Die Java-basierte Graphdatenbank Neo4j

Olaf Herden
Duale Hochschule Baden-Württemberg Campus Horb

Schlüsselworte

Graphdatenbank, Neo4j, NoSQL, Big Data.

Abstract

In der jüngsten Vergangenheit hat es im Datenbankmarkt einen Paradigmenwechsel gegeben. Haben über viele Jahre relationale Datenbanken den Markt klar beherrscht und jede Art von Daten abgespeichert („One Size Fits All"), so sind unter dem Begriff NoSQL (Not only SQL) eine ganze Reihe von neuartigen Systemen entstanden. Eine besondere Kategorie von NoSQL-Datenbanken sind Graphdatenbanken. Im Java-Umfeld ist Neo4j der bekannteste Vertreter dieser Gruppe.

1 Einleitung[1]

Seit Beginn der professionellen Datenverarbeitung finden intensive Untersuchungen statt, wie Daten am praktikabelsten persistent gehalten werden können, d.h. dauerhaft über die Laufzeit des aktuellen Programms hinaus abgespeichert werden können. Nach ersten Ansätzen, die Daten in Dateien zu speichern, kam man schnell zur Erkenntnis mit Datenbanksystemen eigene Software hierfür zur Verfügung zu stellen. Entscheidend dabei ist das Datenmodell, welches festlegt, wie die Daten gespeichert werden und welche Operationen auf ihnen zulässig sind. Zunächst dominierten in den 60er Jahren das hierarchische und Netzwerk-Modell (synonym: CODASYL-Modell), bevor in [Co70] das relationale Modell vorgeschlagen wurde. Wenige Jahre später standen die ersten kommerziellen Systeme zur Verfügung und in den 80er Jahren durchdrangen diese den Markt.

In den 90er Jahren kamen weitere Datenbanktypen an den Markt: Mit dem Aufkommen objektorientierter Sprachen und Modellierungsmethoden objektorientierte Datenbanken, durch zunehmenden Datenaustausch und das XML-Format bedingt XML-Datenbanken und aufgrund von Data Warehouses und analytischen Umgebungen Datenbanken mit multidimensionalem Datenmodell. Die jungen Produkte in allen drei Sparten konnten jedoch am Markt nicht gegen die mittlerweile sehr etablierten relationalen Datenbanken bestehen und verschwanden mehr oder weniger bald wieder. Vielmehr wurden in den relationalen Systemen Erweiterungen zu den genannten Aspekten vorgenommen. Damit galt das „One Size Fits All"-Paradigma, mit relationalen Systemen ein Typ für alle Zwecke. Dadurch ergab sich über eine Dekade ein sehr stabiler Markt, in dem die existierenden Systeme weitere Reife erlangten.

Das Abdecken aller Eigenschaften durch relationale Systeme stößt langsam an Grenzen und in jüngerer Vergangenheit ist ein Paradigmenwechsel [St10, St11] zu beobachten, der zur Bewegung NoSQL (Not Only SQL) geführt hat.

[1] Die Kapitel 1 und 2 dieses Beitrags sind identisch mit denen in [HR12].

Zeitgleich trat auch der Begriff Big Data auf die Bildfläche, wobei häufig falscherweise Big Data und NoSQL in einem Atemzug genannt werden. Dies ist aber nicht korrekt wie wir in Abschnitt 2 darlegen werden. Ebenso sei an dieser Stelle ausdrücklich erwähnt, dass Graphdatenbanken keine Erfindung der letzten Jahre oder der NoSQL-Bewegung sind, sondern es schon seit einigen Jahrzehnten Ansätze in diesem Umfeld gibt. Als erster publizierter Ansatz ist hier das System R&M [RM75] zu nennen. Hierin wurde ein semantisches Netzwerk zum Speichern von Metadaten vorgeschlagen, weil damalige Systeme typischerweise die Semantik der Datenbank nicht berücksichtigten. Den Startschuss einer lebhaften Entwicklungen diverser Vorschläge für Konzepte und Prototypen gab das Logische Datenmodell [KV84], in dem ein explizites Graphdatenmodell relationales, hierarchisches und Netzwerkmodell generalisieren sollte. Die hierauf aufbauenden Ansätze sollen hier nicht alle einzeln erwähnt werden, einen sehr guten Überblick gibt [AG08]. Mitte der 90er Jahre ebbte die Welle dann ab, parallel dem Ende der diversen Vorschläge objektorientierter Datenbanken. In den letzten Jahren entstand dann eine neue Welle von Systemen, wobei verschiedene Hersteller oder Communities sehr verschiedene Konzepte zur Realisierung einer Graphdatenbank verfolgen. Um hier den Überblick zu behalten und realistisch die Fähigkeiten und Grenzen von Systemen einschätzen zu können, wird in diesem Beitrag eine Klassifikationsschema für Graphdatenbanken vorgeschlagen und auf wichtige, aktuelle Systeme angewendet.

