


Fachartikel: Praktischer
Einstieg in Neo4j

*Veröffentlichung
im Java aktuell
Ausgabe - 03/2017*



„Die wahre Stärke von Neo4j ist ihr Graphenmodell und die Tatsache, dass sie vollständig transaktional ist. Sie folgt, im Gegensatz zu vielen NoSQL-Datenbanken, dem ACID-Konzept.“

Inhaltsverzeichnis

- 01 Einleitung
- 02 Praktischer Einstieg in Neo4j
- 03 Installation
- 04 Community und Enterprise
- 05 Import von Daten
- 06 Erweiterbarkeit
- 07 Integration in Anwendungen
- 08 Fazit



➤ 01 Einleitung: Quasi jede Anwendung verwendet eine (oder mehrere) Datenbanken für das Persistieren von Daten. Dabei wird das fachliche Domänen-Modell auf die Datenbank abgebildet. Traditionell kommen dabei relationale Datenbanken zum Einsatz. Es gibt entsprechend viele Erfahrungen damit, welche Stärken und Schwächen sie haben. In den letzten Jahren sind mehr als 200 Datenbanken im NoSQL-Bereich aufgetaucht [1], darunter auch Neo4j, eine Graph-Datenbank. Im Gegensatz zu relationalen Datenbanken, die auf der Mengenlehre basieren, stützt sich Neo4j auf die Graphentheorie. Dieser Artikel zeigt, welche Auswirkungen das auf die Praxis hat.

02 Praktischer Einstieg in Neo4j

Auf den ersten Blick ist das Konzept von Neo4j sehr einfach: Es gibt nur Nodes (Knoten) und Relationships (Kanten). Knoten können für sich allein existieren, Kanten müssen immer an einem Knoten anfangen und wieder an einem (auch demselben) enden. Dabei können Knoten ein Label haben, etwa „Person“ oder „Group“, während Kanten einen Typ haben; letztere werden üblicherweise großgeschrieben. Kanten haben eine explizite Richtung. Beide (Knoten und Kanten) haben Properties (Eigenschaften). Der große Unterschied zur relationalen Datenbank ist, dass die Relationen explizit sind und nicht implizit als Werte in den Tabellenspalten sitzen.

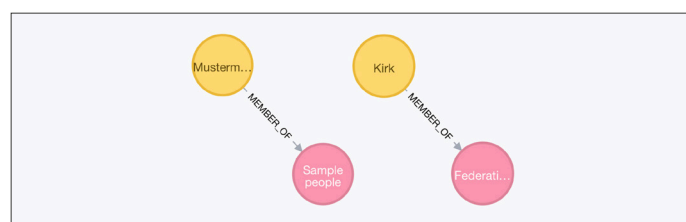
Abbildung 1 zeigt die Tabelle „user“, die mit der Spalte „gid“ auf die Tabelle „group“ verweist. Tabellen vermischen also ihre fachlichen Daten (wie „name“) mit technischen. Die Relation ist schwer zu sehen. Das gleiche Beispiel sieht bei Neo4j wie in Abbildung 2 aus.

```
HeriADB (test) > select * from user;
+----+-----+
| id | name  | gid |
+----+-----+
| 1  | Max   | 42  |
| 2  | Tiberius | 43  |
+----+-----+
2 rows in set (0.00 sec)

HeriADB (test) > select * from group;
+----+-----+
| id | name          |
+----+-----+
| 42 | Sample people |
| 43 | Federation of Planets |
+----+-----+
2 rows in set (0.00 sec)
```

➤ Abbildung 1: Implizite Beziehungen in einer relationalen Datenbank

Um die Knoten und Kanten zu erzeugen, ist ein Befehl in der Sprache Cypher notwendig (siehe Listing 1). Sowohl Knoten als auch Kanten haben beliebige Properties (siehe Abbildung 3).



➤ Abbildung 2: Explizite Relationen (Relationships in Neo4j)

Mit diesem Modell lassen sich relationale Daten problemlos in Neo4j importieren und weiterverarbeiten. Häufig gibt es das Missverständnis, dass Neo4j Daten auch nur grafisch repräsentieren kann, was nicht

```
create (g:Group {name:"Sample people"})
create (g:Group {name:"Federation of Planets"})
create (p:Person {firstNames:["James","Tiberius"],lastName:"Kirk"})
create (p:Person {firstNames:["Max"],lastName:"Mustermann"})
match (p:Person {lastName:"Kirk"}),(g:Group {name:"Federation of Planets"}) create (p)-[:MEMBER_OF]->(g)
match (p:Person {lastName:"Mustermann"}),(g:Group {name:"Sample People"}) create (p)-[:MEMBER_OF]->(g)
```

➤ Listing 1

```
$ match (g:Group)--(p:Person) return g,p
```

g	p
<div>name Federation of Planets</div>	<div>lastName Kirk</div> <div>firstNames [James, Tiberius]</div>
<div>name Sample people</div>	<div>lastName Mustermann</div> <div>firstNames [Max]</div>

Returned 2 rows in 32 ms.

