

Business Strategy

Innovation
Branding
Solution
Marketing
Analysis
Ideas
Success
Management

Veröffentlichung
im JavaSPEKTRUM
Ausgabe - 06/2016

Graphdatenbanken in der Praxis –
Anwendungsbereiche für Neo4j



► Neue Technologien müssen sich stets an dem messen, was bereits vorhanden ist. Was können also Graphdatenbanken gegenüber relationalen Datenbanken leisten? Neo4j ist der aktuell bekannteste Vertreter unter den Graphdatenbanken und seit dem erfolgreichen Einsatz bei den Enthüllungen der Panama Papers in aller Munde. Doch wie steht es um den Einsatz außerhalb der Musterbeispiele? Immerhin sind bei der Entwicklung einer Business-Anwendung Probleme ganz unterschiedlicher Art zu lösen und nicht alle fallen in den Bereich der Graphentheorie. Ergibt sich daraus, dass man Neo4j die Eignung für Business-Anwendungen aberkennen muss? Die kurze Antwort ist: Nein. Da der Einsatz einer jungen Technologie wie Neo4j jedoch nicht nur Vorteile mit sich bringt, soll dieser Artikel einen Überblick verschaffen, welche Vor- und Nachteile bei einem Einsatz von Neo4j entstehen.

Neo4j

Neo4j ist eine native Graphdatenbank. Dadurch, dass die Daten auch intern als Graph persistiert werden, können einige Abfragen deutlich schneller beantwortet werden als mit einer relationalen Datenbank. In Neo4j gibt es Knoten und Kanten, die immer gerichtet sind. Diese bekommen beim Erstellen eine eindeutige ID und können mit Feldern versehen werden. Typen werden wiederum mit Labels abgebildet. Als Query Language wird Cypher verwendet. Cypher ist eine mächtige Sprache, mit der sich auch komplizierte Abfragen leserlich schreiben lassen. Zudem erfüllt Neo4j ACID (Atomicity, Consistency, Isolation, Durability) und ist somit hervorragend für den Einsatz als Persistenzschicht in Business-Anwendungen geeignet. Das Neo4j-Motto „Graphs are everywhere“ erscheint hierbei sehr treffend, denn auch die Java-Objekte bilden einen Graphen und somit scheint es logisch auch die Persistenzschicht in einem Graph abzubilden. Jedoch ist das Mapping nicht so direkt, wie man auf den ersten Blick meinen könnte.

Daten-Layer

In Bezug auf Daten-Layer kommt für gewöhnlich nicht nur die Frage auf, ob etwas möglich ist, sondern auch wie aufwändig die Umsetzung wird.

Mit dem Einsatz von Neo4j und Cypher werden Abfragen gerade auf stark vernetzte Daten deutlich einfacher als mit relationalen Datenbanken und ihrer Abfrage auf SQL. Die Lernkurve bei Cypher ist am Anfang recht steil und es kommt schnell Freude auf Abfragen zu erstellen.

Da Neo4j kein Schema besitzt und Kanten die Verbindung zwischen den Knoten darstellen, muss kein konventionelles 1:1-, 1:N- oder N:M-Mapping vorliegen. Es können auch später noch problemlos weitere Kanten hinzugefügt werden. Diese Dynamik hilft insbesondere bei der agilen Entwicklung, in der sich Anforderungen an die Datenbanken schneller ändern. Um Daten abzufragen, beschreibt man mit Cypher einen Graphen. Als Ergebnis bekommt man dann alle Daten, die auf diesen Graphen matchen.

Hierfür ist es nicht zwingend nötig den gesamten Graphen mit allen Relationen anzugeben. Abfragen auf Hierarchien werden so zum Beispiel stark vereinfacht, die Abfragen werden lesbarer und toleranter gegenüber späteren Änderungen. So kann man auch zu einem späteren Zeitpunkt noch weitere Ebenen in der Hierarchie einfügen, ohne dass Abfragen angepasst werden müssen.

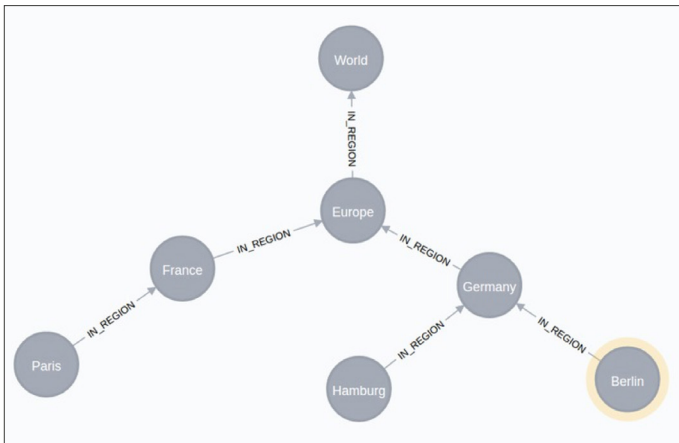


Abbildung 1: Beispiel einer Hierarchie

Bei der Abfrage nach allen Kind Elementen spielt die Ebene in der Hierarchie keine Rolle:

```
match(r:Region)-[:IN_REGION*1..]->(:Region {id:"Germany"}) return r
```

Nun ist aber nicht jedes Problem auf den ersten Blick auf einen Graphen zu übertragen. Zwar gibt es die Möglichkeit Where-Clauses zum Filtern zu verwenden, jedoch werden somit die Stärken einer Graphdatenbank nicht voll genutzt und es kommt zu Performance-Einbußen, weshalb man idealerweise weitestgehend auf Where-Clauses verzichten und die Überführung in einen Graphen wagen sollte. Ein gutes Beispiel für eine solche Übertragung ist das Erstellen eines Timetrees für das Abbilden von Daten (s. Abb. 2).