Der Rest des Papers ist folgendermaßen gegliedert: Im folgenden Abschnitt wird auf die beiden Begriffe Big Data und NoSQL eingegangen. Abschnitt 3 beschreibt das Arbeiten mit Neo4j. Der Beitrag schließt mit einer Zusammenfassung und einem Ausblick.

2 Big Data, NoSQL und Graphdatenbanken

Dieser Abschnitt soll die Begriffe Big Data und NoSQL definieren, den Bezug zueinander herstellen und die Einordnung von Graphdatenbanken in diesem Kontext vornehmen.

2.1 Big Data

Unter dem Begriff Big Data werden Daten verstanden, die in sehr großer Menge anfallen und dabei in den Aspekten Erfassung, Speicherung, Suche, Verteilung und Analyse existierende DB-Systeme an ihre Grenzen bringen.

Big Data werden häufig maschinell erzeugt, z.B.

- Applikations- und Web-Logs
- Netzwerk Monitoring
- Sensoren (z.B. RFID)
- Barcodeerfassung
- Intelligente Stromzähler (SmartMeter)

Ein Beispiel aus dem letztgenannten Anwendungsszenario soll den Begriff „Big“ verdeutlichen [Kö11]. Bei einem kalifornischen Energieversorger sind bisher 1 bis 2 Ablesungen pro Jahr notwendig. Um die Daten intelligenter Stromzähler für eine bedarfsgerechte Energienutzung verwenden zu können, sind Ablesungen im 15-Minuten-Takt notwendig, in der Endausbaustufe werden gar Ablesungen im Minutentakt angestrebt. Abbildung 1 stellt das Datenwachstum graphisch dar.

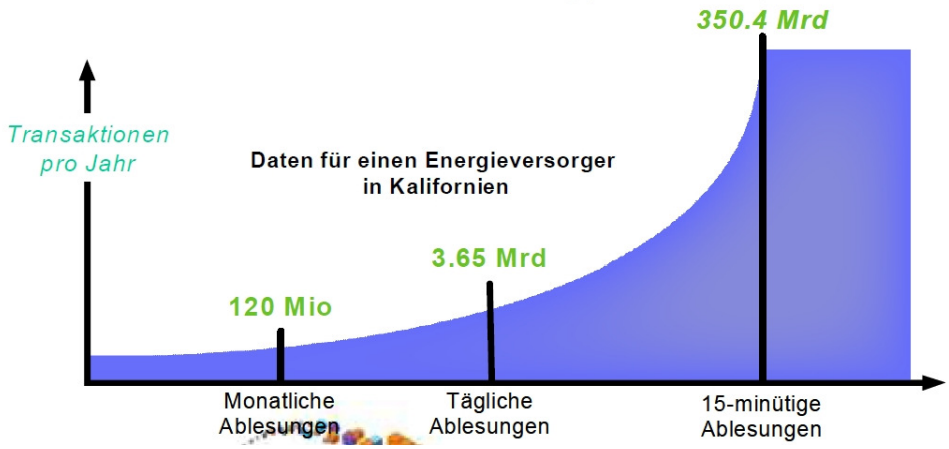

Abbildung 1: Zeilen- und spaltenorientierte Speicherung

Big Data können aber auch durch Personen erfasst werden, z.B. in Blogs, digitalen sozialen Netzwerken oder Online-Foren.

Big Data können dabei sowohl strukturiert als auch halbstrukturiert als auch unstrukturiert sein.