➤ Abbildung 3: Properties von Knoten

der Realität entspricht. Letztendlich ist Neo4j eine Datenbank und die Werte der Eigenschaften lassen sich ganz normal tabellarisch ausgeben (siehe Abbildung 4). Eine Anwendung, die auf Neo4j basiert, kann für den Anwender ganz normal aussehen, allerdings gibt es natürlich auch die Option, dem Anwender eine grafische Repräsentation der Daten anzubieten. Darüber hinaus gibt es noch wesentlich komplexere Abbildungsmöglichkeiten, die für die Modellierung eine ausschlaggebende Rolle spielen:

- Ein Knoten kann beliebig viele Labels enthalten, eine Person kann also zum Beispiel auch gleichzeitig ein Mitarbeiter sein.
- Auch Kanten können beliebige Properties haben, etwa „since“, um darzustellen, ab wann jemand Mitglied ist.
- Kanten haben zwangsweise eine Richtung, A → B ist etwas anderes als A ← B. Die Richtung spielt für die Abfrage und die Modellierung eine große Rolle, hat aber auf die Antwortzeiten keine Auswirkung.
- Zwischen zwei Knoten können beliebig viele Kanten existieren und jede Kante kann ein beliebiger Typ sein.
- Die Werte von Properties können einzeln existieren oder ganze Listen sein.

```
$ match (g:Group)-[:MEMBER_OF]-(p:Person) return g.name,p.firstNames,p.lastName
```

g.name	p.firstNames	p.lastName
Federation of Planets	[James, Tiberius]	Kirk
Sample people	[Max]	Mustermann

➤ Abbildung 4: Tabellarisches Ergebnis

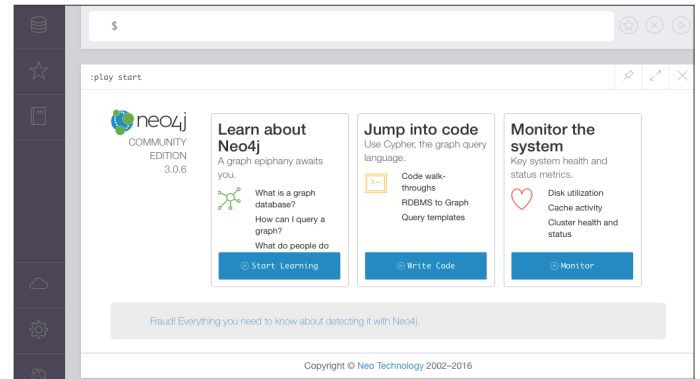
Dadurch ergeben sich völlig neue Möglichkeiten der Modellierung, die weit über das hinausgehen, was mit einer relationalen Datenbank möglich ist. In der Praxis ist es daher sinnvoll, dass die Modellierung an einem Flipchart startet, und zwar nicht auf Entitäten-, sondern auf Objekt-Ebene. Dabei sollte man auf Werkzeuge wie UML verzichten, da diese die Möglichkeiten erst gar nicht abbilden können.

Wenn die ersten Daten in Neo4j abgebildet sind, kann der fachliche Mitarbeiter sie sofort sehen, verstehen und bestätigen, ob das fachliche Modell richtig abgebildet ist. Durch die freie Wahl der Richtung von Kanten kann das Graphenmodell den Vorgaben des fachlichen Modells direkt folgen; bei einer relationalen Datenbank sind die Richtungen von Fremdschlüsseln bei einer „1:N“-Relation immer von „N“ nach „1“.

03 Installation

Neo4j ist vollständig in Java entwickelt und benötigt nur eine aktuelle Java-Runtime für die Ausführung. Es existieren fertige Pakete für

Unix/Linux, macOS, Windows, aber auch für fertige Docker-Container. Zusätzlich gibt es für Debian ein eigenes APT-Repository [2], sodass ein „apt-get install neo4j“ für die Installation ausreicht. So hat man zusätzlich den Vorteil, dass auch gleich Start/Stop-Skripte installiert und die Installation einfach aktuell gehalten werden kann. Gerade für Produktionsumgebungen ist dieses Verfahren von Vorteil. Nach der Installation kann man mit dem Internet-Browser auf „http://localhost: 7474“ gehen und den Neo4j-Browser sehen (siehe Abbildung 5).



➤ Abbildung 5: Der Neo4j-Browser

In diesem Browser lassen sich nun Abfragen mit Cypher machen und die Daten sowohl grafisch als auch tabellarisch darstellen. Der Browser ist im Übrigen keine Oberfläche für Endanwender – dafür ist die Abfragesprache Cypher für Endanwender in der Regel zu kompliziert –, aber er ist ein sehr mächtiges Werkzeug für Entwickler, etwa um Abfragen zu testen. Standardmäßig lauscht Neo4j auf den folgenden Ports:

7474: Zugriff per REST/HTTP. Hier steht ein REST-Interface zur Verfügung, um auf die Datenbank aus diversen Programmiersprachen zuzugreifen und eigene REST-Erweiterungen bereitzustellen (später mehr dazu). Zusätzlich steht hier der genannte Browser zur Verfügung.

1337: Der Port, auf den Kommandozeilen-Befehle zugreifen, etwa „neo4j-shell“ oder „neo4j-import“. Dieser Port ist standardmäßig deaktiviert und sollte nur für Zugriffe von der lokalen Maschine aktiviert sein.

7687: Zugriff per „BOLT“. Dies ist in der Version 3 hinzugekommen und die präferierte Methode für den Zugriff auf die Datenbank, da „BOLT“ im Gegensatz zu „REST“ binär ist und stehende Verbindungen hat – dadurch ist „BOLT“ schneller. „REST“ steht aber weiterhin zur Verfügung, da es quasi von jeder Technologie verwendbar ist.

