Es rockt total

Neo4j – Eine graph-basierte transaktionale Datenbank

Phillip Ghadir

Neo4j ist eine quelloffene NoSQL-Datenbank, die Daten in Graphen organisiert. Sie ermöglicht die effiziente Verarbeitung unterschiedlichster Anwendungsfälle, die zum Beispiel in relationalen oder NoSQL-Datenbanken mit anderen Datenmodellen eher aufwendig sind.

Während die Kolumne von Michael Hunger in dieser Ausgabe den Aufbau von Neo4j und deren Performance genauer betrachtet, wollen wir uns diese Graphdatenbank aus der Perspektive des Anwendungsentwicklers, des „Praktikers", anschauen. Um es mit den Worten des Amazon-CTOs Werner Vogels zu sagen: „Für alles, was mehrere Beziehungen und Verbindungen hat, rockt Neo4j total." [Vog10]

Was die Graph-Basierung bringt

Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten – auch Beziehungen genannt. Kanten verbinden jeweils genau zwei Knoten miteinander. In einer graphenorientierten Datenbank werden Daten in Graphen gespeichert. Knoten und Kanten sind dort also Primitive erster Ordnung.

Neo4j ist eine graphen-orientierte Datenbank und bietet ausgefeilte Strategien für die verteilte Verwaltung von sehr großen, beliebig verknüpften Datenmengen. Dazu bietet die NoSQL-Datenbank Mechanismen an, mit deren Hilfe Indizes auf Knoten und Beziehungen definiert werden können. Auf einer Standard-Hardware lassen sich so mehr als 1 Mio. Beziehungen pro Sekunde traversieren. Dazu enthält Neo4j gleich mehrere Mechanismen, um komplexe Abfragen über einen Graphen zu formulieren.

In der Mathematik und Informatik sind viele allgemeine Graph-Algorithmen bekannt. Neo4j bringt Implementierungen einiger gängiger Graph-Algorithmen direkt mit. Beispielsweise sind für die Suche der kürzesten Wege sowohl der A*- als auch Dijkstras Algorithmus implementiert.

Neo4j eignet sich für wenig strukturierte Daten, die stark vernetzt sind. Wer große Mengen stark zusammenhängender strukturierter Daten verwalten muss, ist vermutlich mit einer relationalen Datenbank besser bedient. Aber für vernetzte Daten, bei denen stets die Beziehungen zwischen den Daten abgefragt werden, ist ein Graph eine natürliche Repräsentation und Neo4j eine angemessene Wahl. Abbildung 1 stellt die Elemente von Neo4j vor, über die wir als Anwender einen Überblick benötigen.

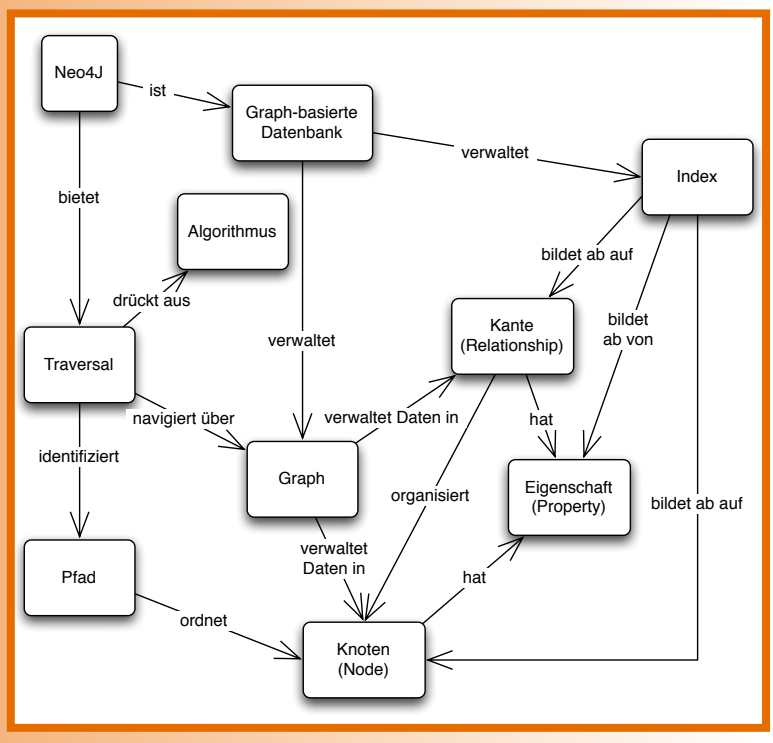

Abb. 1: Neo4j-Überblick

Erste Schritte mit Neo4j

Neo4j lässt sich als Server oder eingebettet in einen Java-Prozess betreiben. Im Folgenden betten wir Neo4j in unsere Software ein.

