Graphendatenbanken

Sven Schönung
sven@schoenung.org

Hochschule Mannheim
Fakultät für Informatik
Paul-Wittsack-Straße 10
68163 Mannheim

Abstract. Das hohe Aufkommen von Daten und deren Strukturierung in Graphen stellen relationale Datenbanken im Web 2.0 vor Probleme, bei denen NoSQL Graphendatenbanken Abhilfe versprechen. Anhand der Graphendatenbank neo4j wird die Rechtmäßigkeit dieses Anspruchs beispielhaft untersucht. Dazu wird zunächst eine kurze Einführung in das neo4j Datenmodell gegeben. Es wird der Komfort der Manipulation und Abfrage von Graphendaten in neo4j und relationalen Datenbanken verglichen. Der Skalierungsansatz von neo4j wird vorgestellt und bewertet. Schließlich wird eine zusammenfassende Empfehlung für den Einsatz von Graphendatenbanken gegeben.

Das von Tim O'Reilly Web 2.0 getaufte Online-Zeitalter[TO04] stellt Entwickler und technische Systeme vor zahlreiche neue Herausforderungen, denen traditionelle relationale Datenbanksysteme teilweise nur schwer gewachsen sind. Zwei der Herausforderungen in dieser Hinsicht sind:

1. *Graphenstruktur der Daten.* Applikationen des Web 2.0 sind "social", d.h. sie setzen verstärkt auf die Vernetzung der Nutzer und stellen deren Aktivitäten und Interaktionen in den Vordergrund. Aus dieser Nutzerzentriertheit erwachsen graphentheoretische Problemstellungen, da sich die Nutzer und deren Daten vielfach in Graphen organisieren. Relationale Datenbanksysteme kommen mit solchen Strukturen nur schwer zurecht, was zu umständlichen und fehleranfälligen Lösungen führen kann.
2. *Hohes Datenaufkommen.* Dieselbe Nutzerzentriertheit führt zu einem vermehrten Aufkommen nutzergenerierter Daten. Vorhandene relationale Datenbanken skalieren dabei oftmals nicht in hinreichendem Maße, was zu Performanz- und Verfügbarkeitsproblemen führen kann.[EFHBB11]

Vor allem diese letzte Problematik hat Mitte der 2000er Jahre zum Aufkommen sogenannter NoSQL-Datenbanken geführt, die versuchen dem wachsenden Datenaufkommen mittels massiver Verteilung und Aufweichung der ACID-Garantien[1] beizukommen[EFHBB11]. Als eine spezielle Untergruppe der

[1] **A**tomicity, **C**onsistency, **I**solation, **D**urability. Die vier Eigenschaften klassischer Transaktionen in relationalen Datenbanken.

NoSQL-Datenbanken versprechen Graphendatenbanken zusätzlich den Umgang mit in Graphen organisierten Daten zu erleichtern.

Im Folgenden sollen anhand der Graphendatenbank neo4j die Vor- und Nachteile dieses Ansatzes gegenüber relationalen Datenbanken beispielhaft untersucht werden.

1 Grundlagen

neo4j ist eine von der Firma Neo Technology seit 2007 entwickelte, in Java implementierte, hochskalierende und hochverfügbare Graphendatenbank, die unter der GPL/AGPL 3.0 veröffentlicht wird.

1.1 Datenmodell

Relationale Datenbanken erfordern die Übertragung der Knoten und Kanten eines Graphenmodells in die Form von Datenbanktabellen. Im Gegensatz dazu unterstützt neo4j diese Graphenelemente nativ, was die Modellierung von in Graphen strukturierten Problemdomänen vereinfacht. Im Folgenden seien die wichtigsten Bestandteile des neo4j Datenmodells kurz beschrieben:

Knoten. Knoten bilden die grundlegenden Einheiten eines Graphen. Jeder Knoten ist in neo4j mit einer eindeutigen, fortlaufenden ID versehen. Knoten können optional auch Eigenschaften besitzen, bei denen es sich um einfache Schlüssel/Werte-Paare handelt. Diese müssen keinem vorgegebenen Schema folgen, d.h. die verwendeten Schlüssel können sich von Knoten zu Knoten unterscheiden. Dies stellt eine erhebliche Flexibilisierung im Vergleich zu relationalen Datenbanken dar, welche für jeden Knotentyp eine eigene Tabelle erfordern würden.