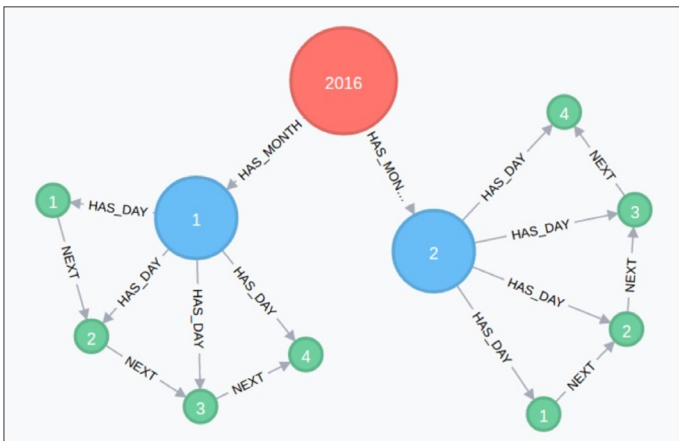


Abbildung 2: Beispiel eines Timetrees

Integration von Neo4j

Spring Data Neo4j (SDN) ist die Spring-Integration für Neo4j. Die Hauptaufgaben dieser Spring-Integration liegen im Transaktionsmanagement und im Object-Graph-Mapping (OGM). Des Weiteren bietet SDN über Neo4jTemplate eine API für den Zugriff auf den Graphen.

Auf den ersten Blick sieht alles recht bekannt aus: eine API für die Abfragen, Annotationen über den Entities und schon kann es losgehen. Dabei gilt es einige Hürden zu überwinden: Bei Spring Data Neo4j wird mit einem Integer-Parameter gearbeitet, der beim Laden und Speichern

gesetzt wird. Der Integer-Parameter gibt an, wie weit der Graph geladen oder persistiert wird, woraus sich zahlreiche Probleme ergeben, die man in der „relationalen Welt“ lange überwunden hat. Unter anderem kann nicht bestimmt werden, was gelesen und geschrieben wird, da der Integer-Parameter nicht beim Mapping angegeben wird.

Beim Persistieren zeigen sich die Probleme durch das Fehlen von Cascade-Typen. Dies macht eine objektspezifische Beschreibung der zu speichernden Daten unmöglich. Mit dem Abbilden der Relationen in Service-Methoden lässt sich das Persistieren über native Cypher-Statements realisieren. Hierbei bietet Cypher mit dem Befehl „MERGE“ in Verbindung mit „ON CREATE“ und „ON MATCH“ ein mächtiges Werkzeug zum Manipulieren des Graphen. „MERGE“ erstellt nur einen neuen Knoten oder eine Relation, wenn diese noch nicht existiert. „ON_CREATE“ und „ON MATCH“ dienen somit der späteren Unterscheidung der Fälle.

Beim Lesen ist es nicht möglich, den Objektgraphen in nur eine Richtung tiefer zu laden. Dies kann insbesondere bei zentralen Objekten, wie einer Länderzuweisung oder Mandanten, zum Laden tausender Objekte und zu einer schlechten Performance führen. Ein späteres Laden über Lazy Loading ist nicht möglich. Daten, die nicht direkt geladen werden, erhalten den Wert NULL. So kommt es beim Zugriff nicht zu einer Exception, was eine Unterscheidung zwischen nicht existenten und nicht geladenen Objekten unmöglich macht.

Die Annotation QueryResult von SDN ist die Lösung: mit einem QueryResult wird das Ergebnis von einem Cypher-Statement in ein Java-Objekt übertragen. Somit lässt sich genau bestimmen, was geladen werden soll. In der neuen Version können QueryResults nicht nur primitive Datentypen, sondern auch Entities enthalten. Beim Implementieren von Business-Anwendungen lassen sich somit oft komplexe Fälle komplett mit einem Cypher-Statement in Neo4j berechnen und über ein Query-Result wieder zurückgeben.

Fazit

Neo4j bietet zusammen mit Cypher ein mächtiges Werkzeug, mit dem sich extrem performante Datenbanksysteme bauen lassen. Neo4j hat den Sprung geschafft und ist durchaus in der Lage eine relationale Datenbank auch in großen Business-Anwendungen zu ersetzen. Mit Cypher bietet Neo4j eine extrem gut lesbare Form der Abfrage, mit der sich sehr komplizierte wie auch ganz normale Anwendungsfälle gut und schnell abbilden lassen. Aber es ist nicht alles Gold, was glänzt. Noch muss auf viel Komfort im Mapping verzichtet werden. Auch ist eine Datenbank ohne Schema nicht für alles geeignet. Für die Zukunft ist eine weitere Entwicklung in Richtung javax.persistence empfehlenswert.

Kay Landeck ist seit August 2015 bei PRODYNA als IT-Consultant und Software-Engineer tätig. Zu seinen Interessen gehören Algorithmen und Datenstrukturen. Seine Freizeit verbringt Kay Landeck mit Kampfsport.