Als letzter Aspekt für Big Data sei hier noch die oft notwendige zeitnahe Verarbeitung der Daten erwähnt, um entsprechenden Nutzen aus den Daten ziehen zu können.

2.2 NoSQL

Unter dem Begriff NoSQL (Not only SQL) werden alle Datenbanken zusammengefasst, die nicht dem relationalen Modell folgen [EF+10, Ti11a, RW12]. Dadurch ergibt sich eine Sammlung sehr heterogener Lösungen. In der Literatur [EF+10] werden üblicherweise folgende Kategorien unterschieden:

- Schlüssel-Wert-(Key-Value-)basierte Datenbanken: Bei diesen Datenbanken wird unter einem eindeutigen Schlüssel ein einzelner Wert gespeichert, wobei prinzipiell die Struktur des Wertes von der Datenbank nicht interpretiert wird. Einige Lösungen weichen hiervon ab und bieten z.B. das Speichern von Listen und Mengen (Redis) oder die Gruppierung von Schlüssel-Wert-Paaren zu Domänen (Amazon Simple DB).

  Großer Pluspunkt der Schlüssel-Wert-basierten Systeme ist die einfache Verteilung großer Datenmengen auf mehrere Rechner. Demgegenüber wird die Strukturierung von Daten jedoch vernachlässigt.

  Bekannte Vertreter dieser Kategorie sind Amazon Simple DB [Am12], Chordless [Ch12], Riak [Ri12] und Redis [Re12].
- Wide Column Stores basieren auf Tabellen und bieten die Möglichkeit in einer Zeile mehrere Attribute zu speichern. Attributname und –wert bilden ein Schlüssel-Wert-Paar, das als Spalte bezeichnet wird. Wesentlicher Unterschied zu relationalen Tabellen ist die Nicht-Existenz eines Schemas, d.h. zu einer Zeile können beliebige Daten hinzugefügt bzw. weggelassen werden. Daraus ergibt sich auch das primäre Anwendungsgebiet von dünn besetzten Daten. In diese Kategorie fallen u.a. die Systeme Google BigTable [CD+06], Apache Cassandra [He10, Ca12], Apache HBase [Ge11, Hb12] und Hypertable [Hy12].

- In dokumentenorientierten DB-Systemen werden die Daten in Form semi-strukturierter Dokumente gespeichert, wobei Dokumente mithilfe eines Schlüssels gespeichert und geladen werden. Abfragen nach Dokumenten mit bestimmten Bestandteilen sind möglich. Anwendungsgebiet sind (semi-)strukturierte Inhalte, die Skalierbarkeit hingegen ist gegenüber den beiden ersten vorgestellten Ansätzen geringer. Die beiden Hauptvertreter dieser Kategorie sind CouchDB [ALS10, WK11, Co12] und MongoDB [CD10, Ba11, Mo12].

- Graphdatenbanken sind auf das Speichern und effiziente Traversieren von Graphen ausgelegt.

3 Arbeiten mit Neo4j

In diesem Abschnitt soll das Arbeiten mit Neo4j systematisch vorgestellt werden. Dabei wird zunächst das Anlegen einer Datenbank beschrieben, dann das Einfügen, Ändern und Löschen von Knoten und Kanten. Abschnitt 3.3 stellt als Alternative zum einzelnen Einfügen die Möglichkeit des Datenimports vor. Im darauffolgenden Abschnitt 3.4 wird die Indizierung eines Graphen vorgestellt. Abschnitt 3.5 schließlich beschreibt das Traversieren als Hauptoperation auf einem Graphen.

3.1 Starten und Beenden einer Datenbank

Zur Demonstration der Funktionalitäten von Neo4j dient für die folgenden Beispiele der in Abbildung 2 dargestellte Graph. Dieser stellt ein kleines digitales soziales Netzwerk dar, Knoten sind die Personen, durch Kanten werden die Beziehungen der Personen untereinander dargestellt, wobei es hier als einzigen Beziehungstypen „KNOWS“ gibt.