Zum Initialisieren von Neo4j erzeugt man im Programm einen Datenbank-Service:

```
private static final String DB_PATH = "data/db.neo4j";
private static GraphDatabaseService graphDb =
  new EmbeddedGraphDatabase( DB_PATH );
```

Mit Hilfe von Indizes kann man direkte Zugriffe auf Knoten und Beziehungen realisieren. Benötigte Indizes kann man für Knoten des Graphen mit der Methode `forNodes(String indexName)` anlegen:

```
private static Index<Node>kundenIndex =
  graphDb.index().forNodes( "kunden" );
private static Index<Node>partnerIndex =
  graphDb.index().forNodes( "partner" );
```

Die Knoten und Beziehungen eines in Neo4j gespeicherten Graphen können beliebig viele Attribute enthalten. Während wir per Java-API die Werte einzeln per `get-/setProperty` lesen und setzen können, steht uns über die REST-API der Server-Variante oder die Shell typischerweise JSON zur Verfügung.

Programmatisches Graph-Traversieren

Nehmen wir an, unsere Datenbank enthält bereits einen Graphen mit Knoten und Kanten (s. Abb. 2). Dann können wir über sogenannte Traversals definieren, auf welche Weise Neo4j über den Graphen traversieren soll. Mit Hilfe eigener Callback-Funktionen können wir spezifizieren, welche Teile des Graphen beim Traversieren zurückgeliefert werden sollen: Knoten, Beziehungen oder Pfade. Unter Pfaden versteht man bei Neo4j eine Menge von Knoten und Beziehungen, die ausgehend von einem Startknoten in definierter Reihenfolge ermittelt wurden.

Ein Traversal liefert stets einen Traverser, der `Iterable` implementiert. Damit lässt sich der Graph ein-

fach mit **foreach** nach eigenem Gutdünken traversieren. Abbildung 3 stellt die Möglichkeiten der Traversals dar.

