abbildung 17: Facetten in der DDB

Damit die Datenlast in einem Graphen durch viele Join-Operationen bewältigt werden kann, wird in der DDB ein simpler Mechanismus verwendet: die Filteranfragen werden vorberechnet und danach indiziert. Das Erstellen eines Filters extrahiert hier nicht direkt Objekte aus dem Graphen durch Suchanfragen eines Teilbaums, sondern befragt lediglich den Index. Abbildung 18 verdeutlicht das Prinzip der vorberechneten Indizes. Es werden zwei Indexe dargestellt, die jeweils eine Bindung zwischen Autor und Titel mit dem Basisobjekt herstellen. Wenn der Index nach einem Autor A befragt wird, wird Objekt 1 als Ergebnis ausgeliefert.

Ein großer Nachteil dieses Mechanismus ist, dass er abseits der bereits vorberechneten Ergebnisse keine weiteren liefern kann, die Freiheit der Anfragen ist damit stark eingeschränkt.

3.2.4. Verbindungen zwischen Objekten

Herkömmliche Suchmaschinen stellen bekannterweise eine einfache Möglichkeit der Suche zur Verfügung. Hierzu gibt ein Nutzer üblicherweise eine Auswahl von Stichworten, die das seiner Meinung nach gesuchte Ergebnis am besten beschreiben in ein Suchfeld

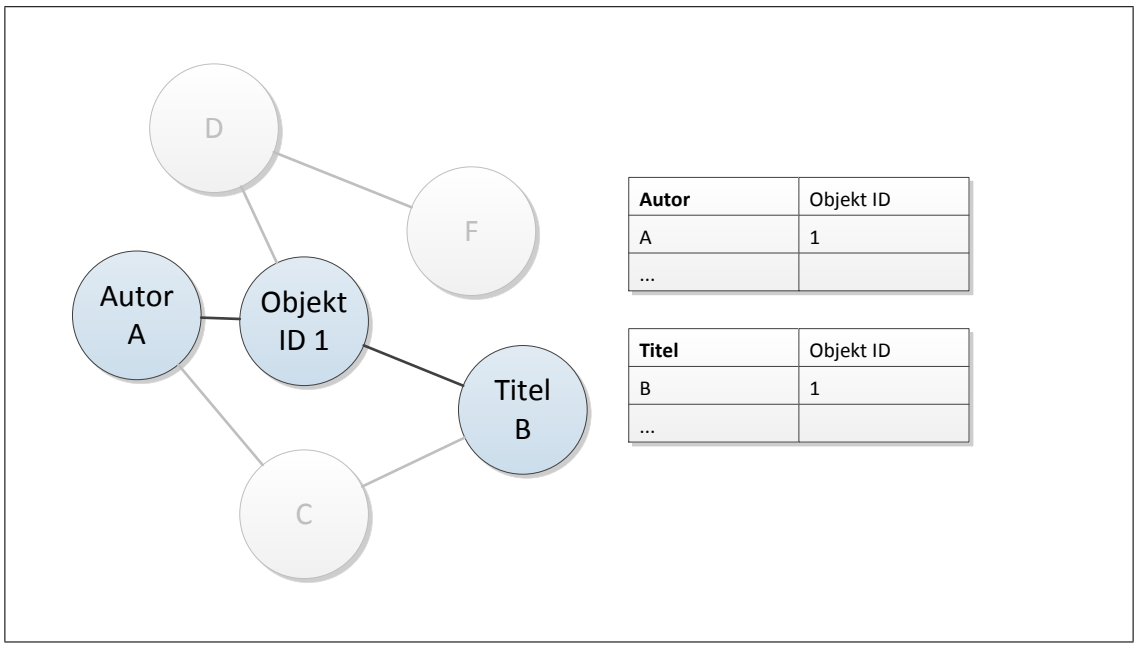


Abbildung 18: Vorberechnete Facetten in Graphen

ein. Eine Volltextsuchmaschine wertet diese Anfrage aus und liefert basierend auf Statistiken und heuristichen Algorithmen eine Liste von Resultaten. Es werden jedoch keine Querverbindungen zwischen Objekten erfasst, so dass lediglich auf Basis der Stichwörter des Nutzers gesucht werden kann. Wünschenswert wäre es, wenn die Suchmaschine alle wichtigen Informationen nicht nur zu dem direkten oder ähnlichem Suchergebnis, sondern gleichzeitig mit allen anhängenden Informationen bereitstellen könnte. Ein Nutzer, der z.B. nach „Immanuel Kant“ sucht, könnte so direkt die Werke aller Kritiker des „Kategorischen Imperativs“ erfahren.

3.2.5. Intelligente Vergleichsalgorithmen

Ein nichttriviales Problem ergibt sich durch die stark heterogenen Methoden der Darstellung der Daten. „Johann Wolfgang von Goethe“ könnte gleichermaßen „J.W. von Goethe“ oder einfach „Goethe“ benannt werden. Die Frage nach der Verbindung zwischen den Schreibweisen ist in diesem Beispiel für einen Menschen recht einfach zu lösen, denn Goethe wird allgemein als Autor berühmter Werke assoziiert. Ungleich schwieriger ist jedoch die Frage bei Personen wie „Peter Müller“, der ein Buch verfasst haben könnte, und „P. Mueller“, der ein berühmtes Bild erstellt haben könnte. Hierzu sind heuristische

Vergleichsmethoden notwendig, die diese Objekte vom Typ „Person“ konsolidieren oder auch voneinander trennen.

Ein Anwendungsgebiet, bei der dieses Prinzip notwendig ist, ist das Importieren neuer Datensätze. Es kann vorkommen, dass nacheinander mehrere Bücher vom selben Autor gespeichert werden. Jede Metadatendatei zu den Büchern kann komplette oder nur teilweise vorhandene Informationen zum Autor enthalten, obwohl es sich faktisch um ein und dieselbe Person handelt, was darin resultiert, dass ein Autor mehrfach in der Datenbank angelegt werden würde und somit mehrere IDs hätte. Um dies zu verhindern, kann ein intelligenter Vergleichsalgorithmus helfen, eine Identitätsprüfung einzelner Teilbäume durchzuführen.

3.2.6. Verbindungen zwischen Teilbäumen

Um einen hinreichenden Vergleich der Identität zwischen zwei Objekten darzustellen, reicht es üblicherweise nicht, Literale, die Werte enthalten auf Inhalt und Typ zu untersuchen. Vielmehr ist es notwendig, ganze Teilstrukturen eines Metadatenbaumes zu überprüfen um eine hinreichende Bedingung für Identität zu erhalten. Ein Autor ist in einer Graphdatenstruktur beispielsweise zunächst ein Knoten vom Typ „Person“, der mit weiteren beschreibenden Knoten verbunden ist, wie „Name“, „Adresse“ oder „Geburtsdatum“. Um ein hinreichendes Identitätsmerkmal zu finden, müssen alle Kanten geprüft werden, die dem Autor anhängen, ohne dass beispielsweise ein kulturelles Objekt wie ein Buch in den Vergleich mit aufgenommen wird. Relationale Datenbanken müssen hierzu für jedes Kriterium, das hinzugefügt wird eine weitere kostenintensive JOIN-Abfrage durchführen.

3.3. Lösungsansätze

Dieser Absatz stellt theoretische Lösungsansätze zu den in Abschnitt 3.2 beschriebenen Problemen dar. Diese Lösungsansätze dienen als Basis für die im Prototyp entwickelten Abfragen und Algorithmen.

3.3.1. Metadatenformat

Das Problem der zueinander oft inkompatiblen, jedoch weit verwendeten Metadatenformate wurde im Projekt DDB gelöst, indem XSLT-Prozessoren[1] definiert wurden, die die Kompatiblität der Klassen zueinander möglichst ohne Informationsverlust herstellen sollen.

Da das Ausgangsformat über OAI Schnittstellen[2] üblicherweise in einer wohldefinierten XML-Struktur abgebildet ist, lassen sich XSLT-Transformatoren nutzen, um automatisiert diese Ausgangsstrukturen in die Zielstruktur zu überführen. Dazu werden die zunächst notwendigen Felder des Ausgangsformates identifiziert. Eine semantische Bindung der Bedeutungen der einzelnen Felder des Ausgangsformates mit dem CIDOC-CRM Format wird manuell hergestellt. Xslt-Prozessoren übernehmen dann automatisiert die Transformation der Strukturen.

Miyasat Alieva beschreibt in ihrer Magisterarbeit die Vorgehensweise im Detail [Ali12].

3.3.2. Finden von Adjazenzen über Metadaten

Ein Objekt kann mit mehreren anderen Objekten implizit über seine Metadaten verbunden sein. Es muss also ein direkter oder impliziter Pfad über eine oder mehr Kanten gefunden werden, der zwischen zwei Objekten oder Clustern existiert. Um einen Pfad zwischen zwei vordefinierten Kanten herauszufinden, muss also ein Suchalgorithmus wie Breiten-, oder Tiefensuche (siehe Abschnitt 2.3.2) sämtliche Pfade traversieren bis zu dem bekannten Zielknoten. Für diesen Anwendungsfall existieren bereits zahlreiche Algorithmen, die die Komplexität durch intelligente heuristische Suchen wie z.B. A*-Suche maßgeblich verringern.

3.3.3. Facetten

Eine Facette ist in einem Graphen gleichzusetzen mit einer wiederkehrenden Teilstruktur von bestimmten Kantentypen. Sprachlich ausgedrückt wäre folgende Facette in einem

[1] XSLT definiert XML Klassen mit Definitionen zur automatisierten Umwandlung von XML Dateien in ein definiertes Zielformat

[2] OAI-PMH beschreibt Protokolle für den webbasierten Austausch von Informationen

Metadatengraphen zu finden: „Alle Bücher, deren Autor im Jahr 1980 geboren ist“. In einer Graphdatenbank wäre dies eine Stuktur folgender Art:

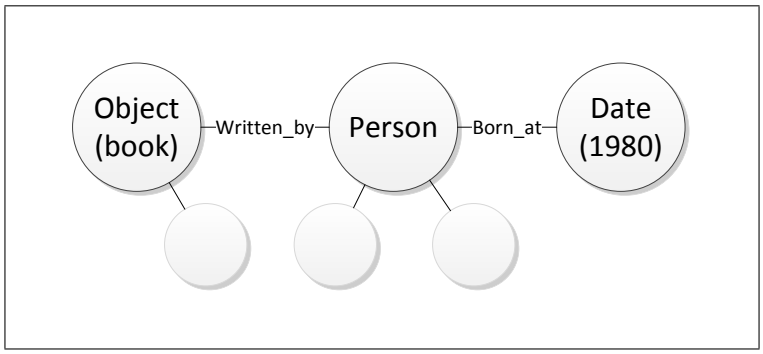


Abbildung 19: Beispielgrafik Facetten

In einer Graphdatenbank ist es einfach, derartige wiederkehrende Teilstrukturen anhand von Merkmalen wie Datentypen zu erkennen und zu extrahieren. Es kann also eine Anfrage formuliert werden, die ausgehend von den Kanten- und Knotentypen, sowie einzelner Werte Kategorien erstellt, die wiederum als Facetten dienen können. Durch die besondere Topologie einer Graphdatenbank vereint diese Möglichkeit gleichzeitig Effizienz, sparsamen Umgang mit Systemressourcen, sowie freie Abfragen und Facetten. [SP12, Kapitel 2]

3.3.4. Verbindungen zwischen Objekten

Graphdatenbanken bieten erstmals die Möglichkeit, Objekte anhand ihrer gespeicherten Metadaten über Traversierungen abhängig von den Metadaten eines Ausgangsobjektes zu identifizieren. Hierzu können adjazente Objekte über Traversierungen ausgewählt oder Pfade gesucht werden, die Verbindungsketten zwischen zwei Objekten anzeigen. Eine interessante Anwendung der Graphdatenbanken ist es, zwei Objekte miteinander zu vergleichen indem die kürzesten Pfade dieser Objekte als Ergebnis einer speziellen Suche bereitgestellt werden. Diese Verbindungsketten würden dem Nutzer einen Aufschluss über die „Güte“ geben, mit der zwei Objekte zusammenpassen. Diese Güte der ausgewählten Pfade ist gesondert zu bewerten.