Abb. 2: Beispielgraph

Eine Datenbank wird wie in Listing 1 dargestellt angelegt: Als Attribute der Klasse werden ein Pfad auf den Speicherort (Zeile 3) und ein GraphDatabaseService-Objekt (Zeile 4) benötigt. In der main-Methode wird dann eine neue Instanz der Klasse angelegt, mit dem Factory-Pattern eine neue Datenbank angelegt, was hier in der createDb-Methode gekapselt ist. Anschließend kann mit der Graphdatenbank gearbeitet werden, das Beenden erfolgt durch Aufruf der shutdown-Methode auf dem GraphDatabaseService-Objekt, die in Listing 1 auch in einer gleichnamigen Methode gekapselt wird.

```
( 1) public class Example{
( 2)
( 3)    private static final String DB_PATH = "Path to database";
( 4)    GraphDatabaseService graphDb;
( 5)
( 6)    public static void main(String[] args){
( 7)        Example myExampleDatabase = new Example();
( 8)        myExampleDatabase.createDb();
( 9)        /* Work with database */
(10)        myExampleDatabase.shutDown();
(11)    }
(12)
```

```
(13)    void createDb(){
(14)     graphDb =
           new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
(15)    }
(16)
(17)    void shutDown(){
(18)      graphDb.shutdown();
(19)    }
(20) }
```

Listing 1: Anlegen und Löschen einer Datenbank

3.2 Knoten und Kanten anlegen und ändern

Als Vorbereitung muss das Interface RelationshipType implementiert werden, um gültige Werte für die Beziehungen zu definieren. Dies geschieht in Listing 2.

```
(1)Private static enum RelTypes implements RelationshipType{
(2)  KNOWS
(3)}
```

Listing 2: Relationstypen definieren

Darauf aufbauend können nun die Knoten und Kanten eingefügt werden, was mittels create-Methoden geschieht, die Eigenschaften werden mit setProperty-Methoden gesetzt. Wichtig ist hierbei, dass das Ganze in einem Transaktionsrahmen ausgeführt werden muss. Listing 3 fasst das Beispiel zusammen.

```
( 1)      Transaction tx = graphDb.beginTx();
( 2)        try{
( 3)            firstNode = graphDb.createNode();
( 4)            firstNode.setProperty( "name", "Frank" );
( 5)            secondNode = graphDb.createNode();
( 6)            secondNode.setProperty( "name", "Tanja" );
( 7)            relationship = firstNode.createRelationshipTo(
( 8)                                      secondNode, RelTypes.KNOWS );
( 9)            relationship.setProperty( "message", "..." );
(10)            tx.success();
(11)        }
(12)        finally{
(13)          tx.finish();
(14)        }
```

Listing 3: Relationstypen definieren

3.3 Datenimport

Das in Abschnitt 3.2 beschriebene Einfügen von Knoten und Kanten ist sehr aufwändig und durch die Transaktionen auch wenig performant. Aus diesem Grunde ist die Verwendung des Datenimporters interessant, der unter Umgehung von Transaktionen einen Bulk Load ausführen kann. Listing 4 demonstriert die Arbeitsweise.

```
( 1)  ArrayList<Long> nodes = new ArrayList<Long>();
( 2)  Map<String,Object> properties = new HashMap<String,Object>();
( 3)
( 4)  FileUtils.deleteRecursively( new File( DB_PATH ) );
( 5)  BatchInserter myInserter = BatchInserters.inserter(DB_PATH);
( 6)
( 7)  nodes.add(new Long(myInserter.createNode(properties)));
( 8)
( 9)  properties.clear();
(10)  properties.put("message", "dummy" );
(11)  myInserter.createRelationship(nodes.get(positionStartNode),
(12)      nodes.get(positionDestinationNode), RelTypes.KNOWS, properties);
```

Listing 4: Datenimport

Zunächst werden eine ArrayList für die Knoten und eine Map für die Properties angelegt. In Zeile 5 wird ein BatchInserter-Objekt instanziiert, Zeile 7 ff. zeigen das Einfügen von Knote und Properties.

Zur Verdeutlichung der Leistungsfähigkeit des Datenimporters dient folgende Untersuchung: Mit dem R-MAT-Algorithmus[GSP10] wurde ein Zufallsgraph erzeugt, der zwischen 32 und 128 Knoten und unterschiedliche Anzahlen von Kanten besitzt. Wie Abbildung 3 zeigt, steigt beim Einfügen der Daten über Transaktionen die Verarbeitungszeit (schon bei diesen relativ kleinen Graphen) schnell an, während sie für den Datenimport nahezu konstant bleibt.