Ein spezieller Knoten in neo4j ist der sogenannte Referenzknoten. Er existiert in jeder neu angelegten neo4j Datenbank und stellt einen allgemein bekannten Einstiegspunkt für jeden Graphen dar[AN10]. Es wird geraten jeden Knoten zumindest indirekt mit diesem Knoten zu verbinden, so dass jeder Knoten der Datenbank vom Referenzknoten aus erreichbar ist.

Kanten. Kanten verbinden Knoten und stellen die Auffindbarkeit derselben in der Datenbank sicher. Wie Knoten auch können sie eindeutig über eine ID identifiziert werden. Zu jeder Kante gehöhrt zwingend ein Start- und ein Endknoten, sowie ein Typbezeichner.

Kanten sind immer gerichtet. Die Tatsache, dass X Fan von Y ist, heißt noch lange nicht, dass Y auch Fan von X ist. Sollte die Richtung der Verbindung keine Rolle spielen, kann sie beim Traversieren des Graphen auch ignoriert werden.

Wie Knoten auch können Kanten optional schemalose Eigenschaften in Form von Schlüssel/Werte-Paaren enthalten.

Pfade. Pfade sind die Aneinandereihung mehrerer Knoten anhand von Kanten und vor allem deshalb wichtig, weil sie häufig das Ergebnis von Suchanfragen an die Datenbank darstellen.

Indizes. Knoten und Kanten können jeweils anhand ihrer Eigenschaft indiziert werden. Suchanfragen wie "gib' mir alle Knoten, deren Vorname 'Bernd' ist", können auf diese Weise wesentlich schneller abgearbeitet werden als wenn der gesamte Graph nach passenden Knoten durchsucht werden müsste.

1.2 Datenintegrität

Anders als viele andere Datenbanken aus dem NoSQL-Umfeld, die häufig zugunsten der Skalierbarkeit auf Garantien bezüglich ACID verzichten[EFHBB11], ist neo4j vollständig transaktionsfähig und weitgehend ACID-konform[2]. neo4j steht relationalen Datenbanken in dieser Hinsicht also fast in nichts nach.

neo4j garantiert zudem bestimmte Formen der Datenintegrität. So können zum Beispiel nur Kanten erstellt werden, die sowohl einen Anfangs- als auch einen Endknoten haben. Knoten können nur gelöscht werden, wenn an ihnen keinerlei Kanten mehr hängen. Was an Datenintegrität in relationalen Datenbanken üblicherweise erst über Fremdschlüssel festgelegt werden müsste, ist in neo4j also von Hause aus bereits garantiert.

Dennoch muss gesagt werden, dass die Schemalosigkeit von neo4j natürlich auch Gefahren für die Integrität der Daten mit sich bringt. So ist es problemlos möglich Daten eines unangebrachten Typs oder unter falschem Namen in einem Knoten zu speichern. Eine relationale Datenbank würde dies nicht erlauben.

2 Datenmanipulation und Datenabfrage

2.1 Embedded vs. Server

neo4j bietet zwei verschiedene Arten des Betriebs an (siehe Fig 1.). Zum einen kann neo4j "embedded" betrieben werden, das heißt als Teil einer Applikation mit direktem Zugriff auf die auf der Festplatte befindliche Datenbank; zum anderen als ein Server zu dem man sich als Client verbindet, ähnlich wie man dies von traditionellen relationalen Datenbanksystemen her gewohnt ist.

Da der neo4j Server prinzipiell auf neo4j Embedded fußt, sei dieses im Folgenden zunächst näher beschrieben.