Der Pfadvergleich zwischen zwei verschiedenen Büchern ist beispielsweise irrelevant, wenn die einzige Gemeinsamkeit darin besteht, dass beide vom gleichen Ressourcentyp, also Bücher sind. Aussagekräftiger ist dagegen ein Pfad des berühmten Gedichtes

„Ozymandias“[1] von Percy Bysshe Shelley, das einen eindeutigen Pfad zur Statue des altägyptischen Pharaos Ramses II[2] aufweist, welche im Britischen Museum in London ausgestellt ist, da Shelley durch diese Statue zu dem Gedicht inspiriert wurde. Auffällig ist, das beide Werke einen unterschiedlichen Ressourcentyp haben und auch zeitlich und geographisch weit auseinander liegen, und trotz allem eine implizite Verbindung zueinander besteht, die nicht direkt ersichtlich ist. Auffällig ist, das beide Werke einen unterschiedlichen Ressourcentyp haben und auch zeitlich und geographisch weit auseinander liegen, und trotz allem eine implizite Verbindung zueinander besteht, die nicht direkt ersichtlich ist.

Eine weitere denkbare Verbindung dieser Werke könnte sein, dass ein Bild von Ramses II zusammen mit dem Gedicht von Shelley an einem gemeinsamen Ort aufbewahrt wird, wie einer Bibliothek oder als Referenzen in einem Buch. Die Anzahl, durchschnittliche Pfadlänge und das Maß der Abstraktion ist ein Indiz für die Güte, mit der diese beiden Werke miteinander in Verbindung stehen.

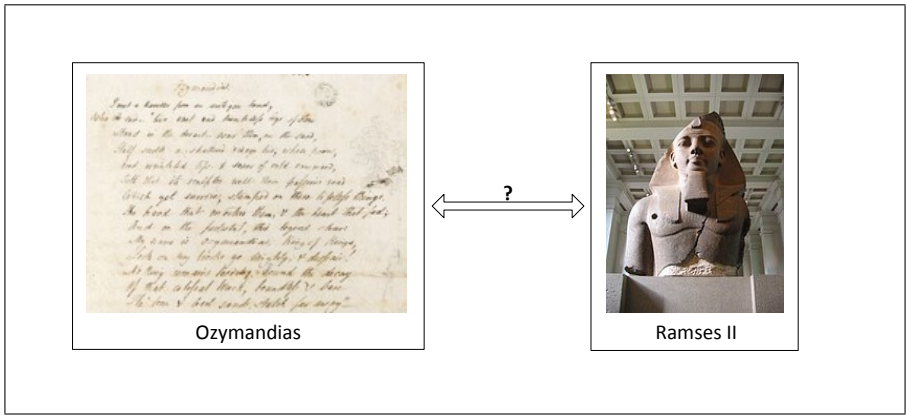


Abbildung 20: Ramses 2 und Ozymandias

3.3.5. Intelligente Vergleichsalgorithmen

Um zwei Datensätze auf eindeutige Übereinstimmung zu prüfen, ist es praktikabel, diese anhand ihrer ID zu prüfen. Da dies jedoch nicht immer möglich ist[3], müsste ein intelligenter Vergleichsalgorithmus eine Prüfung abseits der eindeutigen Merkmale durchführen.

[1]Public Domain Lizenz
[2]Creative Commons Attribution-Share Alike 3.0 Unported license
[3]Beispielsweise beim Speichern neuer identischer Datensätze, die noch keine ID haben

Grundsätzlich sind die Mittel der Computerlinguistik dazu geeignet, ein entsprechendes Distanzmaß für die Wahrscheinlichkeit einer Übereinstimmung zweier Datensätze anhand ihrer Metadaten zu liefern. Am IAIS Fraunhofer wurde im Rahmen des DDB Projekts in der Abteilung „Knowledge Discovery“ ein derartiger Algorithmus entwickelt. Dieser Algorithmus betrachtet sowohl Struktur der Metadaten als auch die Werte der einzelnen Felder. Da sich die Werte der Felder je nach Datentyp unterscheiden, muss für jeden Datentyp ein passender Algorithmus entwickelt werden. Ein Datum kann beispielsweise sowohl in der Form „1.12.2013“, als auch in der Form „2013/12/1“ dargestellt werden. In den Bibliotheksdatenbeständen existieren jedoch auch Datensätze mit Beschreibungen, wie „im 19. Jahrhundert“, die so nicht eindeutig verglichen werden können, da sie kein konkretes Datum darstellen. Eine Heuristik kann jedoch einen Wahrscheinlichkeitswert für die Übereinstimmung dieser Daten liefern, der mittels einem Schwellwert einer Übereinstimmung entweder zustimmt oder diese ablehnt.

Werden den Suchkritierien ferner Werte, wie z.B. ein Name hinzugefügt, können diese in die Anfrage einfügen. Ist jedoch der Name nicht eindeutig, kann dies zu weiteren Problemen führen. In diesem Fall müssten mehr Kriterien angeführt werden, die die Person eindeutig unterscheidbar machen. Die Wissensbasis der Suche muss um weitere semantische Referenzen wie „Der Autor ’Ian Fleming’ hat ’James Bond’ verfasst“ erweitert werden, was graphdatentechnisch ohne großen Aufwand abzubilden ist. Ein derartiges Konzept wird im Folgeabschnitt 3.3.6 dargestellt.

3.3.6. Verbindungen zwischen Teilbäumen

Notwendige Informationen für die Unterscheidung zweier Datensätze durch einen derartigen Algorithmus sind die Kriterien: „Datentyp“, „Wert“ und das „Metadatenumfeld“. Beispiel: Um einen Autor in der Datenbank ohne Angabe einer ID zu finden, ist es zunächst notwendig zu wissen, welche Metadaten eines Autors im Allgemeinen notwendig sind. Ein Autor ist eine Person mit einem Namen, einem Geburtsdatum, einem Geburtsort, etc. Diese Daten liefern in einem Graph ein Basisobjekt X mit Relationen zu weiteren Objekten, wie z.B. eine Relation vom Typ „heißt“ zu einem Objekt vom Typ „Name“ und einem Wert sowie eine Relation vom Typ „geboren_am“ zu einem Objekt vom Typ Zeitpunkt mit dem Geburtsdatum. Diese Informationen ergeben ein Metadatenobjekt, das anhand seiner „Signatur“ verglichen werden kann. [SP12, Seite 80]

Um einen Autor in der Datenbank zu erkennen, muss eine Abfrage nach der Signatur „Personenentity mit Relation vom Typ 'heißt' zu einer Namensentity und Relation 'geboren_am' zu Entity 'Geburtsdatum' und Relation 'geboren_in' zu Entity 'Geburtsort'" suchen. Eine derartige Anfrage sucht sämtliche Personenentities nach den vorgegebenen Kriterien ab, zunächst ohne Betrachtung der Werte. Erst wenn diese der Abfrage hinzufügt werden, kann nach konkreten Datensätzen für Personen gesucht werden. Das Prinzip ist metaphorisch mit einer „Schablone" einer Sternbildkonstellation vergleichbar, die sämtliche Sternenkonstellationen am Himmel abdeckt, aber nur auf eine einzeln Konstellation passt. Abbildung 21 stellt dieses Prinzip der Überdeckung dar. Es ist erkennbar, dass für die Suche irrelevante Knoten in den Hintergrund rücken, eine Suchabfrage jedoch den passenden Teil des Graphen überdeckt. Knoten A gehört zwar zum Subgraph, ist jedoch in der Suche nicht mit inbegriffen.

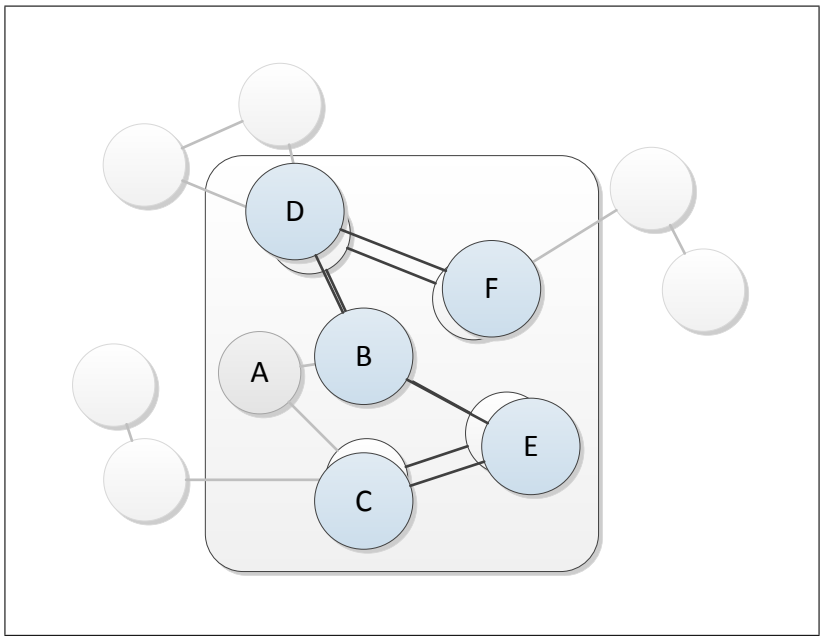


Abbildung 21: Graph Vergleichsalgorithmen

Es ergeben sich zwei Probleme, wenn ein Objekt nicht deckungsgleich mit dem anderen ist. Das Eingangsbeispiel vom Autor könnte durch einen später hinzugefügten Datensatz um weitere Merkmale ergänzt werden oder das Geburtsdatum könnte sich in einem zweiten Datensatz durch eine andere Schreibweise unterscheiden. In beiden Fällen würde bei der Ausführung einer „einfachen" Identitätsprüfung der Originaldatensatz nicht gefunden werden. Einerseits müsste der Algorithmus eine Heuristik zur Prüfung aufweisen, der Objekte, die nicht hundertprozentig übereinstimmen trotzdem zuverlässig identifiziert.

Andererseits müsste der Algorithmus tolerant gegenüber Fehlern in der Schreibweise der Literale sein.

3.4. Abstraktion

Dieser Abschnitt stellt einige der Mittel dar, die sich für eine Implementation als notwendig herausstellen.