Anzahl Knoten	Anzahl Kanten	Transaktion Zeit [sec]	Batch Zeit [sec]
32	10	2,287	8,354
32	70	5,173	7,018
32	150	9,051	8,483
32	1600	40,446	8,005
32	3200	50,672	7,803
64	20	4,482	8,134
64	140	10,892	9,001
64	300	18,021	7,839
64	3200	100,914	7,934
64	6400	146,3	7,959
128	40	9,082	7,995
128	280	21,334	9,116
128	600	35,2	7,96
128	6400	249,96	7,707
128	12800	411,014	8,539

Abb. 3: Werte für Datenimport

3.4 Indizierung

Zur Demonstration der Indizierung von Knoten soll das in Abbildung 4 skizzierte Szenario dienen: Es gibt eine Menge von Benutzern (Knoten mit dem Schlüssel-Wert-Paar („Username“,ID)), die einen Knoten UserReference referenzieren.

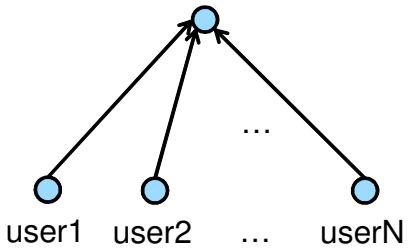

Abb. 4: Beispielgraph für Indizierung

In Ergänzung zu den schon bekannten Beispielen wird in Listing 5 in Zeile 2 ein Knotenindex angelegt. Beim Erzeugen der Knoten werden diese zusätzlich in diesen Index eingetragen (Zeile 7).

```
( 1)  graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
( 2)  nodeIndex = graphDb.index().forNodes( "nodes" );
( 3)  ...
( 4)  ... Erzeugen und Einfügen Knoten wie in Beispiel Listing 3
( 5)  Node node = graphDb.createNode();
( 6)  node.setProperty( USERNAME_KEY, username );
( 7)  nodeIndex.add( node, USERNAME_KEY, username );
( 8)  ...
( 9)  int idToFind = 45;
(10)  Node foundUser = nodeIndex.get( USERNAME_KEY,
(11)           idToUserName( idToFind ) ).getSingle();
(12)  System.out.println( "The username of user " + idToFind + " is "
(13)                + foundUser.getProperty( USERNAME_KEY ) );
```

Listing 5: Knotenindizierung und Anfrage

Zum Auslesen eines Wertes aus dem Index dient die get-Methode, die als Parameter das Schlüssel-Wert-Paar bekommt (Zeilen 1 und 2 in Listing 6). Diese liefert als Resultat ein Objekt der Klasse IndexHits, auf welchem iteriert werden oder alternativ wie im Beispiel mit der Methode getSingle nur der erste Treffer geliefert werden.

Liefert die Indexsuche mehrere Treffer, so kann über die Trefferliste iteriert werden.

```
( 1) IndexHits<Node> hits = nodeIndex.get(USERNAME_KEY,
( 2)                                          idToUserName(idToFind));
( 3) try{
( 4)   System.out.println("Found nodes: ");
( 5)   for (Node node : hits){
( 6)     System.out.println(node.getProperty(USERNAME_KEY));
( 7)   }
( 8) }
( 9) finally{
(10)   hits.close();
(11) }
```

Listing 6: Iteration über Index-Trefferliste

Zur Demonstration der Leistungsfähigkeit des Index soll folgendes Beispiel dienen. Es wurden für den Beispielgraphen aus Abbildung 4 verschiedene Knotenzahlen generiert und die gezielte Suche nach einem Knoten mit und ohne Index durchgeführt. Abbildung 5 zeigt das Anfragezeitverhalten. Bis in den unteren sechsstelligen Bereich macht sich der Index wenig bemerkbar, aber ab 500.000 Knoten nimmt die Anfragezeit ohne Index erheblich zu. Der Index hingegen skaliert sehr gut, auch im Bereich von 5 Millionen Knoten liegt die Antwortzeit noch deutlich unter einer Sekunde.