2.2 Embedded

Zugriff auf die Datenbank erfolgt im Embedded-Modus immer über eine Instanz des `GraphDatabaseService`:

[2] Für die einzige Ausnahme siehe Abschnitt 3.2.

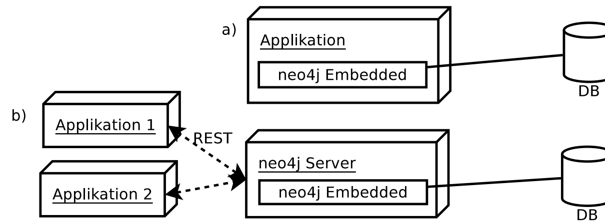

Fig. 1. Embedded (a) vs. Server (b)

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().
                                   newEmbeddedDatabase("target/dir");
```

Datenerzeugung

Über den `GraphDatabaseService` lassen sich Knoten in der Datenbank erzeugen und zu einem Index hinzufügen:

```
Node fred = graphDb.createNode();
fred.setProperty("name", "Fred Foobar");
fred.setProperty("age", 42);

IndexManager index = graphDb.index();
Index<Node> persons = index.forNodes( "persons" );
persons.add(fred, "name", fred.getProperty("name"));
```

Die Erzeugung von Kanten ist ähnlich einfach:

```
public enum RelTypes implements RelationshipType
  { FRIENDS, ENEMIES }

Relationship rel = fred.createRelationshipTo(bernd, RelTypes.FRIENDS);
rel.setProperty("since", "2008");
```

Im Allgemeinen ist das Anlegen, Manipulieren und Löschen von Graphendaten über neo4j etwas einfacher als es im Falle der Benutzung einer relationalen Datenbank in Verbindung mit einer ORM[3]-Bibliothek (z.B. Hibernate) wäre, da man auf das Erstellen eigener Klassen zur Repräsentation der Datenbanktabellen und deren Anpassung bei Anforderungsänderungen verzichten kann.

Datenabruf

Grundlage des Datenabrufs in neo4j ist die Traversal-API, die das Traversieren, d.h. das Durchsuchen des Graphen anhand bestimmter Kriterien, ermöglicht.

Zur Durchsuchung des Graphen bedarf es dabei fünf verschiedener Angaben:

1. *Startknoten.* Gibt den Knoten an von dem aus die Suche gestartet werden soll.
2. *Expander.* Gibt an welchen Kanten bei der Suche gefolgt werden soll (z.B. nur Kanten vom Typ FRIENDS) und in welche Richtung diese Kanten zeigen sollen (z.B. nur ausgehende Kanten).

[3] **O**bject **R**elational **M**apping

3. *Order.* Gibt die Reihenfolge der Abarbeitung der Knoten an (Tiefensuche bzw. Breitensuche).
4. *Uniquiness.* Gibt an ob Knoten oder Kanten mehrfach besucht werden dürfen.
5. *Evaluator.* Gibt an wann genau die Suche abbrechen soll, z.B. wenn ein Knoten mit einer bestimmten Eigenschaft oder einer bestimmten Tiefe gefunden wurde.

Die folgende Abfrage würde z.B. alle Pfade vom Knoten `fred` zu den Freundes-Freunden von `fred` liefern:

```
Iterator<Path> paths = Traversal.description()
   .expand(new OrderedByTypeExpander().add(RelType.FRIENDS))
   .order(CommonBranchOrdering.PREORDER_BREADTH_FIRST)
   .uniqueness(Uniqueness.NODE_GLOBAL)
   .evaluator(Evaluators.atDepth(2))
   .traverse(fred).iterator();
```

Insgesamt gestaltet sich die Abfrage von Graphendaten in neo4j wesentlich einfacher als dies bei einer relationalen Datenbank der Fall wäre. Hier müsste das Traversieren des Graphen in der Geschäftslogik der Applikation per Hand implementiert werden, was fehleranfällig und zeitaufwändig ist. neo4j nimmt einem diese Arbeit ab.

2.3 Server

Neben dem Embedded-Modus bietet neo4j auch einen eigenen Server. Dieser kapselt lediglich die durch neo4j Embedded angebotenen Funktionalitäten und stellt sie über eine REST-Schnittstelle zu Verfügung (siehe Fig. 1).

REST-API

Die REST-API von neo4j nutzt JSON als Kommunikationsformat. Die API ist grundsätzlich nach dem von Roy Fielding definierten REST-Prinzip des HATEOAS[4] aufgebaut, d.h. zurückgelieferte Repräsentationen beinhalten alle Informationen, die ein Client benötigt um verwandte Daten und zusätzlich mögliche Operationen aufzufinden. So liefert eine `GET`-Anfrage an den zentralen Einstiegspunkt in die API, (`http://localhost:7474/db/data/`) folgende Antwort:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "cypher" : "http://localhost:7474/db/data/cypher",
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
  "node" : "http://localhost:7474/db/data/node",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "neo4j_version" : "1.8.M04-1-g892e348",
  "batch" : "http://localhost:7474/db/data/batch",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "node_index" : "http://localhost:7474/db/data/index/node",
```

[4] **H**ypertext **A**s **T**he **E**ngine **O**f **A**pplication **S**tate[RF00]

```
    "reference_node" : "http://localhost:7474/db/data/node/31",
  "extensions" : {
  }
}
```

Ein Client kann auf diese Weise z.B. erfahren, dass Operationen auf Knoten (`"node"`) unter der URL `http://localhost:7474/db/data/index/node` möglich sind, ohne dass diese URL dem Client vorher bekannt sein muss. Dies entkoppelt die Entwicklung von Client und Server, da sich das URL-Schema des Servers ändern kann, ohne dass eine Änderung des Clients nötig wird.

Die REST-API des Servers stellt genau die gleiche Funktionalität bereit, welche die Traversal-API im Embedded-Modus über `GraphDatabaseService` zur Verfügung stellt. So legt z.B. folgender Request:

```
POST /db/data/node HTTP/1.1
Host: localhost:7474
Content-Type: application/json
Accept: application/json

{ "name": "bernd" }
```

einen neuen Knoten mit der Eigenschaft `name=bernd` auf dem Server an. Ebenso sind das Löschen, Ändern und das Traversieren des Graphen über die REST-API möglich.

Die Kommunikation über eine REST-Schnittstelle mit dem Server unterscheidet sich stark vom Ansatz der relationalen Datenbanken, die als Kommunikationsprotokoll überwiegend Eigenentwicklungen einsetzen, hat jedoch den Vorteil, dass sich die Entwicklung von Client-Bibliotheken aufgrund der standardisierten Schnittstelle wesentlich einfacher gestaltet.

Client-Bibliotheken für die REST-API

neo4j selbst bietet jedoch momentan für keine Programmiersprache eine eigene Client-Bibliothek an[5], welche die vom Server angebotene REST-Schnittstelle in der neo4j Traversal-API kapseln und zur Verfügung stellen würde. Glücklicherweise gibt es eine Reihe von Implementierungen von Drittparteien, die sich genau dieses Problems angenommen haben. Die für Java ausgelegte Bibliothek *Java-Rest-Binding*, macht es zum Beispiel denkbar einfach die REST-API aus Client-Code heraus anzusprechen:

```
GraphDatabaseService gds =
  new RestGraphDatabase("http://localhost:7474/db/data");
```

Auf diese Weise stehen die gewohnten Methoden der neo4j Traversal-API über den `GraphDatabaseService` zur Verfügung.

Nachteile dieser Bibliothek sind jedoch zum einen der schwerwiegende Mangel an Unterstützung für Transaktionen, sowie die fehlende Integration in die

[5] Die besprochene Bibliothek *Java-Rest-Binding* findet sich zwar auf der offiziellen *github* Seite des Projekts, gehört jedoch nicht zur Standardauslieferung von neo4j.

