

Masterseminar: Neo4j

- Wintersemester 2020/2021

Leander Féret

Matrikelnummer 631926

Inhaltsverzeichnis

1. Neo4j.....	3
1.1 Property Graph Model.....	3
1.1.1 Nodes.....	4
1.1.2 Edges (Relationships).....	5
1.1.3 Properties.....	6
2. Storage.....	8
2.1 Nodestore.....	9
2.2 Relationshipstore.....	9
2.3 ID-Dateien.....	10
3. Cypher.....	11
3.1 Basic Syntax.....	11
3.2 Clauses.....	12
3.2.1 CREATE.....	12
3.2.2 DELETE.....	13
3.2.3 MATCH.....	14
4. Wann Neo4j?.....	15
4.1 Neo4j Features.....	15
4.1.1 Clustering.....	15
4.1.2 Neo4j Browser.....	16
4.1.3 Cache Sharding.....	16
4.1.4 Leichter Einstieg.....	16
4.2 Use-Cases.....	16
4.2.1 Social Networks.....	16
4.2.2 Matchmaking.....	17
4.2.3 Network Management.....	17
4.2.4 Recommendation Engines.....	17
4.3 Häufige Fehlritte.....	18
4.3.1 Relationale Model-Technik.....	18
4.3.2 First-Time Nutzung an etwas kritischem.....	18
4.3.3 Speichern von Entitäten und Relationen innerhalb einer Entität.....	18
4.3.4 Falsche Nutzung von Relationship-Types.....	19
4.3.5 Speichern großer binärer Objekte.....	19
7. Bildquellen.....	21
6. Anhang.....	22
Installation: Neo4j (Windows).....	22
Installation: Python3 (Windows).....	22

1. Neo4j

Neo4j ist eine Graph-Datenbank und reiht sich damit in die Reihen der NoSQL-Datenbanken ein. Neo4j ist ein OpenSource-Projekt welches 2007 veröffentlicht wurde und von der Firma Neo4j Inc. finanziert wird. Version 1.0 von Neo4j wurde erst 2010 veröffentlicht. Neben Neo4j spezifischen Eigenschaften überzeugt Neo4j vor allem durch die native Umsetzung des „Property Graph Model“s.

1.1 Property Graph Model

Das Property-Graph-Model setzt sich zusammen aus drei Kernelementen:

1. Nodes
2. Edges/Relationships
3. Properties

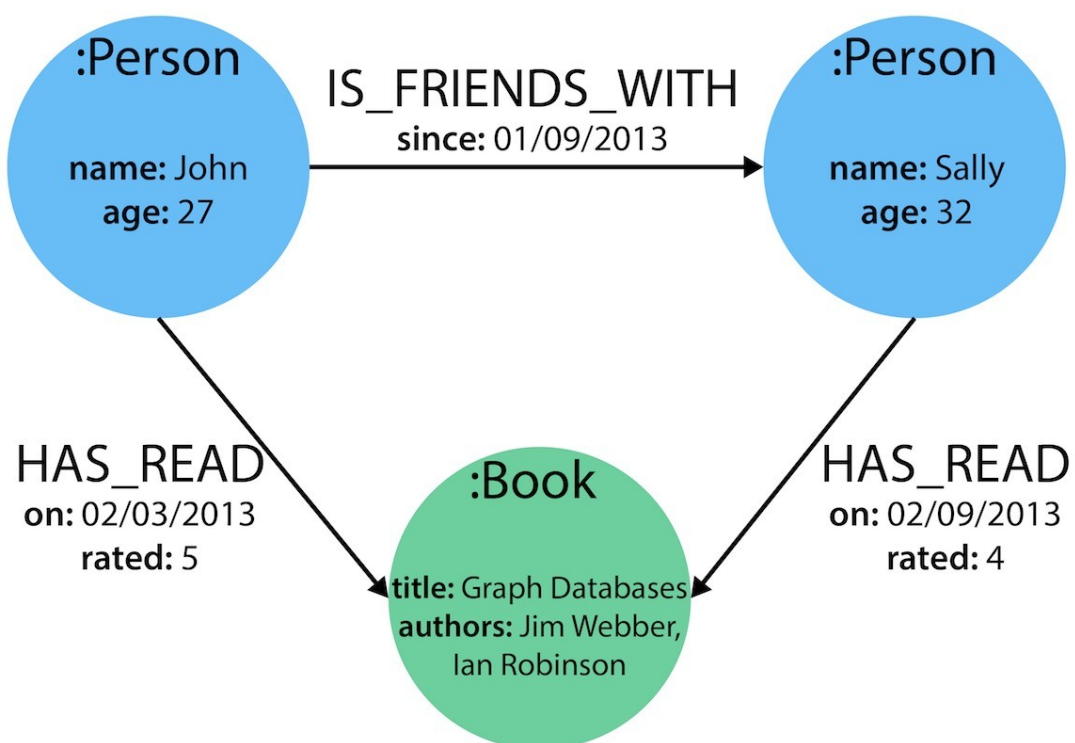


Schaubild 1: Umsetzung eines Property-Graph-Models

1.1.1 Nodes

Die Nodes (Knoten) in einem Property-Graph-Model stellen Entitäten dar. In Schaubild 2 sind drei einfache Nodes ohne Label oder Property dargestellt.