Tripel

Das Semantic Web[1] ist ein Ansatz, den Dokumenten im Internet eine Semantik, also eine Bedeutung zuzuweisen, die algorithmisch verarbeitet werden kann. Dazu ist es notwendig, die von Menschen geschriebenen Ressourcen mit einer kurzen Beschreibung zu versehen, die kurz und prägnant eine Semantik beinhaltet und durch Tripelrelationen miteinander in Verbindung gebracht werden kann. Ein Tripeldatensatz besteht aus den drei atomaren Elementen: Subjekt, Prädikat und Objekt. Als Subjekt wird dabei die Ausgangsressource bezeichnet, das Prädikat ist eine Eigenschaft der Ressource, die auf ein Objekt, also eine Entität hinweist. Beispiele für Tripel sind Sätze, wie:

„Das Gedicht 'The Raven' *hat* '18 Strophen'“
„Die 'Deutsche Digitale Bibliothek' *wurde implementiert am* 'IAIS Fraunhofer'“

Sind die Datentypen der Ressourcen bekannt, können mit Hilfe dieser einfachen Satzkonstruktionen erstmals Metadaten mitsamt ihren Eigenschaften semantisch erfasst werden. Jede Tripelrelation bildet also eine Hierarchie, die demnach eine Graphstruktur ist.

RDF

Das Resource Description Framework, kurz RDF, drückt Tripelrelationen mit Hilfe von XML Standards aus. Es ist wegen des einfachen Datenmodells, formaler Semantik und eines vollständigen Inferenzmechanismus das vom W3C empfohlene Datenmodell zur Repräsentation von Ressourcen im Web. RDF umfasst drei Teilmodelle: „RDF“, „RDFs“

[1] Semantic Web bezeichnet den Ansatz, Informationen für Maschinen semantisch berechenbar zu machen

und „OWL“, die zusammengenommen eine Semantik jeder Information darstellen können [SP12, Seite 335]. Das folgende Codebeispiel stellt einen minimalen Auszug aus der in RDF implementierten CIDOC-CRM Struktur dar.

Codebeispiel 1: RDF XML Beispiel

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:Class rdf:ID="E1.CRM_Entity">
    <rdfs:comment>...</rdfs:comment>
  </rdfs:Class>
  <rdfs:Class rdf:ID="E2.Temporal_Entity">
    <rdfs:comment>...</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#E1.CRM_Entity"/>
  </rdfs:Class>
</rdf:RDF>
```

Das Beispiel lässt erkennen, dass der Knoten `E2.Temporal_Entity` eine Unterklasse von `E1.CRM_Entity` ist. Dieser Sachverhalt wird im folgenden Abschnitt genauer erläutert.

CIDOC-CRM

Das CIDOC-CRM[1] beschreibt einen ISO Standard[2] für ein formales Domänenmodell zur Repräsentation kultureller Metadaten. Mit dem Zweck den Austausch von Metadaten zwischen kulturellen Einrichtungen zu vereinheitlichen und somit für technische Systeme zugänglich zu machen, wird es seit 1994 stetig weiterentwickelt und ist derzeit in Version 5.04 frei[3] verfügbar.

CRM Ontologiemodell

Das Modell enthält Entitäten, die Eigenschaften von Objekten beschreiben und Relationen, die die Beziehung zwischen den Instanzen der einzelnen Entitäten darstellen. Die Benennung von Entitäten und Relationen erfolgt durch einen innerhalb des CRM Modells eindeutigen Schlüssel, kombiniert mit einer textuellen Darstellung, z.B.

[1] CIDOC Conceptual Reference Model
[2] ISO 21127:2006
[3] http://www.cidoc-crm.org/definition_cidoc.html

`E31:Document` oder `P94F:has_created`. Die Instanz, also eine konkrete Ausprägung einer Entität besteht aus dem Tupel (T, V) mit T als Typ der Entität und V, dem Wert als Literal. Eine Relation enhält dagegen (T, S, E) mit S und E als jeweilige Referenz auf einen Start- bzw. Endpunkt einer Entität. Die Kanten des Graphen sind folglich gerichtet. Die Ausrichtung der Relationen spiegelt sich gleichermaßen in der Benennung wider. Eine Relation, wie `P94F:has_created` enthält den Identifikationsschlüssel `P94F` und die textuelle Ausprägung des Namens `has_created`. Das Kürzel `P94F` endet mit dem Buchstaben `F`, was darauf hindeutet, dass die Relation vorwärtsgerichtet ist. Die zu diesem Beispiel passende Relation zur Umkehrung der Richtung wäre `P94B:was_created_by`. Es ergeben sich wie bei RDF Tripelstrukturen der Form Subjekt-Prädikat-Objekt.