	Suchzeit [sec]	
Anzahl Knoten	Ohne Index	Mit Index
10.000	0,01	0,01
20.000	0,03	0,01
50.000	0,07	0,01
100.000	0,11	0,01
200.000	0,15	0,01
500.000	2,55	0,01
1.000.000	81,21	0,01
2.000.000	175,24	0,02
5.000.000	402,64	0,02

Abb. 5: Anfragezeit mit und ohne Index

	Einfügen [sec]			Speicherplatz [kB]			
Anzahl Knoten	Ohne Index	Mit Index	Faktor	Datenbank	Index	Summe	Faktor
10.000	1,40	2,60	1,86	861	678	1.539	2,27
20.000	2,12	4,04	1,91	1.622	1413	3.035	2,15
50.000	4,41	8,23	1,87	4.054	3570	7.624	2,14
100.000	8,32	15,32	1,84	8.106	7275	15.381	2,11
200.000	18,57	32,49	1,75	16.212	14790	31.002	2,10
500.000	65,02	107,94	1,66	40.529	37373	77.902	2,08
1.000.000	159,28	259,29	1,63	81.057	75985	157.042	2,07
2.000.000	340,31	562,32	1,65	162.112	152.474	314.586	2,06
5.000.000	815,92	1473,70	1,81	405.275	389.360	794.635	2,04

Abb. 6: Zeit und Platz für Erstellung eines Index

Der Preis für diese deutlich besseren Anfragezeiten wird durch die Zeit zur Indexerstellung und den benötigten Platz bezahlt. Dazu wurden Datenbanken in den entsprechenden Größen erstellt, jeweils 10000 Knoten en bloc in einer Transaktion eingefügt, in der indexlosen Variante nur in die Datenbank, in der anderen Variante auch in den Index. Die Erstellungszeiten sind im linken Teil der Abbildung 6 dargestellt, die Zeiten für die Indexerstellung sind ungefähr um den Faktor 1,8 höher. Der rechte Teil der Abbildung stellt den zusätzlichen Platzverbrauch für den Index dar, gemessen wurde hier die Größe des lucene-Ordners im Ordner index. Es ist zu erkennen, dass der Index ungefähr der Größe der Datenbank entspricht, d.h. es muss ungefähr der doppelte Speicherplatz zur Verfügung gestellt werden.

3.5 Traversierung

Als Traversieren wird das Besuchen aller oder einer bestimmten Teilmenge von Knoten eines Graphen verstanden, wobei die Menge und Reihenfolge der Knoten durch bestimmte Bedingungen festgelegt wird. Neo4j stellt hierfür das Traverser-Framework zur Verfügung. Bei dessen Verwendung wird zunächst ein TraversalDescription-Objekt erstellt und auf diesem die Art der Traversierung festgelegt, bevor mit der Methode traverse(startNode) die eigentliche Traversierung angestoßen wird.

Als Default werden alle Kantentypen berücksichtigt, mit der Methode relationships lassen sich diese einschränken, ebenso kann mit dieser Methode festgelegt werden, ob die Kanten in eine oder beide Richtungen traversiert werden sollen.

Evaluatoren werden verwendet, um zu entscheiden, ob die Traversierung fortgesetzt werden soll, und/oder der aktuelle Knoten in den Ergebnispfad aufgenommen werden soll. Hierbei stehen vier Möglichkeiten zur Verfügung:

- Evaluation.INCLUDE_AND_CONTINUE: Nimmt den Knoten ins Resultat auf und setzt die Traversierung fort.
- Evaluation.INCLUDE_AND_PRUNE: Nimmt den Knoten ins Resultat auf und beendet die Traversierung.
- Evaluation.EXCLUDE_AND_CONTINUE: Nimmt den Knoten nicht ins Resultat auf, setzt aber die Traversierung fort.
- Evaluation.EXCLUDE_AND_PRUNE: Nimmt den Knoten nicht ins Resultat auf und beendet die Traversierung.

Resultat einer Traversierung (Aufruf der Methode traverse auf einem TraversalDescription-Objekt) ist schließlich ein Traverser-Objekt. Es enthält neben der Traversierung eine Beschreibung des Formates des Resultats.