04 Community und Enterprise

Neo4j existiert sowohl in einer lizenzfreien Community als auch in einer lizenzpflichtigen Enterprise-Version. Letztere bietet zusätzlich unter anderem die folgenden Eigenschaften:

- Clustering: Betrieb auf mehreren Knoten für erhöhte Ausfallsicherheit und Lastverteilung
- Hot Backups: Backup der Daten im Betrieb, ohne die Datenbank zu stoppen

Für den Betrieb einer professionellen Lösung sind die genannten Punkte essenziell, daher sollte man Budget für entsprechende Lizenzen

einplanen. Projekte lassen sich problemlos mit der Community-Version starten, ein Umstieg zu einem späteren Zeitpunkt ist erfahrungsgemäß unproblematisch. Beide Versionen sind immer auf dem gleichen Stand.

Die Neo4j Community Edition ist Open Source und die Quelltexte stehen auf GitHub [5] zur Verfügung. Jeder hat dadurch die Möglichkeit, die Quelltexte zu forken, die Funktionsweise zu verstehen, aber auch selbst Änderungen vorzunehmen und über einen Pull Request sogar die eigenen Änderungen in das Produkt zu übernehmen. So sieht ein vorbildliches kommerzielles Open-Source-Projekt aus.

05 Import von Daten

Zur Verdeutlichung dienen beispielhaft die folgenden beiden CSV-Dateien. „department.csv“ beschreibt Abteilungen in einer Hierarchie (siehe Listing 2), während „subsidiary.csv“ die Niederlassungen erläutert, die jeweils ein oder mehrere Departments haben können (siehe Listing 3).

```
id,name,member_of
management,Management
sales,Sales,management
marketing,Marketing,management
development,Development,management
operations,Operations,management
rnd,Research and Development,development
```

Listing 2

```
id,name,departments,headquarter
ny,New York,management:sales,true
sf,San Francisco,marketing:development,false
la,Los Angeles,operations:development:sales,false
ho,Honolulu,rnd,false
```

Listing 3

Die CSV-Dateien enthalten jeweils eine Kopfzeile mit den Spalten-Namen. Um diese zu importieren, können sie einfach in das „import“-Verzeichnis von Neo4j kopiert werden. Von dort aus werden sie mit einem Befehl geladen (siehe Listing 4).

```
load csv with headers from "file:/subsidiary.csv" as
line merge (s:Subsidiary {id:line.id}) set s.name
= line.name, s.departments = split(line.depart-
ments,","), s.headquarter = line.headquarter;
```

Listing 4

Die CSV-Datei wird geladen, die erste Zeile wird als Header verwendet und gibt die Namen vor. Jede Zeile wird in die Variable „line“ geladen; über einen „merge“ werden die Knoten angelegt oder über die „id“ aktualisiert. Mit dem Befehl „match (p:Department),(c:Department) where p.id = c.member_of create (c)-[:MEMBER_OF]->(p);“ werden die Departments von Kind zu Vater als „MEMBER_OF“ verknüpft. Den Headquarter können wir mit einem zusätzlichen Label auf dem betreffenden Knoten abbilden: „match (s:Subsidiary {headquarter:true}) set s:Headquarter;“. Die Properties „MEMBER_OF“, „Departments“ und „Headquarter“

benötigen wir nicht mehr. Da sie als explizite Kanten abgebildet sind, entfernen wir sie mit „match (d:Department) remove d.member_of;“ und „match (s:Subsidiary) remove s.headquarter, s.departments;“.

Die Niederlassungen importieren wir mit „load csv with headers from „file:/subsidiary.csv“ as line merge (s:Subsidiary {id:line.id}) set s.name = line.name, s.departments = split(line.departments,“,“). Die Spalte „Departments“ wird mit einem Split gleich in mehrere Werte zerteilt. Jetzt verknüpfen wir „Departments“ mit „Subsidiary“ als „LOCATED_IN“: „match (d:Department) match (s:Subsidiary) where d.id in s.departments create (d)-[:LOCATED_IN]->(s);“. Abbildung 6 zeigt das Ergebnis.

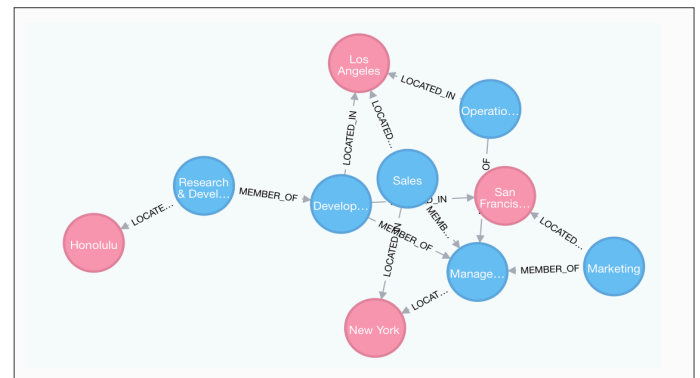


Abbildung 6: Der vollständige Beispiel-Graph

Testweise können wir jetzt die folgenden Abfragen verwenden: „match (d:Department)-[:LOCATED_IN]->(s:Subsidiary) return d.name,collect(s.name)“ zeigt die Liste aller Abteilungen und die Niederlassungen, an denen sie vorkommen. Eine Stärke von Cypher ist, dass die Pfade beliebig lang sein dürfen. Welche Abteilungen stehen direkt oder indirekt unter dem Headquarter? „match (hq:Headquarter)-[:LOCATED_IN]-(d:Department)-[:MEMBER_OF*0..2]-(d2:Department) return hq,d,d2“ liefert das Ergebnis.