Abbildung 22 stellt beispielhaft eine Instanz dieser Struktur dar. Das Buch „Graph Data Management“ [SP12] ist also ein Objekt vom Typ `E31:Document`.

Hierarchie

Topologisch beschreibt das Modell eine Baumstruktur, d.h. die einzelnen Entitäten und Relationen „erben“ ihre Eigenschaften von den jeweiligen generalisierenden Oberklassen. Der Beispielknoten `E31:Document` ist im CIDOC-CRM Modell eine Spezialisierung von `E73:Information Object`. Der Wurzelknoten `E1:CRM_Entity` ist die allgemeinste Darstellung der einzelnen Subknoten. CIDOC-CRM unterstützt Mehrfachvererbung. Eine Entity oder Relation kann somit von mehr als einer Oberklasse erben. Die Entity `E24:Physical Man-made Thing` aus Abbildung 13 erbt gleichzeitig von `E71:Man-made Thing` und `E18:Physical Thing`, welche jedoch beim Verfolgen der Hierarchiekette, allesamt vom Typ `E70:Thing` sind. Die in der Vererbungshierarchie „oberste“ Entität ist `E1:Entity`. Sie beschreibt allgemein eine Ressource, die Oberklasse für sämtliche Entitäten ist.

Jede Entität kann mit Properties verbunden werden. Jede Property hat genau einen Startpunkt und einen Endpunkt, sowie in seltenen Fällen auch eine Vererbungshierarchie. Die Start- und Endpunkte einer Relation sind Entitäten von einem definierten Typ. Durch Ausnutzen der Vererbungsmechanik kann eine „allgemeine“ Relation, also eine Relation, die für alle Entitäten zulässig sein soll durch Einsetzen des Typs `E1:Entity` als Start- bzw. Endpunkt definiert werden.

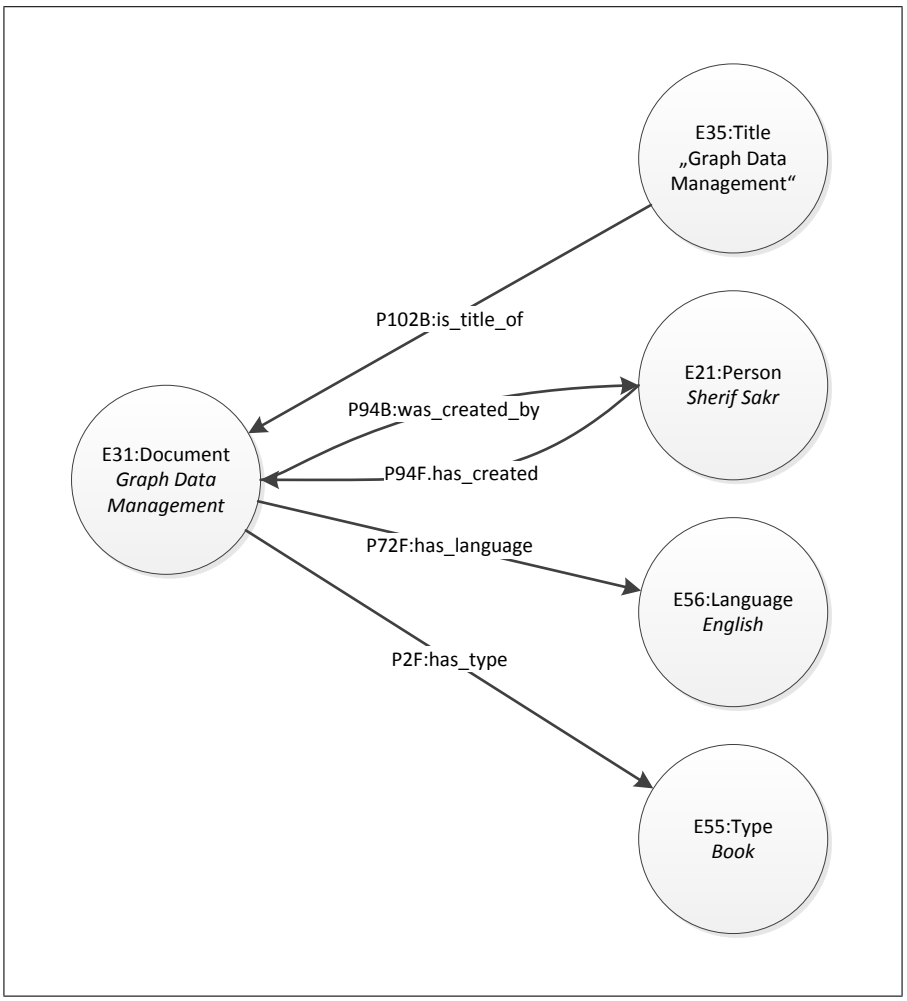


Abbildung 22: CRM Anwendung Beispiel

Eine Entity kann mit mehreren Properties des gleichen oder verschiedenen Typen mit anderen Entitäten verbunden werden. Dabei ist jedoch eine Tripel[1]relation immer systemweit eindeutig. Eine Verknüpfung „A kennt B“ kann nicht mehrmals definiert werden, jedoch sind die Relationen „A kennt B“ und „A kennt C“ zulässig.

Da CIDOC-CRM durch gerichtete, benannte Relationen definiert ist, sind die Kanten eines Graphen entsprechend einzurichten, denn einige Relationen sind durchaus nur einseitig gerichtet.

[1]3-Tupel, das im semantischen Web eine Subjekt-Prädikat-Objekt Satzstruktur abbildet

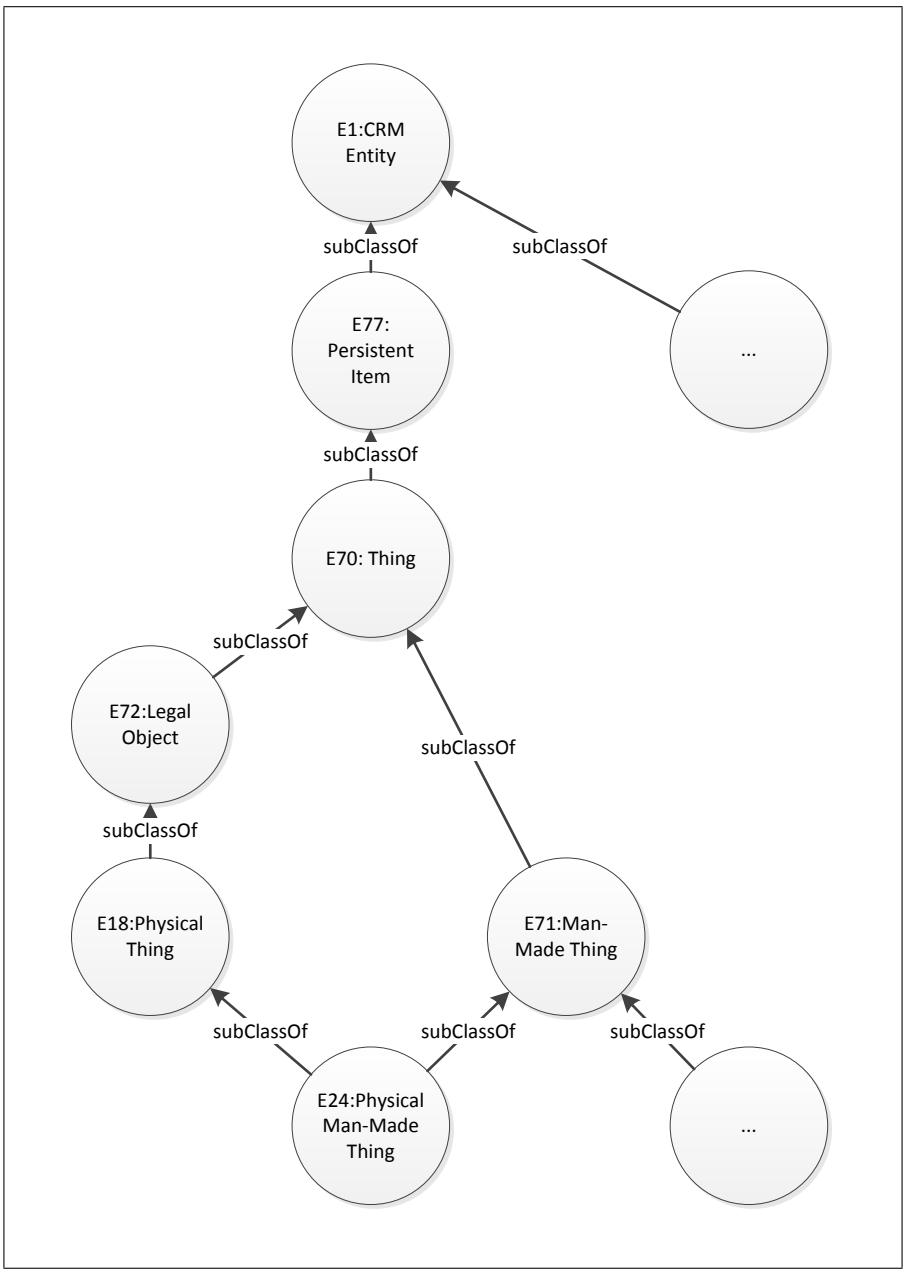


Abbildung 23: CRM Vererbung Beispiel

3.5. Technologien

Damit ein Domänenmodell wie CIDOC-CRM abgebildet werden kann, bedarf es einer günstigen technischen Voraussetzung. Einige der Möglichkeiten werden in diesem Abschnitt vorgestellt und miteinander verglichen, um so die Vor- und Nachteile der einzelnen Techniken herauszustellen.

3.5.1. Relationale Datenbanken

Abbildung 24 stellt ein Beispielmodell zur Abbildung des CRM Modells in einer relationalen Datenbank dar.

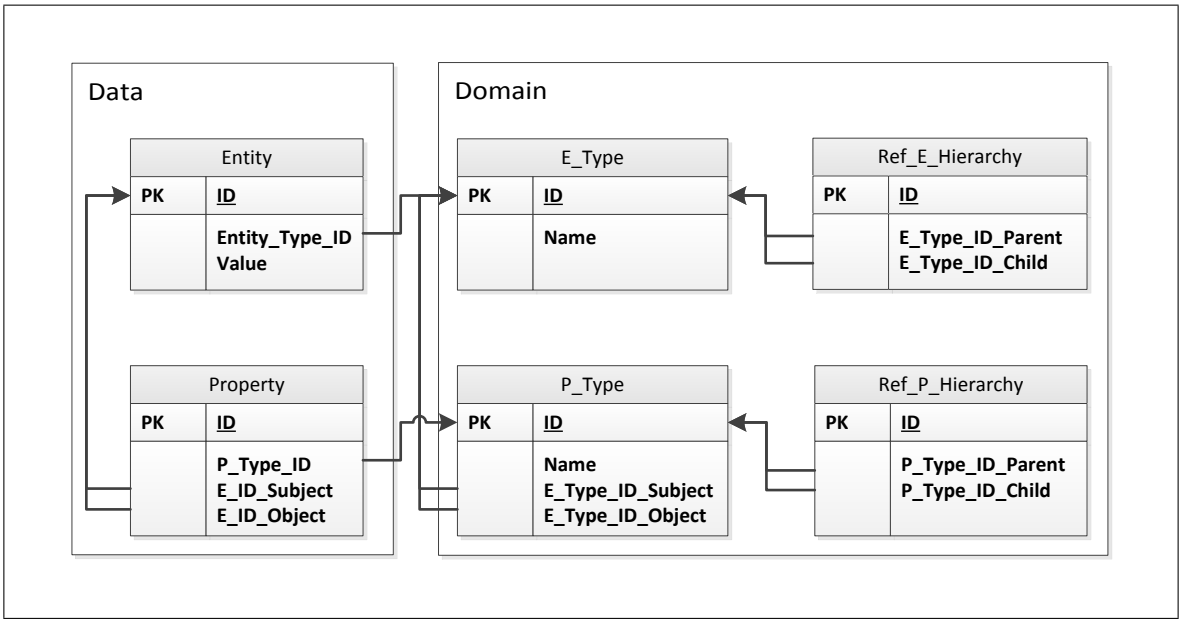


Abbildung 24: CIDOC Relational Database

Das Modell bildet eine einfache Datenbankstruktur ab, die in zwei Teile partitioniert ist: Daten und Domäne. Der Datenteil enthält in der Tabelle `Entity` die Knoten mitsamt eines Wertes und in der Referenztabelle `Relation` Verbindungen zwischen Entities. Das Domänenmodell bildet die in `E_Type` und `R_Type` Datentypen von Entities und Relations ab mitsamt eines Hierarchiemodells für die in **??** definierte Vererbungshierarchie.

Obwohl Relationale Datenbanken als versatiles Modell zur Abbildung von Relationen auf eine lange und erfolgreiche Laufbahn zurückblicken können, werden diese mit der Anzahl der heutzutage zu bewältigenden Datenmengen abhängig von der Aufgabenstellung zunehmend als ineffizient eingestuft und durch leichtgewichtige NoSQL Techniken ersetzt.

Codebeispiel 2: SQL Abfrage für Entity