Standard-Auslieferung von neo4j. So muss die Bibliothek zunächst per Hand über einen Maven-Build erzeugt und in das eigene Projekt integriert werden. Bei der Entwicklung einer Beispielapplikation hat sich dies aufgrund fehlerhafter Abhängigkeiten zunächst als problematisch, wenn auch lösbar erwiesen.

Im Gegensatz dazu bieten Relationale Datenbanken wie mySQL seit jeher voll funktionsfähige Konnektor-Bibliotheken in verschiedenen Programmiersprachen für ihre Datenbanksysteme an. neo4j hat in diesem Bereich also noch gleichzuziehen.

Datenbankmanagementsystem

Verwaltung von und Zugriff auf relationale Datenbanken wird über relationale Datenbankmanagementsysteme (RDBMS) geregelt. Auf diese Weise ist es möglich mit einem Datenbankserver mehrere logisch voneinander getrennte Datenbanken zu verwalten.

neo4j kennt eine solche Trennung nicht. Jedem neo4j Server ist genau eine Datenbank zugeordnet. Dies erhöht den Administrationsaufwand bei mehreren gleichzeitig zu pflegenden Datenbanken im Vergleich zu relationalen Datenbanken. Zudem ist das Rechtemanagement in neo4j wesentlich rudimentärer als das in RDBMS der Fall ist.

2.4 Cypher Query Language

Die bisherigen Beispiele zeigten wie mit neo4j programmatisch, in Form von Java-Code gearbeitet werden kann. Eine der großen Stärken relationaler Datenbanken ist die domänenspezifische Sprache SQL in der Abfragen an die Datenbank gestellt werden können.

neo4j stellt eine eigene Abfragesprache namens Cypher zur Verfügung, mittels derer sowohl Knoten und Kanten manipuliert, als auch Graphen traversiert werden können. Folgende Anfrage z.B. findet alle Freundes-Freunde zu einem Knoten mit Namen "Bernd" in einem Graphen:

```
start n=node:persons(name="Bernd") match n-[:FRIENDS]-m-[:FRIENDS]-x return x
```

Anders als bei SQL handelt es sich bei Cypher jedoch um eine Speziallösung für neo4j. Während bei SQL zumindest der Kern der Sprache mit jedem beliebigen relationalen Datenbanksystem verwendbar ist, lassen sich Abfragen in Cypher bisher für keine andere Graphendatenbank wiederverwenden.

3 High Availability

Die Enterprise-Edition von neo4j kommt mit der Möglichkeit des Aufbaus eines High-Availability-Clusters daher. Dieser versucht zwei verschiedenen Zielen gerecht zu werden:

1. *Fehlertoleranz.* Das Zusammenschließen mehrerer neo4j Server in einem Cluster soll das Gesamtsystem resistent gegenüber dem Ausfall einzelner Server machen und so eine hohe Verfügbarkeit garantieren.
2. *Horizontale Skalierbarkeit.* Der neo4j Cluster erlaubt das Hinzufügen zusätzlicher Server um mehr Leseanfragen abwickeln zu können, als dies eine einzelne neo4j Instanz erlauben würde. Er ist damit auf die Anforderungen des Web ausgerichtet, wo der Leseaufwand einer Datenbank den Schreibaufwand oftmals um ein Vielfaches übersteigt.

3.1 Architektur

neo4j High Availability folgt einer Master-Slave-Architektur, bei welcher der Großteil der Schreiblast von einer Master-Instanz getragen wird, während die Slaves für das Abarbeiten von Leseanfragen zuständig sind.

Das Verteilen der Last übernimmt ein vorgeschalteter Load-Balancer, während Apache Zookeper die Koordinierung der neo4j Instanzen und die Wahl des Master-Servers übernimmt. Siehe Fig. 2. für eine Übersicht über die Architektur.