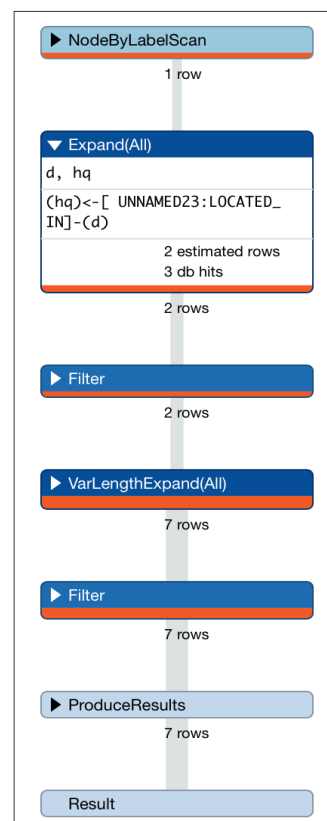


Abbildung 7: Ergebnis eines „profile“

Der Quantifier „*0..2“ erlaubt die Entfernung von null bis zwei Knoten; Neo4j ist also in der Lage, die Pfade selbst zu finden. Damit sind Traversierungen über hierarchische Datenstrukturen sehr einfach. Cypher-Statements lassen sich damit wiederverwenden, unabhängig davon, von welcher Ebene aus man startet. Wenn man dann komplexe Abfragen erstellt, die über große Datenmengen arbeiten, kann man sich durch ein Voranstellen von „profile“ und „explain“ entsprechende Details darstellen lassen (siehe Abbildung 7). Mit diesen Informationen kann man die Zugriffs-Logik und -Reihenfolge nachvollziehen, um Performance-Probleme zu lösen, etwa über Indizes.

06 Erweiterbarkeit

Es gibt mehrere Möglichkeiten, den Funktionsumfang von Neo4j zu erweitern – etwa Procedures, die sich mit „call“ aufrufen lassen. Der Befehl „call dbms.procedures()“ liefert eine Liste der aktuell verfügbaren Procedures. Einige sind bereits in der Grundinstallation enthalten. Beispielsweise gibt „call db.labels()“ eine Liste der Labels aller Nodes in der Datenbank aus, erwartungsgemäß *Subsidiary*, *Department* und *Headquarter*. Diese Procedures können selbst entwickelt werden. Zuvor sollte man sich *Awesome Procedures for Neo4j 3 (APOC)* [3] anschauen, wo bereits etliche gut wiederverwendbare Procedures stehen. Nach dem Download packt man das JAR-File in das Plug-in-Verzeichnis und startet Neo4j neu. Mit „apoc.index.create“ lässt sich ein Volltext-Index auf die Namen der Knoten erzeugen: „CALL apoc.index.addAllNodes(name,{ Department: [„name“], Subsidiary: [„name“] })“. Dabei werden auch die bereits vorhandenen Knoten indiziert.

Mit „call apoc.index.search(„name“,“*lo*“)“ findet man alle Knoten, deren Name „lo“ beinhaltet. Außerdem erhält man erwartungsgemäß das Subsidiary „Los Angeles“ und die beiden Departments, in denen *Development* vorkommt. Bei dieser Antwort sieht man übrigens eine Stärke von Cypher; Antworten können beliebige Knoten und Kanten gemischt enthalten.

APOC bietet sehr viele nützliche Funktionen an, so lassen sich mit „match (s:Subsidiary)--(d:Department)--(s2:Subsidiary) where id(s1) < id(s2) with d as d, s1 as s1, s2 as s2, ceil(distance(point({longitude: s1.longitude, latitude: s1.latitude}),point({longitude: s2.longitude, latitude: s2.latitude}))/ 1000) as kilometers return d.name,s1.name,s2.name,kilometers“ die Subsidiaries geokodieren. Dabei erhalten wir die Geo-Koordinaten als „longitude“ und „latitude“ und speichern diese als weitere Properties. Nun können wir diese Information nutzen, um die Entfernungen zwischen zwei Niederlassungen derselben Abteilung zu berechnen (siehe *Listing 5* und *Abbildung 8*).

```
match (s1:Subsidiary)--(d:Department)--(s2:Subsidiary)
where id(s1) < id(s2) with d as d, s1 as s1, s2 as s2,
ceil(distance(point({longitude: s1.longitude, latitude: s1.latitude}),point({longitude: s2.longitude, latitude: s2.latitude}))/ 1000) as kilometers return
d.name,s1.name,s2.name,kilometers
```

➤ Listing 5

Für diese Berechnung kommen die neuen Funktionen „point“ und „distance“ zum Einsatz, die ab Version 3.0 verfügbar sind. APOC bieten neben Volltextsuche und Geokodierung etliche weitere Prozeduren an, unter anderem das Laden von Daten aus einer JDBC-Datenquelle oder eine Integration mit anderen Datenbank-Technologien wie *ElasticSearch*, *MongoDB*, *Couchbase* oder *Cassandra*.