```
SELECT ID FROM Entity
  INNER JOIN Property ON Property.E_ID_Subject = Entity.ID
```

```
INNER JOIN Entity title ON Entity.ID = Property.E_ID_Object
WHERE P_Type_ID =  1
      -- 1 = Beispiel Referenz ID fuer Property 'has_title'
AND title.value LIKE "DDB"
```

Codebeispiel 2 demonstriert die Komplexität einer SQL Query einer Basisentität, die über eine Relation mit einer Metadatenentität den Titel „DDB“ abfragt.

Da Relationale Datenbanken Suchoperationen durch Vergleiche von Tabellen implementieren, werden in dieser Abfrage drei Tabellen vollständig miteinander verglichen, zusätzlich zur Volltextsuche des Titels und des Typs der Relation. Diese Art der Anfrage ist bereits theoretisch hochgradig ineffizient bei großen Tabellen. Bei Gegenüberstellung der potenziellen kulturellen Metadaten wird jedoch das gesamte Ausmaß deutlich, die eine derartige einfache Suchoperation durchführt:

Rechenbeispiel

Die Daten der DDB umfassen derzeit etwa 6 Mio Testdatensätze. Unter der Annahme, dass jeder Datensatz im Durchschnitt etwa 20 Metadaten umfasst, füllt sich die Tabelle `Entity` bereits mit 20*6 Mio Datensätzen, also 120 Mio Datensätzen. Die Tabelle `Property`, die die Relationen zu den einzelnen Datensätzen erfasst, speichert pro Metadatum bei gerichteten Kanten also etwa die doppelte Anzahl der Entitäten, also 240 Millionen. Die einfache Anfrage in Codebeispiel 2 vergleicht die Tabellen Entity, Property und Entity mit JOIN Operationen miteinander. In einem hypothetischen Beispiel ohne intelligente Indizes wären dies 120*240*120 Millionen Vergleichsoperationen. Durch Einbeziehen des statistischen Mittelwertes einer Komplexität von O(log n) für das Findens eines Elements in einem Datenbankindex reduziert sich die Abfragen auf

$$log(120 * 10^6)^2 * log(240 * 10^6) = \ 547$$

für eine einfache Suchanfrage. Eine Aufgabe, die für relationale Datenbanken nicht zu bewältigen ist. Daher werden bestimmte häufig genutzte Metadaten wie Titel oder Autor oft gesondert in einem eigenen Volltextindex gespeichert. Komplexere Suchanfragen, wie eine facettierte Suche multiplizieren die Zahl der Anfragen pro Kantentraversierung um den Faktor 68. D.h. ein weiteres Suchkriterium würde bereits 37034 Vergleichsoperationen bedeuten. Eine Traversierung über mehr Kanten ist in diesem Modell also undenkbar.

Auch andere Quellen kommen zu Ergebnissen, die die mangelnde Performanz von SQL Joins beschreiben:

> **Inefficiency of SQL Joins**
> In order to find all user's friends on depth 5, relational database engine needs to generate Cartesian product of t_user_friend table five times, With 50,000 record in the table, the resulting set will have 50,000 rows $(102,4 \times 10^{21})$, which takes quite a lot of time and computing power to calculate. Then, we discard more the 99% of that, to return just under 1,000 records we're interested in! [PVW13, S. 6]

Die Anfrage liefert zudem nur den „Hauptknoten" als Ergebnis der Suche. Die einzelnen Metadaten, wie z.B. Autor, Datum, Auflage, Art, Titel etc müssen in zusätzlichen Abfragen erschlossen werden. Sollen ganze Pfade als Anfrage gestellt werden, wächst die Komplexität der Anfragen exponenziell. Ebenso verhält es sich mit der Traversierung von Pfaden, also dem Folgen der Pfadketten bis zu einem bekannten Ergebnis.

3.5.2. Apache Solr

Apache Solr[1] ist eine in Java implementierte Volltextsuchmaschine, die Apache Lucene um Elemente wie hohe Skalierbarkeit, verteilte Suche, Index Replikation[2] etc. erweitert. Solr ist im Gegensatz zu relationalen Datenbanken nicht tabellenbasiert und zählt somit zu den NoSQL Datenbanken. Das unter APL 2.0 veröffentlichte Projekt ist hochperformant, optimiert für extreme Datenmengen und eine große Anzahl an Transaktion[3]en und ist somit bestens geeignet für Dokumentverarbeitung mit groß angelegten Datenmengen. Das Projekt „Cortex[4]", das den Kern der Implementation der DDB bildet, verwendet Solr als technische Basis für die Haltung und Verarbeitung der Metadaten, von der herkömmlichen Volltextsuche bis zur Verarbeitung der Graphdaten.

Damit eine verknüpfte Graphstruktur in einem dokumentbasierten Index wie Solr implementiert werden kann, müssen die Referenzen innerhalb des Indexes eindeutig iden-

[1] In Java implementierter Apache Volltextindex
[2] mehrfache Speicherung derselben Daten an meist mehreren verschiedenen Standorten und die Synchronisation dieser Datenquelle
[3] Möglichkeiten der Kapselung ganzer Sätze von Anfragen, die als im Fehlerfall als Ganzes rückgängig gemacht werden können
[4] Kernsoftware der Deutschen Digitalen Bibliothek

tifizierbar organisiert werden. Ein Basisobjekt muss also Felder zu den anhängenden Metadaten enthalten. Dies ist in Solr theoretisch weitgehend unproblematisch, da die einzelnen Datensätze eine unterschiedliche Anzahl von Feldern enthalten können, so auch Referenzfelder mit Schlüsseln auf die Metadatensätze. Praktisch entsteht jedoch ein Problem bei der Datenhaltung eines Graphs von Kulturdaten in einer zweidimensionalen Dokumentverwaltung durch die abfallende Leistung beim Speichern neuer Daten. Da die Daten in Cortex während der Datenaufnahme miteinander vernetzt werden, müssen große Mengen von Vergleichsoperationen durchgeführt werden. Diese werden nicht Solr-intern durchgeführt, sondern müssen extern verknüpft werden, so dass ein Meßwert von 4000 Transaktionen auf Testsystemen pro Sekunde[1] im laufenden Betrieb keine Seltenheit darstellen.

Jedoch hat Solr einen entscheidenden Vorteil dadurch, dass es für große Indexe konzipiert worden ist: häufig angewendete Facetten können vorberechnet und indexiert werden. Das Problem und eine Lösung wird unter Abschnitt 3.2 intensiver diskutiert.

3.5.3. Neo4J

Springdata Neo4j ist eine in Java implementierte Graphdatenbank, die sich durch besonders große Performanz bei der Verwaltung und Analyse von Graphstrukturen auszeichnet. Das unter zwei Open-source Lizenzmodellen stehende Projekt wurde im Februar 2010 als stabile Version 1.0 freigegeben und steht seit September 2012 in Version 2.1.0 zur Verfügung.

Neo4j Community enthält die Graphdatenbank und sämtliche für die Entwicklung und Verwendung notwendigen Basismittel.

Neo4j Advanced enthält die Community Edition und Module für erweitertes Monitoring.

Neo4j Enterprise enthält zusätzlich Online Backup, High Availability Clustering und Advanced Monitoring-Module.

Neo4j kann durch ein zusätzliches, kommezielles Lizenzmodell auch in Closed-source und kommerziellen Anwendungen verwendet werden[2].

[1] Meßwert der Lasttests vom Dezember 2011
[2] Quelle: http://www.neo4j.org/learn/licensing

Die Software wird in zwei Versionen ausgeliefert:

Neo4j Embedded: kann per Java in Softwareprojekte eingebettet werden. Sämtliche Programmlogik wird vom entwickelten Programm verwaltet.

Neo4j Server: liefert einen Jetty Webserver mit REST Webschnittstellen zur Verfügung, sowie einfache Steuerungs- und Überwachungsmechanismen

Das gut dokumentierte Framework beschreibt Szenarien für die Analyse großer vernetzter Datenmengen und ist daher ideal für die Verwendung von vernetzten Metadatenstrukturen. Gleichzeitig werden Fähigkeiten relationaler Datenbanken wie ACID-konforme Transaktionen, Indizes, abstrakte Abfragesprachen wie Cypher[1] und Gremlin, sowie REST[2] Unterstützung angeboten. Neo4j ist ein für den Entwickler leichter zugänglicher Ansatz zu Graphstrukturen, als es mit relationalen Datenbanksystemen möglich ist, denn eine vernetzte Graphstruktur kann direkt von einer Konzeptzeichnung in die endgültige Datenbank überführt werden ohne eine weitere Abstraktionsebene über Tabellenmodelle hinzuzufügen. [Hun12]

Um die Performanzunterschiede bei tiefer Traversierung von Graphstrukturen zu analysieren, wurde ein fiktives soziales Netz mit 1.000.000 Nutzern erstellt, die jeweils einen Durchschnitt von 50 Bekanntschaften haben. Die Anfragen beschränken sich dabei auf Traversierungen in den verschiedenen Ebenen mit den entsprechenden Verzweigungen durch die „kennt“ Verknüpfungen. Tabelle 3 zeigt die Performanceunterschiede bei tiefer Traversierung.

Tiefe	Relationale Datenbank (s)	Neo4j (s)	Ergebnisse
2	0,016	0,010	ca 2.500
3	30,267	0,168	ca 125.000
4	1.543,505	1,359	ca 600.000
5	*nicht messbar*	2.132	ca 800.000

Tabelle 3: Vergleich Performanz MySQL und Neo4j [nach PVW13, Seite 9-10]

Bei diesem Test steigt die Zeit für die Abfragen in RDBMS deutlich an, während Neo4j vergleichsweise wenig Zeit beansprucht. RDBMS sind bei Traversionstiefe 5 bereits weit

[1] Abfragesprache für in Graphstrukturen organisierte Datenquellen.
[2] Representational State Transfer

über der maximalen Auslastungsgrenze, während die Performance von Neo4j bei dieser Komplexität im Vergleich nur langsam ansteigt.

Der Grund dafür liegt teilweise in der optimierten Datenhaltung der Neo4j Datenbank. In der Vorausgabe des Buchs Neo4j in Action wird als Erklärung dazu folgende Analogie hergestellt.

> ***What is the secret of Neo4j's speed?***
> *No, Neo4j developers haven't invented super-fast algorithm for the military. Nor is the Neo4j speed product of fantastic speed of the technologies it relies on – it's implemented in Java after all! The secret is in the data structure – the localized nature of graphs makes it very fast for this type of traversals. Imagine yourself cheering your team on the a football stadium. If someone asks you how many people is sitting five meters around you, you will get up and count them. If the stadium is half empty, you will count people around you as fast as you can count. If the stadium is packed, you will still do it in a similar time! Yes, it may be slightly slower, but only because you have to count more people because of the higher density. We can say that, irrespective of how many people are on the stadium, you will be able to count the people around you at predictable speed – as you're only interested in people near you, you won't be worried about packed seats on the other end of the stadium for example. This is exactly how Neo4j engine works in our example – it counts nodes connected to the starting node, at the predictable speed. Even when the number of nodes in the whole graph increases (given similar node density), the performance can remain predictably fast. If you apply same football analogy to the relational database queries, we would count all people in the stadium and then remove those that not around us – not the most efficient strategy given the interconnectivity of the data.*
>
> [PVW13, S.10]

4. Implementation eines Prototypen

Die Graphstruktur der Kulturdaten kann durch Neo4j erfasst werden. Dabei muss zunächst das CIDOC-CRM Modell in eine Neo4j-kompatible Klassenstruktur überführt werden. In diesem Abschnitt werden Möglichkeiten diskutiert, die diese Abbildung ermöglichen.

4.1. Modell

Um das CIDOC-CRM Domänenmodell zu erfassen, müssen die einzelnen Entities und Relations in eine Neo4j-kompatible Java Klassenstruktur überführt werden. Neo4j bietet hierzu für verschiedene Anwendungsfälle passende Ansätze. Ursprünglich wurde Neo4j in einfachen Java Objekten implementiert. Die Neo4j Fähigkeiten mussten zum größten Teil ohne vorgefertigte Struktur selbst implementiert werden. Dieses Konzept bietet zwar die beste Anpassbarkeit, bedarf jedoch eines großen Entwicklungsaufwandes.

Die mit Springdata eingeführte und seit Version 2.0 in Neo4j implementierte Möglichkeit, Klassen durch Annotationen für die Verwendung mit Neo4j zu kennzeichnen, reduziert den Entwicklungsaufwand, da die Klassen nun nativ erstellt werden müssen. Neo4j übernimmt sämtliche Verbindungen vollautomatisch. Das unter AspectJ bekannte Aspektorientierte Programmierparadigma bietet die Möglichkeit, generische Funktionalitäten klassenübergreifend einzusetzen. Hierzu werden mit Annotationen aus dem Neo4j Namespace versehene Klassen zur Entwicklungszeit automatisch um Methoden erweitert, die bestimmte Neo4j Techniken implementieren.

Es wird empfohlen, Neo4j per Spring[1] anzubinden und den Buildprozess per Maven zu verwalten. Im Prototyp wurde Neo4j Embedded zum Befüllen der Datenbank per Maven eingebunden. Die Steuerung und Instanziierung der Objekte wurde zu Demonstrationszwecken manuell übernommen.

4.2. Machbarkeit

Die vorangehenden Abschnitte stellen dar, dass Springdata Neo4j grundsätzlich dazu geeignet ist, Graphstrukturen abzubilden. Auch Kanten können als Java-Objekte be-

[1] Quelloffenes Framework zur Entkopplung der Applikationskomponenten

trachtet mit Eigenschaften versehen werden.

Um CIDOC-CRM vollständig abzubilden, müssen allerdings sämtliche Eigenschaften des Modells darstellbar sein. Das Konzept CIDOC-CRM beschreibt einen unidirektionalen, gerichteten Graphen mit benannten Kanten (Zur Erklärung siehe Abschnitt 2.3), der zusätzlich Schleifenverknüpfungen enthält. Es ergibt sich eine Baumstruktur, die durch Neo4j weitgehend problemlos abgebildet werden kann. Ein Problem ergibt sich jedoch, bei der Betrachtung der Vererbung der einzelnen Entitäten oder Relationen. Neo4j unterstützt dank POJO[1] zwar Vererbung, lässt jedoch durch die Bindung an Java keine Mehrfachvererbung zu. Da die einzelnen Neo4j-Entitäten zum Erfassen der Typsicherheit innerhalb des CRM-Modells mit Feldern für mögliche Kantentypen ausgestattet werden muss, kann eine Entität aus dem Domänenmodell durch die im CRM-Modell definierte Mehrfachvererbung keine Felder auf die zulässigen Relationentypen enthalten.

Codebeispiel 3: Beispiel typisierte Relationen in Entitätenklassen