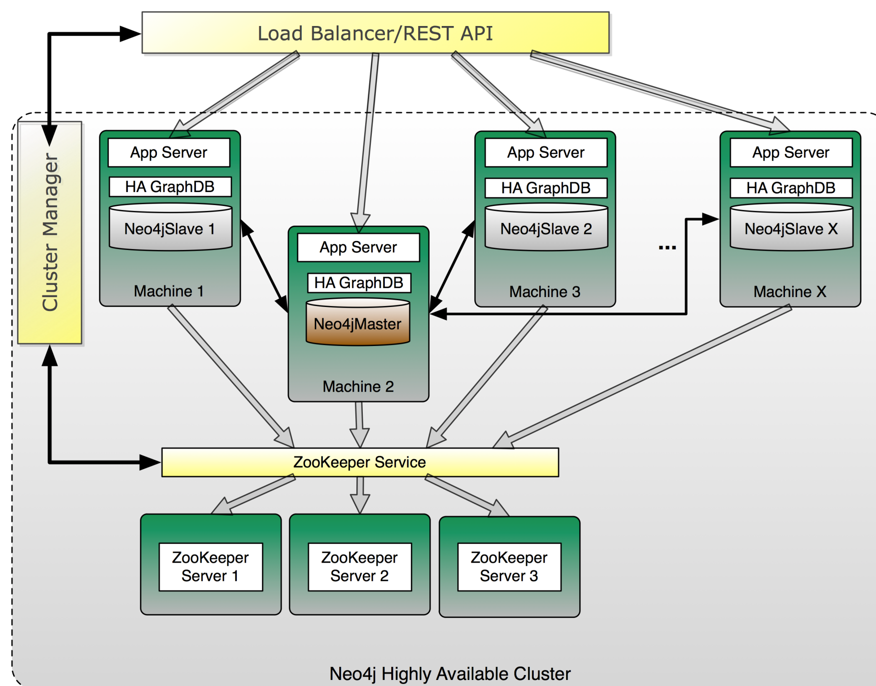

Fig. 2. neo4j High-Availability-Architektur[NM12]

3.2 Szenarien

Im Folgenden seien eine Reihe von Szenarien durchgespielt, die die Arbeitsweise des High-Availability-Clusters verdeutlichen:

Weiteren Server hinzufügen. Ein neu gestarteter Server meldet sich beim ZooKeeper an. Ist noch kein anderer Server dort angemeldet, ernennt der ZooKeeper den Server zum Master, ansonsten zum Slave. Via Cluster-Manager informiert der ZooKeeper den Load-Balancer über den neuen Server, so dass diesem von nun an Anfragen zukommen.

Slave fällt aus. Der ZooKeeper bemerkt den Ausfall und notifiziert den Load-Balancer, so dass keine Anfragen mehr an den Slave gesendet werden.

Master fällt aus. Der ZooKeeper bemerkt den Ausfall innerhalb von Sekunden und wählt unter den vorhandenen Slaves einen neuen Master aus. Der Load-Balancer wird notifiziert, so dass keine Anfragen mehr an den alten Master gesendet werden.

Schreibanfrage an Master. Diese können vom Master sofort durchgeführt werden. Der Master benachrichtigt die Slaves über diese Änderung jedoch nicht. Stattdessen müssen sich die Slaves in regelmäßigen Abständen mit dem Master synchronisieren. Die sich dabei ergebende zeitweilige Inkonsistenz zwischen Master und Slaves stellt die einzige Verletzung der ACID-Garantien in neo4j dar.

Schreibanfrage an Slave. Vor jedem Schreiben muss sich ein Slave mit dem Master synchronisieren. Darauf hin wird sowohl auf dem Master als auch dem Slave ein Lock gesetzt und der Slave sendet die Schreibanfrage an den Master. Ist die Transaktion auf dem Master abgeschlossen schreibt der Slave die Änderung in den eigenen Datenbestand.