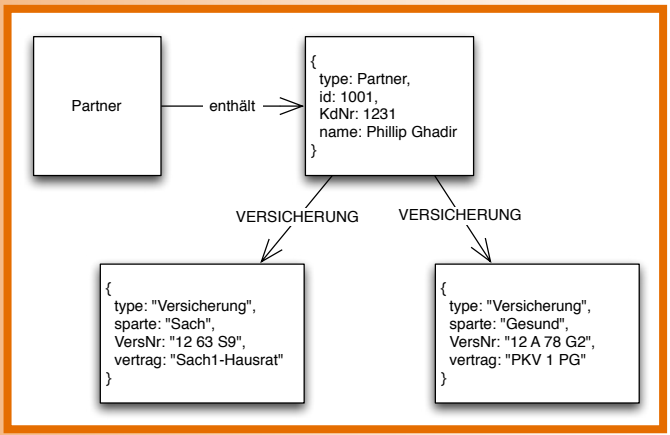

Abb. 2: Beispiel-Graph Partner mit Versicherungen

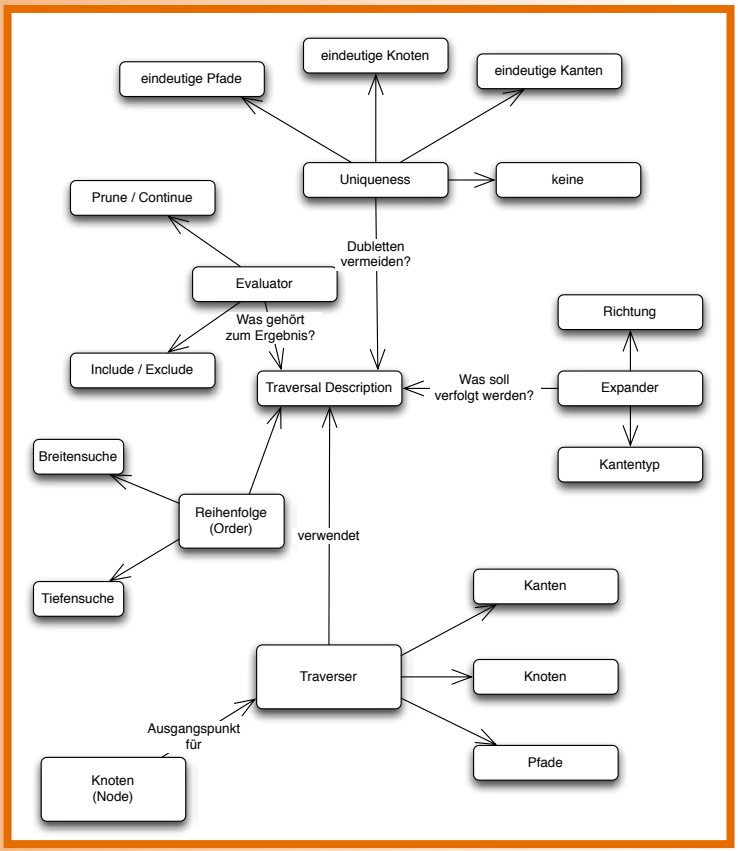

Abb. 3: Struktur des Traversers und der Beschreibung eines Traversals

Beispielsweise könnten wir in unserer Anwendung zu einem Partner die Sachversicherungen laden (s. Listing 1).

```
Node kundenKnoten = //...

Traverser vertragsTraversierer =
  kundenKnoten.traverse( Order.BREADTH_FIRST,
    StopEvaluator.END_OF_GRAPH,
    ReturnableEvaluator.ALL_BUT_START_NODE, RelTypes.VERSICHERUNG,
     Direction.OUTGOING );

Collection<Node> result = new ArrayList<Node>();
for (Node vertrag : vertragsTraversierer ) {
  if ( 'sach'.equals( vertrag.getProperty( 'sparte' ) ) ) {
    result.add( vertrag );
  }
}

//...
```

Listing 1: Traversiere über die Sachversicherungsverträge des Partners

Deklarative Abfragen

Neo4j bietet weitere Mechanismen, um über die Datenbank zu traversieren und Abfragen zu realisieren. Einer dieser Mechanismen ist Cypher, eine Sprache zur Formulierung von komplexen Abfragen.

```
START kunde=node:KUNDEN(id='1001')
MATCH kunde-[:VERSICHERUNG]-versicherung
WHERE versicherung.sparte = 'Sach'
RETURN versicherung
```

Listing 2: Cypher-Abfrage zum Auffinden der Sachversicherungen zum Partner mit der ID 1001

Listing 2 zeigt eine Cypher-Abfrage, mit der aus dem Graphen in Abb. 2 die Sachversicherungsverträge des Kunden mit der ID 1001 ermittelt werden können:

- Ausgehend von einer Menge von Startknoten (Zeile 1)
- werden Elemente des Graphen – Knoten und Kanten – (Zeile 2) selektiert,
- die gewissen Bedingungen genügen (Zeile 3) und
- auf die gewünschte Struktur der Antwort projiziert (Zeile 4).

Cypher-Abfragen kann man im Java-Code als String-Parameter an die Neo4j-Engine übergeben. Angenommen die Abfrage aus Listing 2 wird als String-Literal in der Variablen **abfrageDerSachversicherungen** gespeichert, dann können wir mit folgenden Zeilen die Abfrage an Neo4j zur Ausführung geben:

```
ExecutionEngine engine = new ExecutionEngine( graphDb );
ExecutionResult result =
  engine.execute( abfrageDerSachversicherungen );
```

Natürlich könnten wir auf gleiche Weise auch parameterisierte Abfragen formulieren und die Parameter als Map beim Aufruf von **ExecutionEngine.execute()** als zweites Argument übergeben.