```
@org.springframework.data.neo4j.annotation.RelatedToVia(
  type = "P126F_employed",
  direction = org.neo4j.graphdb.Direction.OUTGOING,
  elementClass = P126F_employed.class)

  private java.util.Set<P126F_employed> P126F_employed_OUT;
```

Codebeispiel 3 stellt die Annotation dar, mit der Relationen in Neo4j modelliert werden können. Der Code besteht aus einem Set, das die zukünftigen Relationen enthält und den Annotationen, die dieses Set als Neo4j Referenzfeld markieren. Dabei erklärt der Annotationsparameter `type` den Namen der Relation, `direction` die Richtung der gerichteten Verbindung und `elementClass` den Java-Klassentyp der Relation.

Es gibt folgende Lösungsansätze zu dem Vererbungsproblem:

Lösungsansatz 1: vollständige Auflösung der Vererbungen

Eine Neo4j Entitätsklasse enthält alle eigenen, durch CRM definierten möglichen Felder Relationen und zusätzlich alle Felder für Relationen aller Oberklassen. Dies ermöglicht eine große Präzision, hat jedoch eine große Anzahl an Feldern pro Klasse

[1] Plain old Java Objects

in absteigender Tiefe des CRM-Hierarchiebaums zur Folge. Ausserdem kann bei der Objektgenerierung die Klassenherkunft der Relationen nicht mehr nachvollzogen werden.

Lösungsansatz 2: Vereinfachen des Domänenmodells

Es werden sehr einfache Neo4j-Klassen generiert, die ohne Typisierung der Relationen auskommen. Die möglichen Relationen werden also erst beim Füllen des Objektmodells mit Realdaten ausgefüllt und sind bis dahin untypisiert. Dies ermöglicht auch Abweichungen vom Domänenmodell, denn es wird nicht festgelegt, welche Relationen mit welchen Entitäten verbunden werden können. Erst das Füllen eines String Identifiers mit dem Bezeichner der Relation ermöglicht die Wiedererkennung des Typs der Relation.

Lösungsansatz 3: Methoden zur dynamischen Prüfung der Oberklassen

In einem Post in einem Webforum zu einem ähnlichen Problem schlug Michael Hunger, ein Mitarbeiter der Neo4j Gruppe vor, eine algorithmische Prüfung der Relationen zu implementieren.

Eine Neo4j Entitätsklasse enthält dabei ausschließlich alle eigenen, durch CRM definierten möglichen Felder Relationen. Diese Klasse wird erweitert um eine Möglichkeit der Prüfung der Zulässigkeit der Verknüpfung einer Relation durch die Vererbung der Oberklassen. Es muss also eine Funktion implementiert werden, die bei dem Versuch der Verknüpfung einer Entität zunächst die eigenen zulässigen Relationen gegen das Domänenmodell prüft. Wenn die entsprechende Relation nicht gefunden wurde, müssen rekursiv die Oberklassen in der CRM Baumstruktur bis zur obersten Entität `E1.Entity` auf die angefragte Relation geprüft werden. Dies erfordert bei jedem Hinzufügen einer Verknüpfung eine Traversierung der CRM Struktur und entsprechende Fehlerbehandlung, wenn die Relation nicht gefunden wurde, bietet jedoch eine hohe und präzise Abdeckung des CRM Modells.

Da diese Methode jedoch einige der Neo4j Strukturen und Sicherheitsmechanismen teilweise übergeht und ersetzt, können bei nicht sachgemäßer Sicherung der API wie in Lösungsansatz 2 fehlerhafte CRM Modelle eingespeist werden.

4.3. Implementation

Um den Nodestore in Neo4j zu implementieren, ist es notwendig, zunächst die Entitäten und Relationen in Neo4j kompatible Java Klassen abzubilden. Dazu wurde mit Hilfe von SAX[1] ein XML-Parser implementiert, der die CIDOC Domänenenstruktur automatisiert in die entsprechende Struktur abbildet. Dies erfolgt durch einen JCodeModel[2] Codegenerator. Die derzeitige Implementation verfolgt den Lösungsansatz 1 aus Abschnitt 4.2, so dass sämtliche Eigenschaften der Parents der Entity in die Entity integriert werden.

4.3.1. Entities

Alle CIDOC-CRM Entities sind abgeleitet von einer abstrakten `BasicEntity`, die Grundfähigkeiten und für alle Entities notwendige Felder definiert, wie eine innerhalb der Neo4j Graphdatenbank eindeutigen ID oder einer pro Entity eindeutigen Nutzwertvariable. Ferner ist es notwendig, die `equals()` und `hashCode()` Methoden aus `Object` zu überschreiben, damit die Identität der Entitäten überprüft werden kann.

Codebeispiel 4: BasicEntity

```
@NodeEntity
public class BasicEntity {
  @GraphId
  @Indexed
  private Long ID;

  @Indexed
  private String value;

  public BasicEntity(){ }

  public BasicEntity(String value){
    this.value = value;
  }

  public String getValue(){
```

[1] Simple API for XML, de-facto Standard für das Parsen von XML
[2] Codegenerator Framework für Java

```
    return value;
  }

  public boolean equals(Object other) {
    if (this == other) return true;
    if (ID == null) return false;
    if (! (other instanceof BasicEntity)) return false;
    return ID.equals(((BasicEntity) other).ID);
  }

  \\hashcode based on ID
  transient private Integer hash;
  public int hashCode() {
     if (hash == null) hash = ID == null ?
        System.identityHashCode(this) : ID.hashCode();
     return hash.hashCode();
  }
```

Die für Neo4j benötigten Annotationen sind im Codebeispiel bereits angefügt. Speziell `@GraphId` und `@Indexed` definieren die Felder für die von Neo4j automatisch vergebenen Identifikationsschlüssel.

Eine konkrete Implementation einer Entity wird abgeleitet von der BasicEntity und angereichert mit den Feldern für die möglichen Verbindungen, die diese Entität eingehen kann. Dabei werden entsprechend Lösungsansatz 1 die Relationen der Parentklassen direkt in die Klasse integriert.

Codebeispiel 5: E18_Physical_Thing

```
@NodeEntity
public class E18_Physical_Thing
    extends BasicEntity
{
    public String Value;

    \\Eigene Relationen und Parent Relationen
    @RelatedToVia(type = "P13F_destroyed",
      direction = Direction.INCOMING,
      elementClass = P13F_destroyed.class)
    private Set<P13F_destroyed> P13F_destroyed_IN;
    @RelatedToVia(type = "P13B_was_destroyed_by",
```

```
    direction = Direction.OUTGOING,
    elementClass = P13B_was_destroyed_by.class)
  private Set<P13B_was_destroyed_by>
          P13B_was_destroyed_by_OUT;
  @RelatedToVia(type = "P24F_transferred_title_of",
    direction = Direction.INCOMING,
    elementClass = P24F_transferred_title_of.class)
  private Set<P24F_transferred_title_of>
          P24F_transferred_title_of_IN;

  \\ weitere Felder fuer Relationen ausgelassen
}
```

Die Annotationen `@RelatedToVia` erweitern die Annotation `@RelatedTo` um die Möglichkeit, den Typ der Relation anzugeben, also um den Parameter `elementClass`. Die möglichen annotierbaren Strukturen sind:

Klassen sind gleichbedeutend mit einer Einfachbeziehung in RDBMS,

java.util.Iterable bilden eine readonly Mehrfachbeziehung ab,

java.util.Set stellen eine von Neo4j veränderliche Mehrfachbeziehung dar.

4.3.2. Relations

Relationen werden ähnlich wie die Entitäten von einer gemeinsamen `BasicRelation` abgeleitet, die die für alle Relationen gültigen Methoden und Felder enthält. Da jedoch in den Entity-Klassen nicht nur Felder für die durch CIDOC-CRM definierten Relationen eingesetzt werden, sondern auch die Relationen durch die Parent-Klassen, müssen die Relationen mehrfach verwendbar sein. Das führt dazu, dass in den Klassen für die Relationen nicht die konkreten Entitäten eingesetzt werden können, sondern der abstrakte `BasicEntity` Klassentyp, der Oberklasse aller Entitätenklassen ist (siehe Codebeispiel 6).

Codebeispiel 6: BasicRelation

```
@RelationshipEntity
public class BasicRelation {
```

```
  @GraphId
  private Long ID;

  @Fetch
  @StartNode
  private BasicEntity startNode;

  @Fetch
  @EndNode
  private BasicEntity endNode;

  public BasicRelation() {}

  public void createRelation(BasicEntity startNode,
      BasicEntity endNode) {
    this.startNode = startNode;
    this.endNode = endNode;
  }

  public BasicEntity getStartNode(){
    return startNode;
  }
  public BasicEntity getEndNode(){
    return endNode;
  }
}
```

Die Implementation einer Relation enhält zusätzlich zur BasicEntity Felder für den Kurznamen und den vollständigen Namen der Relation, so dass diese einfach identifizierbar sind.

Codebeispiel 7: P102B__is__title__of

```
@RelationshipEntity
public class P102B_is_title_of
    extends BasicRelation
{
  public final String ShortName = "P102B";
  public final String LongName = "is_title_of";
}
```

4.3.3. Import von Daten

Ein Programm übersetzt mit Hilfe eines DOM Parsers und Java Codegenerators[1] das CIDOC-CRM Modell aus einer XML-Struktur in die Java-Klassen für die 83 Entity und 262 Relation Definitionen, die die einzelnen spezifischen Eigenschaften enthalten. Diese so generierten Java-Objekte können dann durch Neo4j verarbeitet werden.

Die so erstellten Java Klassen werden dann in ein neues Programm integriert. Dieses Programm führt sukzessiv für die 141 XML Metadaten-Testdateien ein Marshalling[2] der beschriebenen Objekte und Relationen in die Java Klassen durch, die daraufhin durch die Graphdatenbank transaktionsbasiert übernommen werden.

Dabei wird vorab durch eine Identitätsüberprüfung festgestellt, ob ein zu verknüpfendes Objekt bereits in der Datenbank (oder der Transaktion) enthalten ist. Wenn dem so ist, wird das neue Objekt mit diesem verlinkt statt neu erstellt.

4.3.4. Visualisierung

Abbildung 25 auf Seite 62 stellt ein Objekt mit einigen seiner mit ihm in den Testdaten verknüpften Metdaten dar[3]. Im Zentrum ist das Metaobjekt selbst erkennbar, also das Objekt, mit dem alle Metadaten verknüpft sind. Üblicherweise enthält es lediglich eine eindeutige Referenznummer, hier wird jedoch gleichzeitig die ID der Herkunftsdatenbank als Wert gespeichert. Sämtliche anhängenden Metadaten sind mit Properties versehen, die sowohl den Typ, als auch den Wert des Metadatums aufzeigen.

4.4. Abfragen

Abfragen können in Java oder Cypher verfasst werden. Cypher ist eine SQL-ähnlichen Abfragesprache, die zur Beschreibung von Abfragen für Graphdaten entwickelt wurde und sich im Neo4j Umfeld bewährt.

[1] JCodemodel
[2] Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht
[3] Zur Darstellung wird Neoclipse verwendet: https://github.com/neo4j/neoclipse

> ***Introduction to Cypher***
> *Cypher is a declarative query language for graphs, which uses graph pattern matching as a main mechanism for graph data selection (for both read-only and mutating operations). The declarative pattern matching nature of Cypher means that users can query the graph by describing what they need to get from the graph.*
>
> [PVW13, S. 64]

Cypher kombiniert einige Prinzipien von SQL[1] mit einer ASCII-Art ähnlichen Syntax zu Darstellung von Relationen[2]. Ausgehend von einem Startknoten, der entweder eindeutig per ID angegeben werden kann oder durch einen Index gefunden werden kann, werden Traversierungen definiert, die dann das gewünschte Ergebnis ausgeben.

4.4.1. Facetten

Die folgende Cypher Abfrage liefert durch eine Traversierung dreier Kanten bekannten Typs die Standorte aller Werke zurück.

Codebeispiel 8: Cypher: Facetten

```
//find place facet
start x=node:node_auto_index(__type__='E1_CRM_Entity')
match x-[:`P12B_was_present_at`]->
      ()-[:`P7F_took_place_at`]->
      ()-[:`P87F_is_identified_by`]->y
return distinct y.value,y.__type__,count(y)
```

Die Abfrage sucht zunächst als Startpunkt alle Knoten aus dem automatisch angelegten Knotenindex, die vom Typ „E1_CRM_Entity“ sind. Auf dieses Teilergebnis wird ein Prädikat angesetzt, das eine Traversierung über bestimmte Kantentypen erstellt, an dessen Ende jeweils ein Endknoten steht, dessen Werte ausgegeben werden.

[1] Structured Query Language
[2] http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

Codebeispiel 9: Cypher: Facetten Ergebnis