07 Integration in Anwendungen

Nachdem Neo4j installiert ist und die Daten zur Verfügung stehen, folgt die Integration in die Anwendungen. Prinzipiell kann jede Programmiersprache auf Neo4j zugreifen, die in der Lage ist, REST aufzurufen. Neo4j selbst bietet fertige Treiber für Java, JavaScript, .Net und Python an. Diese vereinfachen den Zugriff auf die Datenbank. Uns interessiert im Kontext dieser Zeitschrift der Treiber für Java. Alle Artefakte, die wir benötigen, stehen direkt im *Maven Central Repository* [6] zur Verfügung und können daher direkt in *Maven* oder *Gradle* verwendet werden.

Neo4j kann übrigens „standalone“ laufen oder „embedded“ innerhalb der Anwendung. Letztere ist keine eingeschränkte Version, sondern vollständig; sie eignet sich daher für Frameworks wie *Spring Boot* und natürlich auch für das Testen mit *JUnit*. Für produktive Anwendungen ergibt es allerdings mehr Sinn, Neo4j „standalone“ zu starten, idealerweise auf einer dedizierten Maschine.

Für eine optimale Integration in Java steht *Neo4j-OGM (Object Graph Mapper)* [7] zur Verfügung, um das Mapping zwischen Java-Objekten und Knoten/Kanten in der Datenbank zu vereinfachen. Ähnlich wie *JPA* werden dafür entsprechende Annotationen, und zwar „@NodeEntity“ für Knoten und „@RelationshipEntity“ für Kanten, verwendet. Zusätzlich existiert mit *Neo4j-Harness* eine Erweiterung für *JUnit*, die es sehr einfach ermöglicht, Neo4j-Instanzen „embedded“ über die in *JUnit* bereitgestellten „@Rule“-Annotationen zu integrieren und damit das Testen zu vereinfachen. Als Erstes benötigen wir ein paar *Maven-Dependencies*. Die Code-Beispiele sind unter [8] zu finden. Im Einzelnen sind das:

- **neo4j-ogm-core:** Der Hauptteil von Neo4j-OGM
- **neo4j-ogm-bolt-driver:** Zugriff auf die Datenbank mittels *BOLT*. Einige Abhängigkeiten auf „org.neo4j“ müssen ausgeschlossen werden, da sie aktuell noch auf eine veraltete Neo4j-Version zeigen.
- **neo4j-ogm-http-driver:** Wird nicht benutzt; Neo4j-OGM beschwert sich allerdings, wenn er fehlt.
- **neo4j-harness:** Das Neo4j-Harness-Test-Framework, das eine passende „@Rule“ liefert, bringt auch transitiv die zurzeit aktuelle Neo4j 3.0.6 selbst als Abhängigkeit mit.
- **JUnit:** Das JUnit-Test-Framework: Für die beiden Entitäten müssen wir *Department* und *Subsidiary* als DTOs anlegen und mit *@NodeEntity* annotieren. *Listing 6* zeigt beispielsweise „Department.java“.

```
@NodeEntity
public class Department {
    @GraphId
    private Long graphId;
    private String id;
    private String name;
    @Relationship(type="LOCATED_IN")
    private Set<Subsidiary> subsidiaries = new
HashSet<Subsidiary>();
    // getter und setter, hashCode und equals und
    evtl. toString
}
```

➤ Listing 6

Development	San Francisco	Los Angeles	560
Sales	New York	Los Angeles	3942

➤ Abbildung 8: Tabellarisches Ergebnis der Entfernung einzelner Abteilungen



Es handelt sich um eine ganz normale DTO-Klasse. „@GraphId“ definiert das Field, das die interne, von Neo4j automatisch zugewiesene ID enthält. Diese wird für die interne Verwaltung gebraucht. Die Verbindung zu Subsidiary vom Typ „LOCATED_IN“ wird mit einem „Set<Subsidiary>“ und der Annotation „@Relationship(type=“LOCATED_IN“)“ abgebildet.

Die Richtung kann mit „direction“ angegeben werden, wobei „OUTGOING“ der Standardwert und daher nicht angegeben ist. Ausgehende Kanten und die dazugehörigen Knoten werden von OGM automatisch geladen, sie sind daher implizit „EAGER“. Die Kante an sich erhält man hier nicht. Um die Eigenschaften der Kante zu erhalten, kann man eine weitere DTO hinzufügen und diese mit „@RelationshipEntity“ annotieren. Die Geschäftslogik wird entsprechend in eine Klasse „DepartmentService“ gepackt [8].

„DepartmentService“ erhält im Constructor eine Session. In Frameworks, die Dependency Injection unterstützen, wie Java EE 6 (mittels CDI) oder Spring, kann man sich dieses Objekt injizieren lassen. Session ähnelt dem JPA EntityManager, enthält die passenden Methoden zum Finden/Laden und Speichern von Objekten und arbeitet mit Entitäten, die, wie oben gezeigt, mit „@NodeEntity“ oder „@RelationshipEntity“ annotiert sind.

Die Methode „findByName“ zeigt, wie „session.query“ verwendet wird, um eine Cypher-Query ausführen zu lassen, dabei werden Variablen wie „{name}“ durch die Inhalte der übergebenen Map ersetzt. Das Ergebnis ist ein Department. Das Mapping zu diesem Objekt wird von OGM erledigt. Man sieht hier weder die internen Datenstrukturen noch die Kommunikation mittels REST oder BOLT. In [8] ist der Test zu sehen.

Die „@Rule“ Neo4jRule startet Neo4j als Embedded-Version, die Authentifizierung wird deaktiviert. Standardmäßig sind HTTP und BOLT aktiviert und BOLT lauscht in diesem Fall abweichend auf Port 5001.