Leseanfrage an Slave. Leseanfragen können von jedem Slave zu jedem Zeitpunkt beantwortet werden. Es gibt jedoch keine Garantie, dass die Datenbasis des Slave 100% konsistent mit der des Masters ist.

Leseanfrage an Master. Leseanfragen an den Master liefern immer die aktuellsten Daten.

3.3 Bewertung

Im Allgemeinen kann gesagt werden, dass sich der Verfügbarkeits- und Skalierungsansatz von neo4j (mit Ausnahme der teilweisen Schreibfähigkeit der Slaves) nicht wesentlich von der Master-Slave Replikation traditioneller relationaler Datenbanksysteme unterscheidet.

Dennoch kann davon ausgegangen werden, dass neo4j bei hohem Aufkommen von Graphendaten besser skaliert als dies bei einer relationalen Datenbank der Fall wäre, da die dort üblicherweise angewandte Datennormalisierung beim Traversieren des Graphen zu einer hohen Anzahl an kostspieligen JOIN-Operationen führt, die bei einer Graphendatenbank nicht notwendig sind[EE09].

4 Fazit

neo4js Stärke liegt vor allem in der Einfachheit des Traversierens von Graphenstrukturen, was im Vergleich zu relationalen Datenbanken eine wesentliche Reduzierung des Entwicklungsaufwandes und der Fehleranfälligkeit zur Folge hat. Die Flexibilisierung der Datenmodellierung aufgrund der Schemalosigkeit der Datensätze ist gerade bei sich häufig ändernden Anforderungen von Vorteil. Beides prädestiniert neo4j für Entwicklungsmethodologien wie Rapid Prototyping, die sich zeitnahe lauffähige Systeme zum Ziel setzen. Insbesondere für die schnelllebige Welt des Web, in der kontinuierliche Weiterentwicklung und Auslieferung von besonderer Bedeutung sind, bietet sich neo4j an, gerade auch aufgrund der besseren Skalierungseigenschaften im Umgang mit Graphendaten als dies bei relationalen Datenbanken der Fall ist.

Dennoch ist zu beachten, dass es sich bei neo4j und allen Graphendatenbanken im Vergleich immer noch um recht junge Projekte handelt, was, wie am Beispiel der Client-Bibliotheken zu sehen, gewisse Unausgereiftheiten nach sich zieht. Zudem bedeutet, aufgrund der fehlenden Standardisierung zwischen den Graphendatenbanken (z.B. bei der Abfragesprache), eine Entscheidung für neo4j eine stärkere Bindung an ein spezifisches Produkt als dies bei relationalen Datenbanken der Fall wäre. So fällt die Empfehlung für neo4j im Rahmen dieser Arbeit zwar durchaus positiv aus, aber eben nicht ohne Einschränkungen.

References

[AN10] Andres Nawrot. *Modeling categories in a graph database.* `http://blog.neo4j.org/2010/03/modeling-categories-in-graph-database.html`. Abgerufen am 21.06.2012.

[EE09] Emil Eifrem. *Neo4j - The Benefits of Graph Databases.* Vortag auf der O'Reilly Open Source Convention, San Jose, California, 23.07.2009. `http://www.oscon.com/oscon2009/public/schedule/detail/8364`. Abgerufen am 21.06.2012.

[EFHBB11] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer, Markus Brückner. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Hanser. 2011.

[NAPI12] Neo Technology. *Neo4j Community API.* `http://api.neo4j.org/1.7/`. Abgerufen am 21.06.2012.

[NM12] Neo Technology. *The Neo4j Manual v1.7.* `http://docs.neo4j.org/chunked/stable/`. Abgerufen am 21.06.2012.

[RF00] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* Dissertation, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. Abgerufen am 21.06.2012.

[TO04] Tim O'Reilly. *What Is Web 2.0. Design Patterns and Business Models for the Next Generation of Software..* 2004. `http://oreilly.com/web2/archive/what-is-web-20.html`. Abgrufen am 21.06.2012.