```
y.__type__,count(y),y.value
E44_Place_Appellation,2,Leipzig
E44_Place_Appellation,1,[Strassburg]
E44_Place_Appellation,1,[S.l.]
E44_Place_Appellation,1,[Mainz]
E44_Place_Appellation,1,Basel
E44_Place_Appellation,1,[Koeln]
E44_Place_Appellation,1,[Nuernberg]
E44_Place_Appellation,2,[Basel]
```

Das Ergebnis stellt also basierend auf der Anfrage die Städte dar, in denen die Werke physisch liegen. Diese Anfrage kann dazu benutzt werden, Facetten zu erstellen wie in Abschnitt 3.3.3 beschrieben. Das Ergebnis ist also eine direkte Lösung des Problems der selbst erstellbaren Facetten.

4.4.2. ShortestPath

Die kürzeste Verbindung zwischen zwei Knoten mit den IDs 1 und 128 kann in Cypher folgendermaßen angefragt werden:

Codebeispiel 10: Cypher: shortestPath

```
start x=node(1),y=node(128)
match p=allShortestPaths(x-[*..7]-y)
return rels(p),nodes(p)
```

Ausgehend von zwei bekannten Knoten findet die Anfrage die kürzesten Verbindungen zwischen den beiden Knoten, mit einer maximalen Pfadlänge von 7. Es werden also die kürzesten Pfade aufgezeigt, die eine Verbindung zwischen diesen beiden Knoten darstellen. Eine Anwendung dieses Prinzips ist in Abschnitt 3.3.4 beschrieben.

Codebeispiel 11: Cypher: shortestPath Ergebnis

```
rels(p),nodes(p)
"relId:0,relId:159","nodeId:1,nodeId:0,nodeId:128"
"relId:34,relId:187","nodeId:1,nodeId:18,nodeId:128"
```

Das Ergebnis ist eine Auflistung der traversierten Kanten (beginnend mit „relId“) und Knoten (beginnend mit „nodeId“) und ihren entsprechenden datenbankinternen Primärschlüsseln. Es kann demnach sehr einfach nachvollzogen werden, welche Pfade und Verbindungen den kürzesten Weg zwischen zwei definierten Knoten darstellen.

4.4.3. Subtree matching

Eine etwas komplexere Anfrage ist das Subtree matching, welches basierend auf bekannten Metadaten eine Filterung durchführt.

Codebeispiel 12: Cypher: Subtree matching

```
start x=node:node_auto_index(__type__='E1_CRM_Entity')
match
x-[:‘P102F_has_title‘]->y,
x-[:‘P2F_has_type‘]->z,
x-[:‘P67B_is_referred_to_by‘]->(),
x-[:‘P130B_features_are_also_found_on‘]->
  ()-[:‘P2F_has_type‘]->
  ()-[:‘P48F_has_preferred_identifier‘]->a,
x-[:‘P106F_is_composed_of‘]->
  ()-[:‘P72F_has_language‘]->
  ()-[:‘P1F_is_identified_by‘]->E41,
x-[:‘P128B_is_carried_by‘]->E73,
E73-[:‘P50F_has_current_keeper‘]->
  ()-[:‘P131F_is_identified_by‘]->E82,
E73-[:‘P49F_has_former_or_current_keeper‘]->
  ()-[:‘P131F_is_identified_by‘]->E82_2,
x-[:‘P2F_has_type‘]->
  ()-[:‘P1F_is_identified_by‘]->E41_2,
x-[:‘P12B_was_present_at‘]->
  ()-[:‘P4F_has_time_span‘]->E5,
E5-[:‘P82_at_some_time_within‘]->Time1,
E5-[:‘P82_at_some_time_within‘]->Time2
where y.value="Cosmographia Dans Manuductionem
  in tabulas Ptholomei - Ink.324.4:2"
and z.value="hierarchy_type:monograph"
and a.value="mediatype_003"
and E41.value="ger"
and E82.value="Sächsische Landesbibliothek -
```

```
  Staats- und Universitaetsbibliothek Dresden"
and E41_2.value="Monographie"
and Time1.value="time_61545"
and Time2.value="time_61500"
return id(x),y.value
```

Diese Abfrage durchsucht die Datenbank angefangen bei allen „Basisknoten“ nach einem bestimmten Objekt ohne eine ID zu kennen. Dennoch ist das Ergebnis potenziell eindeutig, da das Objekt anhand sämtlicher bekannter Metadatenverknüpfungen identifiziert wird. Das Prinzip und eine Anwendung werden in Abschnitt 3.3.6 beschrieben.

In der Abfrage ist zu erkennen, dass auch Mehrfachverzweigungen abgefragt werden:

Codebeispiel 13: Cypher: Subtree matching Ausschnitt

```
...
x-[:‘P12B_was_present_at‘]->
  ()-[:‘P4F_has_time_span‘]->E5,
E5-[:‘P82_at_some_time_within‘]->Time1,
E5-[:‘P82_at_some_time_within‘]->Time2
...
```

Die Abfrage verzweigt also über einen Knoten `E5` weiter zu den Knoten `Time1` und `Time2`, was ein sehr hohes Maß an Präzision bei der Suche eines Objekts ermöglicht.

Das Ergebnis ist eine Referenz auf ein einzelnes Objekt, dessen Titel ausgegeben wird.

Codebeispiel 14: Cypher: Subtree matching Ergebnis