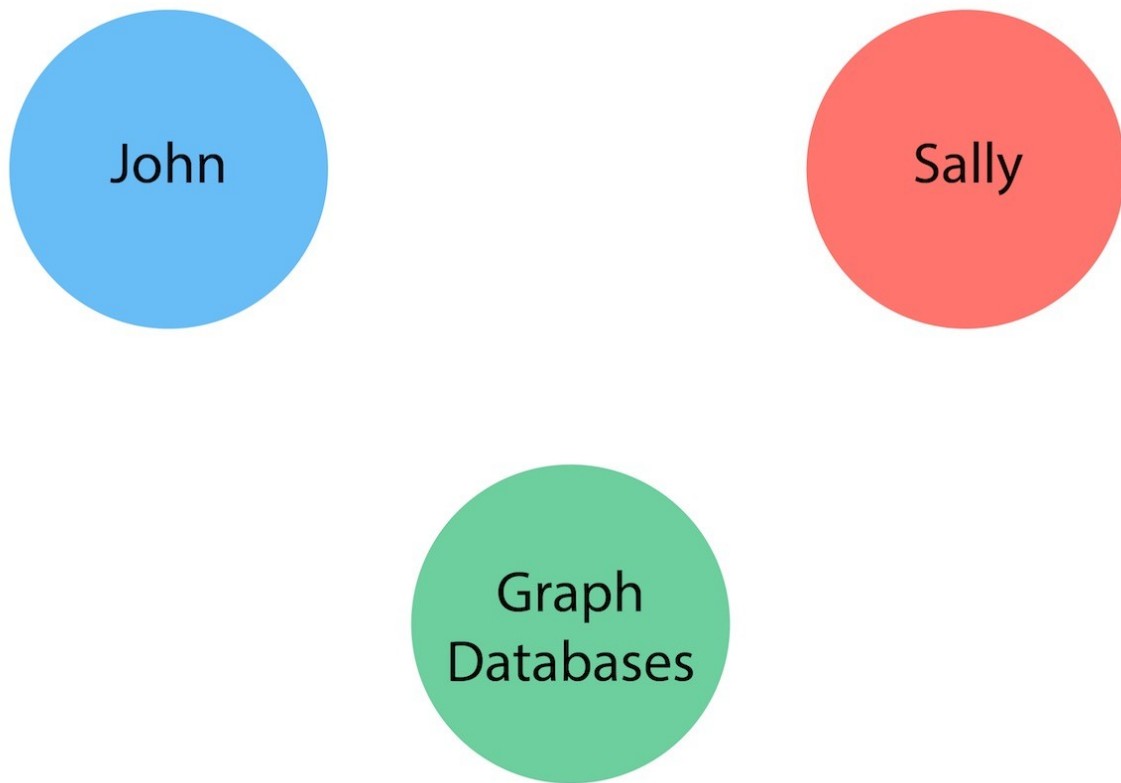


Schaubild 2: Nodes

Diese entsprechen dem, was man aus der objektorientierten Programmierung als Objekte kennt, oder aus den SQL-Datenbanken als Reihe in einer Tabelle. Daher sollte verstanden werden, dass diese keine Klassen oder Tabellen darstellen. Man erstellt also keinen Node namens Personen und hängt an diesem alle Personen der Datenbank, sondern erzeugt für jede Person einen Node.

Um etwas analoges zu Klassen oder Tabellen zu haben, oder abstrakter eine Kategorisierung der Nodes vornehmen zu können, bieten diese sogenannte Label an. Anders als Klassen und Tabellen kann ein Node mehreren Labeln zugeordnet sein. Sie übernehmen also die Funktionen von Klassen und Tabellen, sind aber so gesehen mächtiger, da man mehrere Nodes über diese in unterschiedliche Kategorien unterteilen kann. Label definieren auch anders als Klassen und Tabellen keine Struktur der zugrunde liegenden Nodes. Sie dienen einzig und allein, der Spezifizierung, für Suchzugriffe und dem Verständnis der Datenstruktur.