Die Methode uniqueness legt fest, ob bestimmte Positionen während einer Traversierung mehrfach besucht werden dürfen oder nicht. Hierfür gibt der Aufzählungstyp Uniqueness folgende Werte vor:

- NONE – Jede Position im Graphen darf erneut besucht warden.
- NODE_GLOBAL – Kein Knoten im gesamten Graphen darf mehr als einmal besucht werden. Dieses kann einen großen Speicherverbrauch nach sich ziehen, da eine interne Datenstruktur benötigt wird, die alle bereits besuchten Knoten vorhält.
- RELATIONSHIP_GLOBAL – Keine Kante im gesamten Graphen darf mehr als einmal besucht werden. Aus den oben genannten Gründen ist auch diese Option sehr speicherintensiv. Weil die meisten Graphen erheblich mehr Kanten als Knoten besitzen, tritt dies sogar noch verschärft auf.
- NODE_PATH – Ein Knoten darf in einem Pfad nicht schon einmal besucht worden sein.
- RELATIONSHIP_PATH - Eine Kante darf in einem Pfad nicht schon einmal besucht worden sein.
- NODE_RECENT – Ähnlich der Option NODE_GLOBAL gibt es eine Sammlung aller bereits besuchten Knoten. Allerdings wird hier der Speicher zum Merken bereits besuchter Knoten beschränkt, indem nur die zuletzt besuchten Knoten gespeichert werden. Quantifiziert wird die Menge zuletzt besuchter Knoten, indem bei Aufruf der TraversalDescription.uniqueness()-Methode ein zweites nummerisches Argument angegeben wird.
- RELATIONSHIP_RECENT – Wie Option NODE_RECENT, aber für Kanten statt Knoten.

Über den Parameter der order-Methode wird festgelegt, wie bei mehreren Möglichkeiten an einer bestimmten Stelle im Graphen verzweigt werden soll. Hier können entweder mit depthFirst und breadthFirst mit Tiefen- bzw. Breitensuche zwei standardmäßig implementierte Suchstrategien genannt werden. Außerdem kann der Anwender seine eigene Verzweigungsstrategie implementieren.

Ebenso kann die Verzweigung einer Traversierung beeinflusst werden, indem eine Option aus BranchSelector verwendet wird, zur Verfügung stehen hier

- Traversal.preorderDepthFirst() – Traversierung in Tiefensuche, besucht Knoten vor seinen Nachfolgern.
- Traversal.postorderDepthFirst() - Traversierung in Tiefensuche, besucht Knoten nach seinen Nachfolgern.
- Traversal.preorderBreadthFirst() - Traversierung in Breitensuche, besucht Knoten vor seinen Nachfolgern.
- Traversal.postorderBreadthFirst() - Traversierung in Breitensuche, besucht Knoten nach seinen Nachfolgern.

Ein BranchSelector besitzt einen Zustand und muss daher für jede Traversierung instanziiert werden. Deshalb wird er einem TraversalDescription-Objekt durch das BranchOrderingPolicy-Interface zugeordnet. Dieses wiederum ist eine Fabrik für BranchSelector-Instanzen.

Das Interface Path ist generischer Bestandteil des Neo4j API. In Bezug auf Traversierungen warden Pfade in zweierlei Hinsicht verwendet. Zum einen ist das Resultat einer Traversierung ein Path-Objekt, zum anderen werden Path-Objekte auch zur Evaluation von Positionen im Graphen benutzt, um zu entscheiden, ob die Traversierung fortzusetzen ist ode rein Knoten ins Resultat aufzunehmen ist oder nicht.

4 Zusammenfassung und Ausblick

In der jüngsten Vergangenheit hat sich ein gewisser Wechsel in der Datenbanklandschaft vollzogen. Waren bis vor kurzem nahezu nur relationale Datenbanken anzutreffen, so haben sich im Zuge der NoSQL-Bewegung einige andere Ansätze etabliert. Ein besonderer Typ von NoSQL-Datenbanken sind Graphdatenbanken, die das effiziente Abspeichern von und Arbeiten mit Graphen ermöglichen. Ein populärer Vertreter von Graphdatenbanken im Java-Umfeld ist Neo4j, diese Datenbank wurde in diesem Beitrag im Detail angeschaut.