Mit „copyFrom“ wird ein bereits vorhandenes Datenbank-Verzeichnis geladen. Dazu wurden die „standalone“-Version von Neo4j gestoppt und der Inhalt von „\$NEO4J_HOME/data/databases/graph.db“ in das Projekt nach „src/main/resources“ kopiert. Es ist sehr wichtig, dass Neo4j nicht läuft, während man die Daten wegekopiert. Neo4rule kopiert nun bei jedem Starten den Inhalt dieses Verzeichnisses in ein temporäres Verzeichnis und löscht dieses nach dem Beenden. Dadurch bleibt der Inhalt der Datenbank unverändert. Es handelt sich dabei um eine Best Practice, da es sehr einfach ist, ein bestimmtes komplexes Datenbank-Szenario als Grundlage für die Tests zu erzeugen.

In unserem Fall ist der Graph aus den Beispielen enthalten. Die Methode „createSession“ erzeugt eine Session für OGM, die „DepartmentService“ benötigt. „SessionFactory“ erhält in diesem Fall den Parameter „com.prodyna.neo4j.sample“, das Basis-Package, in dem unsere Entitäten zu finden sind. Neo4j-OGM scannt alle Klassen in diesem Package, um die Annotationen zu ermitteln. Es werden der BOLT-Driver verwendet und auf Port 5001 (den BOLT Port) zugegriffen. Zusätzlich wird mit „setEncryptionLevel(“NONE“)“ jede Art von Security und Zertifikatsprüfung ausgeschaltet.

Die Testmethode „readDepartmentWithSubsidiaries“ holt sich mit „createSession“ eine Session, erzeugt damit eine Instanz von „DepartmentService“ und kann sie dann verwenden. In diesem Fall wird geprüft, dass sechs Departments gefunden werden und das Department Sales sowohl in Los Angeles als auch in New York zu finden ist.

Es existiert noch Spring Data Neo4j (SDN), das auf Neo4j-OGM aufbaut und damit Neo4j in das Konzept der Repositories von Spring Data integriert, wo grundsätzlich keine Implementierungen notwendig sind, da alles deklarativ ist. Die Klasse „DepartmentService“ könnte man als „DepartmentRepository“ dann wie in Listing 7 implementieren.

```
public interface DepartmentRepository extends
GraphRepository<Department> {
    @Query("match (d:Department {name:{name}}) return
d")
    Department findDepartmentByName(@Param("name")
String name );
}
```

► Listing 7

SDN empfiehlt sich, wenn man sowieso auf der Spring-Plattform entwickelt. Dann kann man insbesondere mit Spring Boot und einer „embedded“ Neo4j schnell und einfach auszurollende und startende Anwendungen bauen, die sich auch sehr einfach in Docker-Container integrieren lassen. Dies ist ideal für Microservices.

Sowohl Neo4j-OGM als auch SDN erfordern ein Scannen der Klassen. In Umgebungen, in denen Java keinen direkten Zugriff auf die Klassen und JAR-Dateien hat, wie OSGi oder Java EE, hat sich das teilweise als sehr kompliziert ergeben. Während SDN damit noch relativ gut umgehen kann, ist die Implementierung von Neo4j-OGM etwas holprig und verursacht teilweise große Probleme mit der Auswirkung, dass es die Entitäten nicht findet. Um jedes Scannen zu vermeiden, könnte man die annotierten Klassen bei der Instanziierung übergeben.