Schreiben von Erfassungs-Clients

Möchten wir nun einen Graphen „zu Fuß“ anlegen, müssen wir als Erstes eine Transaktion starten, in der wir die Änderungen am Graphen vornehmen können.

```
Transaction tx = graphDb.beginTx();
```

Mit folgendem Code-Fragment können wir dann einen Knoten im Graphen anlegen:

```
Node n = graphDb.createNode();
n.setProperty( "key", "value" );
```

Wenn wir das Setzen der Properties in handgeschriebenen Domänenklassen kapseln wollen, könnten wir also schreiben:

```
Kunde k = new Kunde( graphDb.createNode(), "Phillip Ghadir" );
```

Des Weiteren sollten wir unsere Knoten natürlich auch mit definierten Beziehungen verbinden. Dies geschieht analog zum Anlegen von Knoten. Um die Beziehungen ordentlich pflegen zu können, erwartet Neo4j, dass wir die Beziehungstypen am Besten in einer Menge von Aufzählungen definieren, die das Marker-Interface `RelationshipType` implementieren. Im Falle unserer Versicherung könnte das zum Beispiel Beziehungen derart definieren:

```
enum RelTypes implements RelationshipType {
  VERSICHERUNG, ZAHLUNGSPLAN, DETAILS //, ...
}
```

Auf dem Startknoten der Beziehung ruft man `createRelationshipTo( ZIELKNOTEN, BEZIEHUNGSTYP)` auf:

```
Node kunde = //...
  Node vertrag = // ...
  kunde.createRelationshipTo( vertrag, RelTypes.VERSICHERUNG );
```

Abschließend markieren wir die Transaktion abhängig vom Erfolgsfall mit `tx.success()` oder `tx.failure()` und beenden sie in jedem Fall – also im `finally`-Block – mit `tx.finish()`.

Beispiel Partner

Um darzustellen, wie wir fachliche Zusammenhänge in einem Graphen organisieren könnten, soll die Entität `Partner` im Kontext einer trivialen, fiktiven Versicherung als Beispiel dienen. Sie kennen vermutlich das Problem: Der gemeinsame Nenner der Partner-Daten, die für die Verwaltung der Geschäftsbeziehung benötigt werden, ist im Vergleich zu der Vielzahl an Daten aus den Fachbereichen klein.

Beispielsweise möchte man aus der Perspektive des Inkasso-Systems Kontodaten und Mahnstand am Partner verwalten, während für andere Perspektiven andere Informationen, wie z. B. die Vertragsdetails oder kampagnenrelevante Partnerdaten, wichtig sind.

Domänenklassen mit Neo4j

Wir könnten zum Beispiel – wie in Abb. 4 dargestellt – schematisch festlegen, wie wir unsere Informationen in einem Graphen verwalten möchten.

Die skizzierten Domänenklassen können wir mit dem bisherigen Wissen recht einfach entwickeln. Beim Erzeugen eines Domänenobjekts verlangen wir einfach einen Knoten des Graphen, in dem wir die Daten des Domänenobjekts ablegen, z. B. wie in Listing 3.

```
public class Kunde {
  private final Node node;

  public Kunde(Node n) {
    this.node = n;
  }
  public Kunde(Node n, String name) {
    this( n );
    setName( name );
  }

  // Attribut-Zugriffsmethoden
  public void setName(String name) {
    node.setProperty( "name", name );
  }
  public String getName() {
    return node.getProperty( "name" );
  }

  // Sonstige Funktionen
  // ...
}
```

Listing 3: Domänenklasse Kunde – erfordert Neo4j-Knoten als Argument bei der Konstruktion und speichert direkt die Attribute im Knoten des Graphen