In der Zukunft sollen weitere Untersuchungen an Graphdatenbanken durchgeführt werden, insbesondere sollen die Performanz und Skalierung verschiedener Systeme betrachtet werden. Auch soll hier ein Vergleich mit relationalen Datenbanken durchgeführt werden.

Kontaktadresse

Prof. Dr.-Ing. Olaf Herden
Duale Hochschule BW Campus Horb
Studiengang Informatik
Florianstr. 15
D-72160 Horb

Telefon: +49 (0) 7451-521 146
Fax: +49 (0) 7451-521 101
E-Mail o.herden@hb.dhbw-stuttgart.de
Internet: www.dhbw-stuttgart.de/horb

Referenzen

[AG08] Renzo Angles, Claudio Gutiérrez: Survey of graph database models. ACM Computing Surveys. 40(1): (2008).

[ALS10] Anderson, J.C., Lehnardt, J. und Slater, N.: CouchDB: The Definitive Guide. O'Reilly Media, 2010.

[Am12] http://aws.amazon.com/de/simpledb/.

[Ba11] Banker, K.: MongoDB in Action. Manning-Verlag, 2011.

[Ca12] http://cassandra.apache.org/.

[CD+06] Chang F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra T., Fikes A., Gruber R.: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006: 205-218.

[CD10] Chodorow, K. und Dirolf, M.: MongoDB: The Definitive Guide. O'Reilly Media, 2010.

[Ch12] http://sourceforge.net/projects/chordless/.

[Co12] http://couchdb.apache.org/.

[Co70] Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970).

[EF+10] Edlich, S., Friedland, A., Hampe, J., Brauer, B. und Brückner, M.: NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Erste Auflage, Hanser-Verlag, München, 2010.

[Ge11] George, L.: HBase: The Definitive Guide. O'Reilly, 2011.

[GSP10] Groer, C., Sullivan, B., und Poole, S.: A Mathematical Analysis of the R-MAT Random Graph Generator. Oak Ridge National Laboratory, 2010.

[Hb12] http://hbase.apache.org/.

[He10] Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly, 2010.

[HR12] Herden, O. und Redenz, K.: Open Source Graphdatenbanken - Konzepte und Klassifikation. Tagungsband GI-Jahrestagung, Braunschweig, 2012.

[Hy12] http://hypertable.org/.

[Kö11] Körner, A.: Geschäftsinnovationen mit Zeitreihendaten: SmartMeter-, Sensor- und Finanz-Daten erschließen. Vortragsfolien, IBM Information Management Forum, Darmstadt, 2011. Online verfügbar unter ftp://ftp.software.ibm.com/software/emea/de/ telefonkonferenz/2011-07-15_SWG_PartnerUniversity_Informix_Zeitreihen_v3.pdf

[KV84] Kuper, G.M. und Vardi,M. Y. A new approach to database logic. In Proceedings of the 3th Symposium on Principles of Database Systems (PODS). ACM Press, S. 86-96. Waterloo (Ontario, Kanada), 1984.

[Mo12] http://www.mongodb.org/.

[Re12] http://redis.io/.

[Ri12] http://wiki.basho.com/Riak.html.

[RM75] Roussopoulos, N. und Mylopoulos, J. Using semantic networks for database management. In Proceedings of the 1st International Conference on Very Large Data Bases (VLDB), S. 144-172. Framingham (Massachusetts, USA), 1975.

[RW12] Redmond, E. und Wilson, J.R.: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Programmers Publisher, 2012.

[St10] Stonebraker, M.: SQL databases v. NoSQL databases. Communications of the ACM (CACM) 53(4):10-11 (2010).

[St11] Stonebraker, M.: Stonebraker on NoSQL and enterprises. Communications of the ACM (CACM) 54(8):10-11 (2011).

[Ti11a] Tiwari, S.: Professional NoSQL (Wrox Programmer to Programmer). John Wiley & Sons, 2011.

[WK11] Wenk, A. und Klampäckel, T.: CouchDB: Das Praxisbuch für Entwickler und Administratoren. Galileo Computing, Bonn, 2011.

Für alle genannten Online-Quellen gilt: Letzter Abruf am 10.9.2012.