08 Fazit

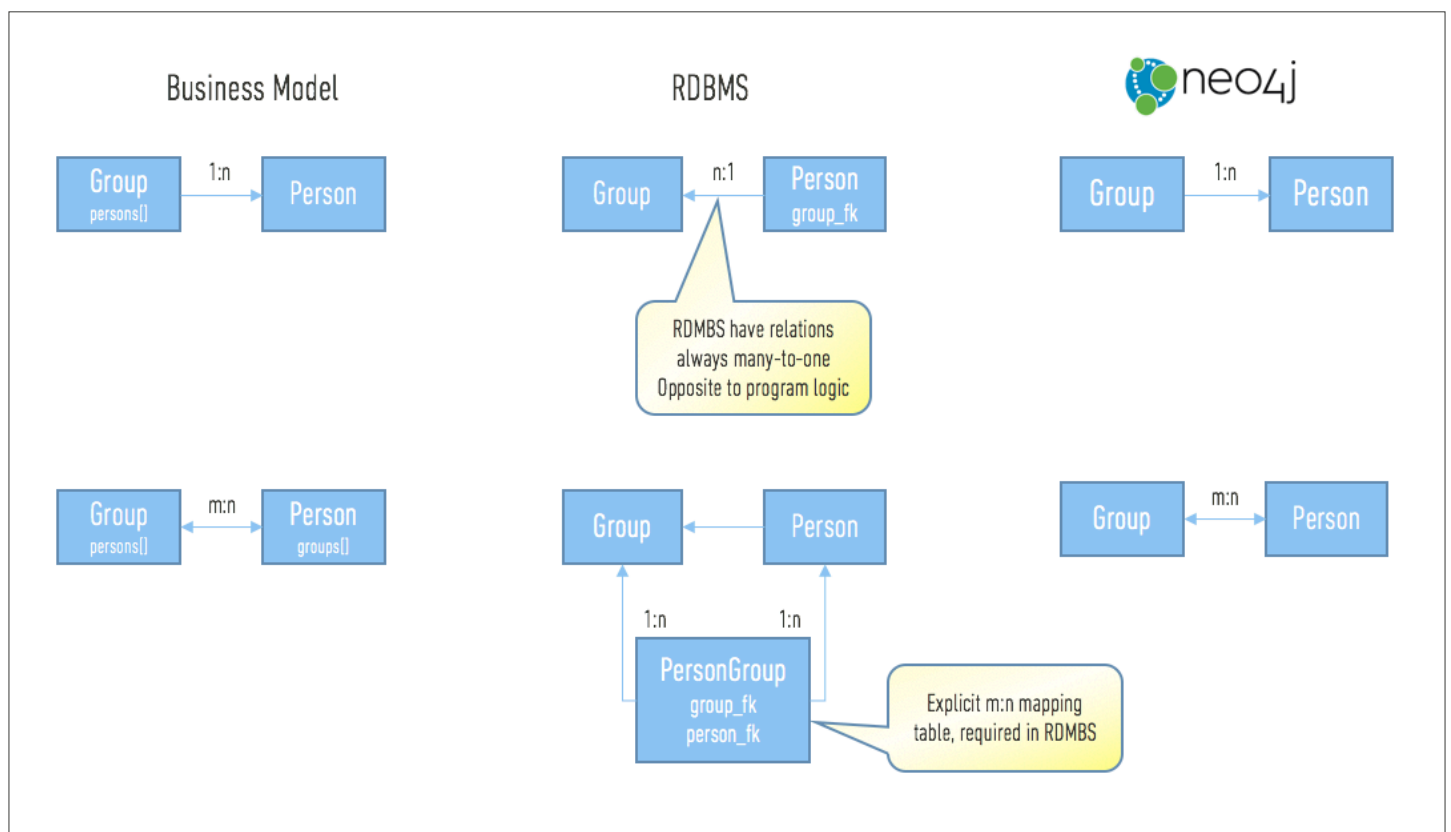
Neo4j ist eine hochinteressante Datenbank, die ihre Popularität mit Recht erhält. Die Tatsache, dass alle Artefakte bereits im Maven Central Repository und die Quelltexte im GitHub verfügbar sind, erleichtert die Entwicklung enorm. Auch die Verfügbarkeit einer Community-Edition sorgt dafür, dass ein Projekt schnell starten kann und eine kommerzielle

Lizenz sowie ein Umstieg auf die Enterprise Version später einfach möglich sind.

Die verfügbaren REST-Schnittstellen lassen eine Integration in beliebige Technologien und Programmiersprachen zu, wobei Treiber für gängige Plattformen, insbesondere Java, die Entwicklung von Anwendungen erleichtern. Einzig das unglückliche Scan-Verhalten von Neo4j-OGM (siehe Abbildung 9) sorgt für Probleme in diversen Umgebungen. Die wahre Stärke von Neo4j ist natürlich ihr Graphenmodell und die Tatsache, dass sie vollständig transaktional ist – sie folgt, im Gegensatz zu vielen NoSQL-Datenbanken, dem ACID-Konzept. Daher gibt es mit Transaktionalität keine Probleme, eine Anforderung, die im Geschäftsumfeld häufig benötigt wird. Relationale Modelle lassen sich problemlos auf das Graphenmodell abbilden, aber mit den genannten Eigenschaften, wie mehrere Labels pro Knoten, lassen sich wesentlich komplexere Modelle abbilden, die am Ende immer noch beherrschbar sind.

Auch viele unvorhersehbare Anforderungen lassen sich meist ohne viel Aufwand integrieren. Ein typisches Beispiel wäre: „Eine Person kann Mitglied mehrerer Gruppen sein“. Ein typisches Problem, bei dem eine „1:N“-Relation zu einer „M:N“-Relation umgebaut werden muss. Bei relationalen Datenbanken ist dafür eine technische Hilfstabelle notwendig, die eine „M:N“-Relation in zwei „1:N“-Relationen abbildet (siehe Abbildung 9).

Bei Neo4j gibt es dieses Problem nicht, es können beliebig viele Kanten zwischen zwei Knoten existieren, es gab also vorher eher eine künstliche Beschränkung, die aufgehoben werden kann. Bei relationalen Datenbanken gibt es zusätzlich die Eigenschaft, dass bei „1:N“-Relationen die Relation immer von der „N“-Seite auf die „1“-Seite zeigen



► Abbildung 9: Neo4j kann fachliche Beziehungen direkt als Relationship abbilden



muss (Fremdschlüssel). Bei einer Graph-Datenbank kann der Entwickler die Richtung selbst festlegen; es spielt für die Datenbank keine Rolle. Daher kann das Graphenmodell wesentlich näher an das fachlichen Modell angelehnt sein, ohne auf Hilfskonstruktionen zurückgreifen zu müssen.

Die Abfragesprache Cypher sieht auf den ersten Blick kompliziert aus, nach etwas Übung schreibt man jedoch komplexe Abfragen, ohne nachzudenken. Es existieren viele Funktionen, die auch sehr komplexe Abfragen erlauben. Zusätzlich kann der Umfang durch fremde oder eigene Procedures beliebig erweitert werden. Die Möglichkeit, selbst Pfade zu finden, erlaubt es beispielsweise, problemlos mit hierarchischen Strukturen oder mit Ontologien zu arbeiten. Bei relationalen Datenbanken sind teilweise viele SQL-Abfragen notwendig, die zusätzlich mit Java-Logik integriert sein müssen. Bei Cypher reicht meist ein einziges Statement aus.