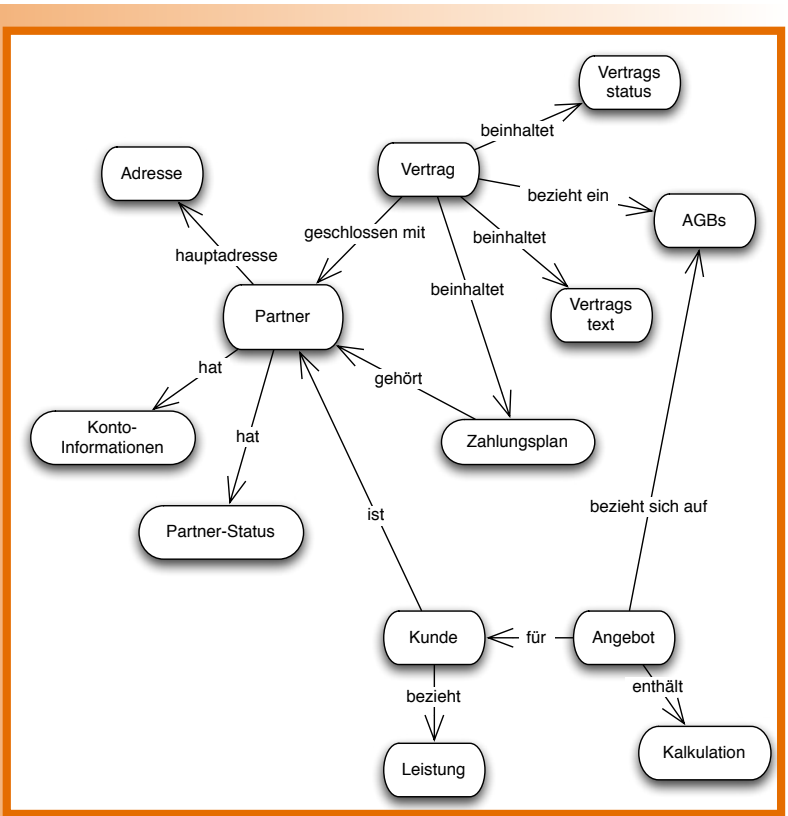

Abb. 4: Darstellung eines Schemas um fachliche Daten in einem Graphen zu organisieren.

Spring Data Neo4j API

Wer sich den manuellen Aufwand sparen möchte, die Attribut-Namen im Knoten zu definieren und (im Kopf) zu verwalten, kann mit dem Spring Data Neo4j API (oder auch Neo4j SDG API, Spring Data Graph) auf einfache Weise Modellklassen programmieren, die mittels Annotationen auf Graph-Strukturen abgebildet werden. Das vorige Beispiel sähe mit den Spring-Annotationen wie in Listing 4 aus.

```
public class Kunde {
  @GraphId
  private Long id;

  @Indexed
  private String name;

  public Kunde(Name name) {
    this.name = name;
  }

  // Attribut-Zugriffsmethoden
  public String getName() { return name; }
  public void setName( String name ) { this.name = name; }

  // Sonstige Funktionen
  // ...
}
```

Listing 4: Domänenklasse mit dem Neo4j SDG API - hier sogar mit Verwaltung des Index

Dank der Annotationen können wir auf das explizite Verwalten der Indizes im Graphen verzichten. Ähnlich werden einfache Abfragen für uns aufgelöst.

Auswirkungen auf das Programmiermodell

Eigenschaften von Knoten und Kanten können nur „einfache" Attribute enthalten: Integer, Double, String, Boolean usw. Das Abbilden von zusammenhängenden Informationen auf Knoten, Kanten und Attribute eines Graphen ist sehr geradlinig. Einfache Zusammenhänge können unmittelbar in der Graph-Struktur ausgedrückt werden. Zusätzliche Einteilungen wie zum Beispiel die Zuordnung zu einer Tabelle, einem Dokument oder einer Modellklasse entfallen.

Bei einer graphen-orientierten Datenbank definieren wir die Knoten und Verbindungen einfach so, wie wir sie auf eine Serviette kritzeln würden. Bei der Definition von Knoten- und Beziehungstypen ist man gut beraten, Namensräume zu verwenden. Praktisch ist auch das Einfügen von Knoten, die Meta-Informationen enthalten. Über Beziehung zwischen diesen Knoten und Entitäten können größere Kontexte gebildet werden. Man kann so im Graphen Kategorien-Bäume oder Tag-Netze einfügen, die das traversieren erleichtern können.

Verwendet man das Spring Data Neo4j API, bekommt man die Abstraktion von Knoten und Beziehungen in Modellklassen. Man definiert in einer Modellklasse Attribute inklusive Getter und Setter, wie in Listing 4 dargestellt. Den Zugriff auf in Beziehung stehende Knoten kann man dann ähnlich als annotiertes Attribut inkl. Zugriffsmethoden anbieten wie die Attribute, die Properties des Knotens kapseln.