```
id(x),y.value
1,Cosmographia Dans Manuductionem in tabulas Ptholomei
```

Die Abfragen zeigen, dass es möglich ist, die in den Lösungsansätzen in Abschnitt 3.3 dargestellten Vorschläge praktisch zu implementieren. Die in Cypher formulierten Abfragen sind durchaus geeignet, Objekte mit einer abstrakten Abfragesprache abzurufen. Gleichzeitig kann die gesamte CRM Strukur effizient in einer Neo4j Datenbank verwaltet werden.

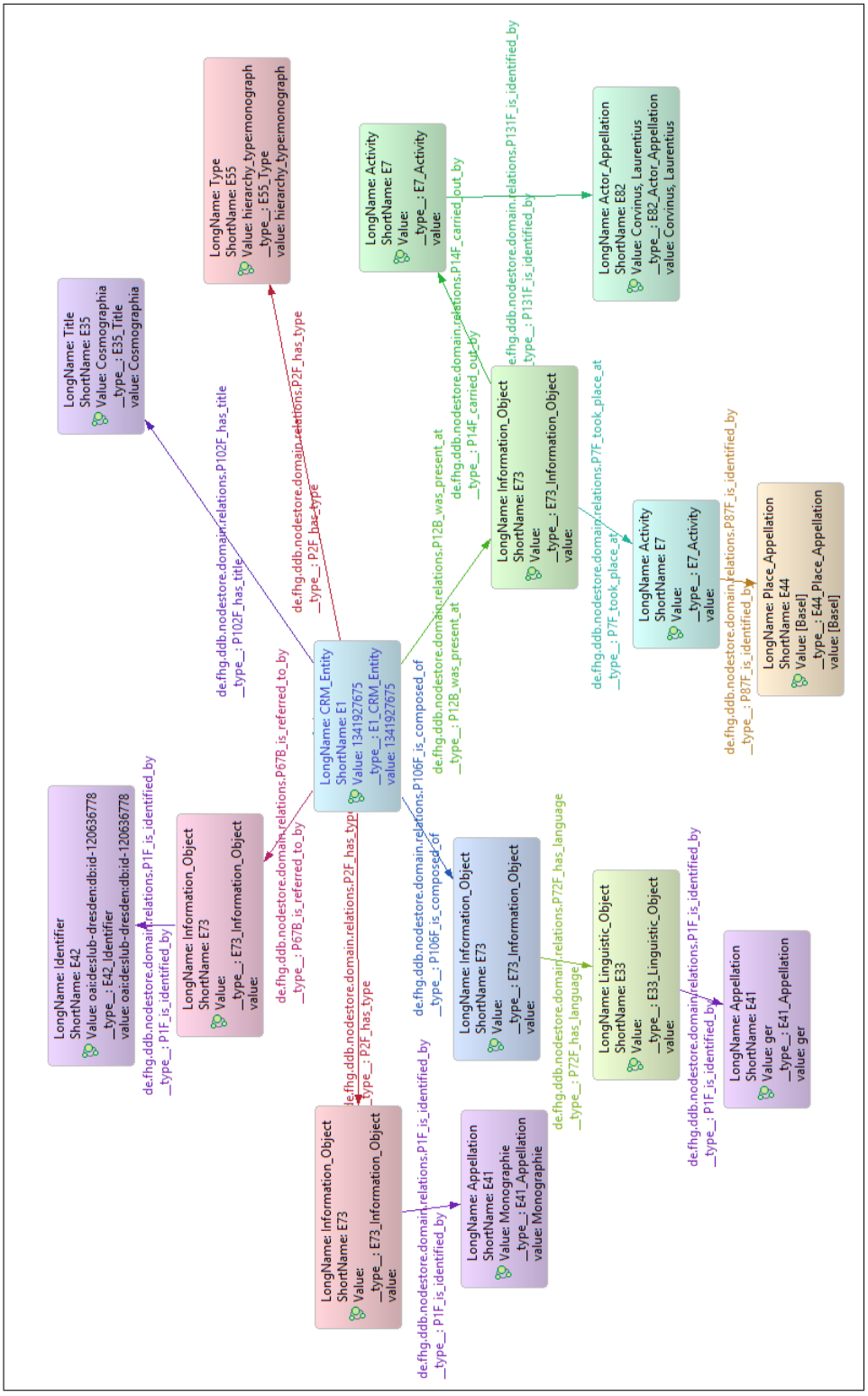


Abbildung 25: Beispielobjekt in Neoclipse

5. Ausblick und Fazit

Graphdatenbanken zeigen ein großes Potenzial beim effizienten Speichern frei strukturierter, aber stark verknüpfter Daten bei gleichzeitiger Verbesserung der Performanz von Abfragen und Traversierungen verglichen mit konventionellen Datenstrukturen. Diese Eigenschaften sind zum Teil darin begründet, dass die Kosten für die Verknüpfung der Daten zum Zeitpunkt der Importierung der Daten stattfinden und nicht - wie z.B. in relationalen Datenbanken üblich - in den Abfragen. Das zugrundeliegende Prinzip ist, dass die Daten nicht nur topologisch nahe beieinander liegen, sondern auch technisch, so dass die Abfragezeiten für Indexaufrufe um mehrere Komplexitätsklassen reduziert wird. Die Daten werden abhängig von der Entfernung physikalisch benachbart abgelegt mit der Folge, dass eine Traversierung von einem Knoten zum nächsten unabhängig von der Größe des Datensatzes in konstanter Zeit erfolgen kann.

Dies ermöglicht die Evaluierung und Berechnung gleichermaßen interessanter und komplexer Pfadkombinationen in Echtzeit. Eine Kombination mit Volltextindizes wie Lucene oder Solr macht darüber hinaus die Elemente innerhalb des Graphen auch abseits von Traversierungen effizient suchbar. Im Abschnitt 2.5 wurden dazu einige typische Problemstellungen diskutiert, deren Lösung Graphdatenbanken auszeichnen, wie beispielsweise die Identifikation von Adjazenzen.

Einige Beispiele, wie z.B. die in Abschnitt 2.4 besprochenen sozialen Netze, Verkehrsnetze oder auch semantische Wikis sind hierbei typische Anwendungen für Graphstrukturen, die besonders gut durch Graphdatenbanken berechnet werden können. Eine neue auf Graphen basierende Technologie sind die auf Hierarchien beruhenden Metadaten in kulturellen Einrichtungen.

Das CRM-Modell stellt beschrieben in Abschnitt 3.3.1 eine einheitliche Domänenstruktur für Metadaten zur Verfügung. Durch die hierarchische Topologie der Elemente innerhalb der CRM Domäne können kulturelle Objekte unabhängig von der Art der Ressource in Graphen ausgedrückt werden, so dass diese durch Graphenalgorithmen suchbar gemacht werden können.

Damit eine wie in Abschnitt 3.3.6 beschriebene Verbindung zwischen den einzelnen Metadatenobjekten hergestellt werden kann, führen intelligente Vergleichsalgorithmen (Abschnitt 3.3.5) eine Identitätsprüfung aus, in der festgestellt wird, ob ein Objekt zumindest teilweise in der Datenbank existiert. In diesem Fall werden die Objekte verlinkt statt

neu angelegt. Durch diesen Mechanismus entsteht ein Netzwerk, welches frei definierte facettierte Suchanfragen, horizontal skalierte Infrastrukturen und semantische Suchen ermöglicht. Zukünftige Weiterentwicklungen der Abfragen ([Ale11]) und Datenstrukturen ([TD11]) beschrieben, lassen tiefer strukturierte Analysen zu, so dass Semantiken der Datenverknüpfungen im Kontext des Semantic Web entstehen. Die Suchabfragen nähern sich damit weiter dem Nutzer an, dem damit ein wirksames Werkzeug zur kontextbezogenen Suche zur Verfügung steht.

Eine Kombination aus Graphenalgorithmen, Datenbankmechanismen und Volltextindizes ermöglicht es, Graphstrukturen zu erfassen und zu effizient verarbeiten. Derartige Mittel sind in der Graphdatenbank Neo4j implementiert, die wie in Kapitel 4 für die Prototypimplementation verwendet wurde. In de Kapitel wird beschrieben, wie eine CRM Domäneninfrastruktur in Java erstellt werden kann. Das Hauptproblem besteht darin, dass Java die durch CIDOC-CRM definierte Mehrfachvererbung nicht unterstützt. Dementsprechend kann eine so formulierte Struktur nicht ohne Informationsverlust in Neo4j abgebildet werden. Die im Prototyp implementierte Lösung entkoppelt die Klassenhierarchien, indem sämtliche Vererbungen aufgelöst, und die Referenzen vollständig in jede einzelne Klasse in der Hierarchie abgebildet wurden. Das ermöglicht zwar ein Beibehalten der Domänenstruktur, hat jedoch einen Informationsverlust zur Folge, da ein Objekt keine Referenz auf sein übergeordnetes Objekt hat. Obwohl diese Informationen für die in Abschnitt 4.4 beschriebenen Abfragen weitgehend irrelevant sind, kann eine testweise Implementation ohne forciertes Domänenmodell, dafür mit Unterstützung der Mehrfachvererbung interessant für noch zu entwickelnde Algorithmen sein, die diese Information verwenden können.

Die Neo4j Prototypimplementation demonstriert, dass in CRM strukturierte Kulturdaten ein großes Potenzial aufweisen, um die Abkehr von relationalen Datenbanken effizient zu gestalten. Die Sprache Cypher liefert ein probates Mittel, um programmiersprachenunabhängig frei definierte und von Java entkoppelte Abfragen wie in Abschnitt 4.4 zu erstellen, die dennoch eine hinreichende Präzision zulassen.

Obwohl das Potenzial für den Einsatz von Graphdatenbanken im Kontext kultureller Metadaten technisch gesehen sehr hoch ist, müssen speziell entwickelte Lasttests für einen industriell einsatzfähigen Prototypen beweisen, ob eine praktische Anwendung dieser Techniken sinnvoll ist oder ob eine Kombination aus mehreren Techniken zielführend ist.

A. Glossar

ACID (engl.: atomicity, consistency, isolation, durability) Voraussetzungen für die Verlässlichkeit von Datenbanksystemen. 7, 48
Adjazent mit dem Ausgangsknoten verbundener Nachbarknoten. 21
Apache Solr In Java implementierter Apache Volltextindex. 46
API application programming interface. 8

CIDOC-CRM CIDOC Conceptual Reference Model. 5, 16, 34, 40, 53
Cluster Gruppe zugehöriger Knoten mit hoher Zentralität untereinander und wenigen Verbindungen zu anderen Knoten. 8, 11, 24
Cortex Kernsoftware der Deutschen Digitalen Bibliothek. 46
Cypher Abfragesprache für in Graphstrukturen organisierte Datenquellen.. 48, 57

DDB Deutsche Digitale Bibliothek. 4, 27, 30, 34, 46
DMCI Dublin Core Metadata Initiative. 28
Dublin Core flaches, erweiterbares Metadatenformat. 28

Echtzeit Die Berechnung wird garantiert innerhalb einer gegebenen zeitlichen Beschränkung durchgeführt. 23

Fremdschlüsselrelation implizite Verknüpfung zweier Datensätze durch Datenbankverweis auf die ID. 8

IAIS Fraunhofer Fraunhofer Institut für Intelligente Analyse und Informationssysteme. 4, 27
Index Register zur Beschleunigung der Suche von Datensätzen innerhalb einer Datenbank. 6, 8, 21, 23, 30, 46, 48, 58

JCodeModel Codegenerator Framework für Java. 53
JOIN-Operation SQL Operation zur Abfrage von Daten zweier oder mehr Tabellen, basierend auf einer Verbindung einzelner Spalten. 4

KWE Kulturelle und wissenschaftliche Einrichtung. 4, 27

MARC-21 Machine-Readable Cataloging Standards zur Beschreibung von Metadaten in Bibliotheken. 28
Marshalling Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht. 57

Metadaten eine Ressource beschreibende Daten, nicht jedoch die Ressource selbst. 27, 28
Metadatenformat technisches Austauschformat für Metadaten. 28

NoSQL (engl.:"Not only SQL") Oberbegriff für nichtrelationale Datenbanken. 4, 7, 46

Pagerank Maß zur Gewichtung der Güte von Dokumenten anhand der Anzahl ihrer Verlinkungen. 8
Pfad traversierbare, zusammenhängende Struktur mehrerer Kanten und Knoten. 10
Pfadlänge Anzahl der traversierten Kanten zwischen zwei definierten Knoten. 10
POJO Plain old Java Objects. 51

RDBMS Relational Database Management System. 46
RDF Resource Description Framework. 39
Replikation mehrfache Speicherung derselben Daten an meist mehreren verschiedenen Standorten und die Synchronisation dieser Datenquelle. 24, 46
REST Representational State Transfer. 8, 48

SAX Simple API for XML, de-facto Standard für das Parsen von XML. 53
Semantic Web Semantic Web bezeichnet den Ansatz, Informationen für Maschinen semantisch berechenbar zu machen. 39
Skalierbarkeit Skalierbarkeit ist die Fähigkeit eines Systems, Netzwerks oder Prozesses, eine wachsende Anzahl an Aufgaben durch Wachsen des Systems zu bewältigen. 7, 46
Spring Quelloffenes Framework zur Entkopplung der Applikationskomponenten. 50
Springdata Neo4j Hochperformante Graphdatenbank in Java. 4, 47, 50
SQL Structured Query Language. 7, 58

Transaktion Möglichkeiten der Kapselung ganzer Sätze von Anfragen, die als im Fehlerfall als Ganzes rückgängig gemacht werden können. 7, 8, 46, 48
Traversierung Folgen eines Pfades zwischen zwei Knoten. 8, 14, 23, 24, 58
Tripel 3-Tupel, das im semantischen Web eine Subjekt-Prädikat-Objekt Satzstruktur abbildet. 39, 41, 42

B. Literaturverzeichnis

[Ale11] Alexiev, Vladimir: Implementing CIDOC CRM Search Based on Fundamental Relations and OWLIM Rules. (2011)

[Ali12] Alieva, Miyasat: Metadatenmapping im Projekt Deutsche Digitale Bibliothek anhand von Dublin Core, Lido und EAD. (2012), S. 80

[Cod70] Codd, E. F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387. http://dx.doi.org/10.1145/362384.362685. – DOI 10.1145/362384.362685. – ISSN 00010782

[CTS06] Chatzigeorgiou, Alexander ; Tsantalis, Nikolaos ; Stephanides, George: Application of graph theory to OO software engineering. In: *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*. New York and NY and USA : ACM, 2006 (WISER '06). – ISBN 1–59593–409–X, 29–36

[DG08] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce. In: *Communications of the ACM* 51 (2008), Nr. 1, S. 107. http://dx.doi.org/10.1145/1327452.1327492. – DOI 10.1145/1327452.1327492. – ISSN 00010782

[Fac12] Facebook ; Facebook.com (Hrsg.): *Key Facts: Statistics*. https://newsroom.fb.com/content/default.aspx?NewsAreaId=22. Version: 2012

[GR93] Gray, Jim ; Reuter, A.: *Transaction processing: Concepts and techniques*. San Mateo and Calif : Morgan Kaufmann Publishers, 1993. – ISBN 1–55860–190–2

[Hun12] Hunger, Michael: *Good Relationships: The Spring Data Neo4J Guidebook*. InfoQ, 2012. – ISBN 978–1–105–06556–9

[PB13] Pollack, Mark ; Brisbin, Jon: *Spring Data: The definitive guide*. Farnham : O'Reilly, 2013. – ISBN 978–1–449–32395–0

[PVW13] Partner, Jonas ; Vukotic, Aleksa ; Watt, Nicki: *Neo4j in Action: Early Access Edition*. Manning Publications Co, 2013. – ISBN 9781617290763

[RN04] Russell, Stuart ; Norvig, Peter: *Künstliche Intelligenz: Ein moderner Ansatz*. 1. München and Boston [u.a.] : Pearson Studium, 2004. – ISBN 3–8273–7089–2

[Sin11] Singh, S. K.: *Database systems: Concepts, design and applications*. 2. Delhi : Dorling Kindersley (India), 2011. – ISBN 978–8–131–76092–5

[SKL11] Styliaras, Georgios ; Koukopoulos, Dimitrios ; Lazarinis, Fotis: *Handbook of research on technologies and cultural heritage: Applications and environments*. Hershey and PA : Information Science Reference, 2011. – ISBN 978–1–60960–044–0

[SP12] Sakr, Sherif ; Pardede, Erric: *Graph data management: Techniques and applications*. Hershey and PA : Information Science Reference, 2012. – ISBN 978–1–61350–053–8

[TD11] Tzompanaki, Katerina ; Doerr, Martin: A New Framework for Querying Semantic Networks. In: *Technical Report: ICS-FORTH/TR-419, May 2011* (2011)

[WC05] Wilton, Paul ; Colby, John W.: *Beginning SQL*. Indianapolis and IN : Wiley Pub, 2005. – ISBN 978–0–764–57732–1

C. Abbildungsverzeichnis

D. Tabellenverzeichnis

Erklärung zur selbstständigen Abfassung der Bachelorarbeit

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Ort, Datum	Unterschrift