Eine Eigenschaft relationaler Datenbanken ist das Problem, dass Antworten selbst immer eine Tabelle sein müssen; bei komplexen Abfragen ist also die Tabelle denormalisiert und enthält Redundanzen, die im Programm dann wieder verarbeitet sein müssen. Bei Cypher sind die Ergebnisse von Abfragen auf Wunsch auch tabellarisch, vorzugsweise sind es jedoch Aufzählungen von Knoten und Kanten, mit dem Verweis darauf, wie die Knoten untereinander verbunden sind.

Wie fast alle NoSQL-Datenbanken ist die Technologie proprietär. Selbst ein Umstieg auf eine andere Implementierung würde einen hohen Aufwand bedeuten, da etwa die Abfragesprache Cypher nicht genormt ist. Neo Technologies (der Hersteller von Neo4j) möchte dem entgegenwirken und hat die Sprache Cypher als OpenCypher [9] zur Verfügung gestellt mit dem Ziel, dass auch andere Hersteller von Graph-Datenbanken Cypher (wenn auch zusätzlich zur vorhandenen)

als Abfragesprache verwenden. Damit wäre eine Migration zwischen verschiedenen Herstellern wesentlich leichter und auch Frameworks wie SDN könnten besser wiederverwendet werden.

Neo4j ist clusterfähig, es laufen also mehrere Instanzen. Man muss sich bewusst sein, dass Neo4j kein Sharding unterstützt – jede Neo4j-Instanz hält eine Kopie aller Daten. Der Grund ist einfach: Bei Graphen gibt es keine gute Logik dafür, wie die Knoten auf verschiedene Instanzen aufgeteilt werden könnten. Die Clusterfähigkeit ist aber essentiell für eine hochverfügbare Umgebung und steht auch nur in der Enterprise-Version zur Verfügung.

Darko Križić ist Gründungsmitglied von PRODYNA und in der Funktion des Chief Technical Officers tätig. Damit bildet er die Brücke von der Software-Entwicklung über den hochverfügbaren Betrieb kritischer Anwendungen bis zur strategischen Management-Beratung. In seiner Freizeit entspannt er sich als Privatpilot bei einem Rundflug oder ist auf dem Mountainbike unterwegs.

Weitere Informationen

- [1] <http://nosql-database.org>
- [2] <http://debian.neo4j.org>
- [3] <https://github.com/neo4j-contrib/neo4j-apoc-procedures>
- [5] <https://github.com/neo4j/neo4j>
- [6] <http://search.maven.org>
- [7] <https://neo4j.com/docs/ogm-manual/current/>
- [8] <https://github.com/dkrižic/neo4j-sample>
- [9] <http://www.opencypher.org>

Kontakt

PRODYNA AG

Ludwig-Erhard-Str. 12-14 65760 Eschborn

T +49 69 597 724 - 0 F +49 69 597 724 - 700

info@prodyna.com prodyna.com



Ihr Ansprechpartner

Darko Krizic

Chief Technology Officer (CTO)

darko.krizic@prodyna.com

Autor: Darko Krizic Copyright by PRODYNA AG

Unternehmensprofil

PRODYNA ist ein innovatives IT-Beratungsunternehmen, spezialisiert auf das Thema Digital Business. Wir beraten Firmen zum Thema Digital Business Transformation und entwickeln die Custom Software Applikationen und Systeme, die Ihr Unternehmen benötigt, um Ihre Wettbewerbsfähigkeit zu gewährleisten.

Gegründet im Jahr 2000, mit Hauptsitz in Frankfurt am Main und 6 weiteren Standorten in Deutschland sowie Gesellschaften in der Schweiz, Österreich und Serbien ist PRODYNA ein privat gehaltenes und international aktives Unternehmen mit aktuell 280 Mitarbeitern.

Der Name PRODYNA steht für PROfessionalität und DYNAmik. Obwohl wir alle meistens in der Einheit „Projekt“ denken, ist der Weg zum langfristigen Ziel viel länger. Somit ist es PRODYNA wichtig, Kontinuität für unsere Kunden zu garantieren. Unser Kapital ist das Wissen und die Erfahrung unserer Berater. PRODYNA investiert überdurchschnittlich viel in Weiterbildung und Mitarbeiterbindung und schützt somit ihr wertvollstes Gut. Um dies noch zusätzlich zu fördern, arbeitet PRODYNA – im Unterschied zu den meisten IT-Dienstleistern – grundsätzlich nur mit festangestellten Mitarbeitern.

Das Zeitalter des Kunden verlangt Agilität und kurze Projektlaufzeiten. Unsere Größe erlaubt es uns schnell und persönlich nach Ihren Bedürfnissen zu agieren. Wir haben keine langen internen Abstimmungsprozesse, denn wir kennen uns gegenseitig und wissen, wer über die benötigten Skills für Ihr Projekt verfügt.

Gezielte Kundenorientierung auf Seiten von PRODYNA und eine außergewöhnlich hohe Kundenzufriedenheit sind die Voraussetzungen für langfristige Kundenbeziehungen. Mit hoher Transparenz und klarer Kommunikation arbeitet PRODYNA intensiv daran, das Vertrauensverhältnis zu unseren Kunden zu stärken. Mit dieser Philosophie in Kombination mit kontinuierlichen Verbesserungsprozessen ist PRODYNA in den letzten Jahren außerordentlich erfolgreich gewesen.



Visit prodyna.com