Das Erstellen von Domänenklassen mit Neo4j ist – ob mit oder ohne die Spring-Erweiterung – nicht viel anders als die Abbildung von OO-Konzepten auf ein relationales Datenbanksystem mit O-/R-Mappern. Allerdings unterscheiden sich Cypher-Abfragen deutlich von SQL-Abfragen. Während man z. B. in SQL-Abfragen Daten über Primärschlüssel und Fremdschlüssel-Spalten verknüpfen muss, um zusammengehörende Daten zusammenzubekommen, beschreibt man in einer Cypher-Abfrage, welche Muster im Graphen prinzipiell gesucht werden.

Quellen auf Github

Sie finden die kommentierten Quelltexte zu diesem Artikel auf GitHub. Unter `http://github.com/pgh` gibt es das Git-Repository `practitioner-at-javaspektrum`, das die Quelltexte zu diesem Artikel enthält. Wer kein Git verwenden möchte, kann dort den Quelltext über die Weboberfläche ansehen. Sollten sich in den Code Fehler oder aber klare Don'ts eingeschlichen haben, bitte ich um eine kurze E-Mail oder besser noch um entsprechende Pull-Requests oder um das Erstellen eines Tickets. (Ein entsprechendes Issue-Tracking ist bei GitHub direkt integriert und lässt sich ebenfalls über die Weboberfläche verwenden.)

Das Projekt erfordert keine spezielle Entwicklungsumgebung. Die Software sollte leicht abgerufen und lokal in einer Java-IDE Ihrer Wahl mit einem aktuellen Neo4j kompiliert werden können.

Man sollte schematisch eine konkrete Vorstellung haben, wo im Graphen welche Informationen abgelegt werden. Insbesondere für komplexe, föderierte Graphen muss man sich selbst um Namensräume und Glossare für Knoten- und Kantentypen kümmern.

Fazit

Neo4j ist eine großartige Ergänzung zur relationalen Datenhaltung. Wenn wir Massen von zusammenhängenden Daten verwalten müssen, sind relationale Datenbanken immer noch unschlagbar. Wenn es allerdings um die Verwaltung von komplexen Beziehungen geht, bieten graphen-orientierte Datenbanken die Möglichkeit, diese komplexen Zusammenhänge unmittelbar abzubilden und effektiv zu verwalten.

Die Verwendung der generischen API ermöglicht auf einfache Weise die Entwicklung beliebig komplexer generischer Frameworks und Applikationen. Mit dem Spring Data Neo4j API existiert darüber hinaus die Option, Domänenklassen zu implementieren, die die Vorzüge der Graph-Orientierung und der Kapselung fachlicher Zusammenhänge verbinden.

Insgesamt lohnt sich die Evaluation von Neo4j.

Links

[a-star] A*-Algorithmus,
`http://de.wikipedia.org/wiki/A*-Algorithmus`
[dijkstra] Dijkstra-Algorithmus,
`http://de.wikipedia.org/wiki/Dijkstra-Algorithmus`
[Hung12] M. Hunger, Good Relationships,
The Spring Data Neo4J Guide Book, InfoQ, Free eBook, 2012,
`http://www.infoq.com/minibooks/good-relationships-spring-data`
[neo4j] Neo4j-Homepage, `http://neo4j.org`
[neo4j-js-sample] Quelltexte auf Github,
`http://github.com/pgh/practitioner-at-javaspektrum`
[neo4j-overview] `http://docs.neo4j.org/chunked/stable/images/graphdb-overview.svg`
[neo4j-prs] `http://sliwww.slideshare.net/jexp/neo4j-graph-database-presentation-german`
[neo4j-trav] `http://docs.neo4j.org/chunked/stable/images/graphdb-traversal-description.svg`
[Vog10] Neo4j Blog: CTO of Amazon:
"Neo4j absolutely ROCKS!", 23.3.2010,
`http://blog.neo4j.org/2010/03/cto-of-amazon-neo4j-absolutely-rocks.html`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com