Nodes mit Labeln werden im Schaubild 3 dargestellt. Hierbei werden die Nodes „John“ und „Sally“ mit dem Label „Person“ versehen. Dies wird auch noch einmal über die selbe Farbgebung unterstrichen.

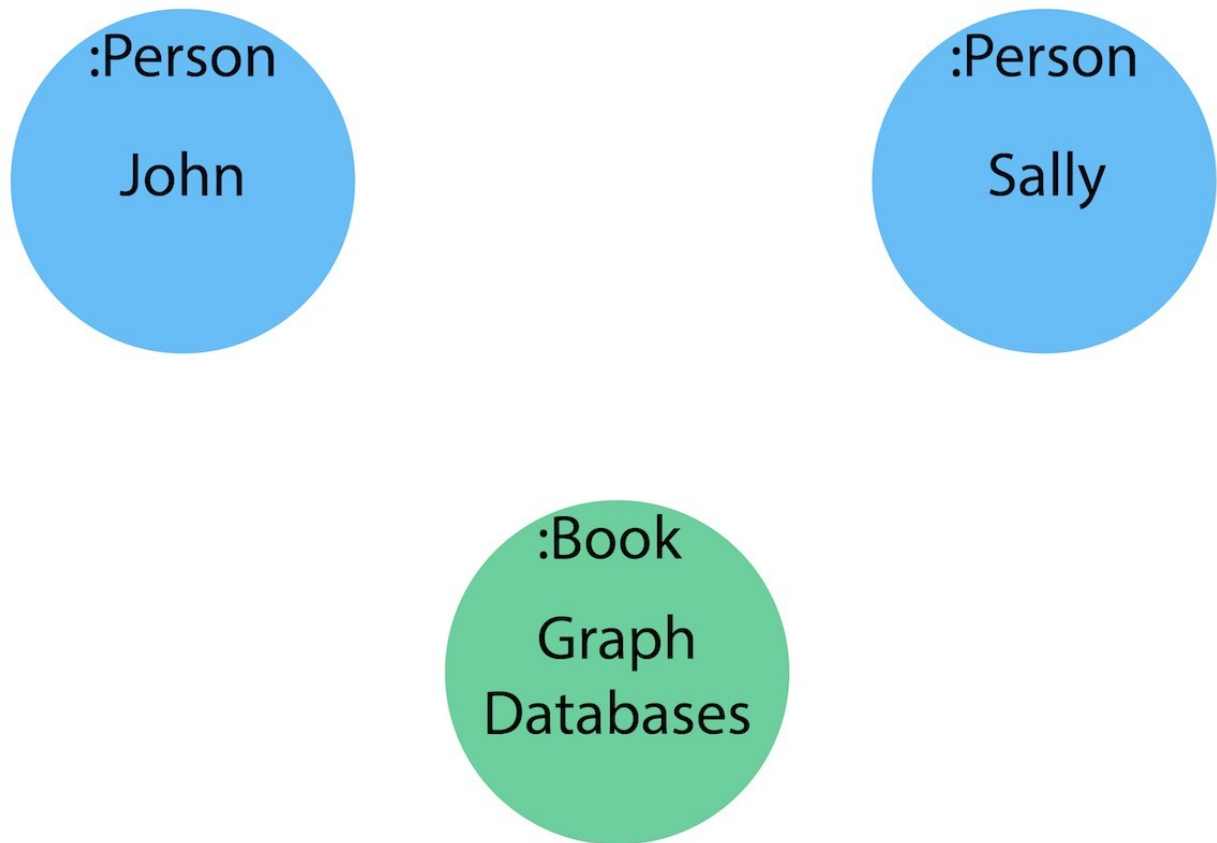


Schaubild 3: Nodes mit Labeln

Nodes können Properties zugewiesen werden. Dies wird im Abschnitt 1.1.3 Properties näher beschrieben.

Nodes werden im sogenannten Nodestore gespeichert.

1.1.2 Edges (Relationships)

Die Verbindungen zwischen den einzelnen Nodes werden im Property-Graph-Model als Edges oder Relationships bezeichnet.

In Schaubild 4 sind zwischen drei Nodes drei Relationships zu sehen, in diesem Fall als mit „IS_FRIENDS_WITH“ und „HAS_READ“ bezeichnet. Relationships haben immer eine Richtung und immer genau einen Start- und Endnode.

Jeder Node kann von 0 bis n eingehende und ausgehende Relationships besitzen. Jede Relationship kann vorwärts und rückwärts abgefragt werden. Relationships können beliebig viele Typen besitzen, die in etwa die selbe Aufgabe wie Label für Nodes übernehmen.

Relationships können Properties zugewiesen werden und werden im relationshipstore gespeichert.

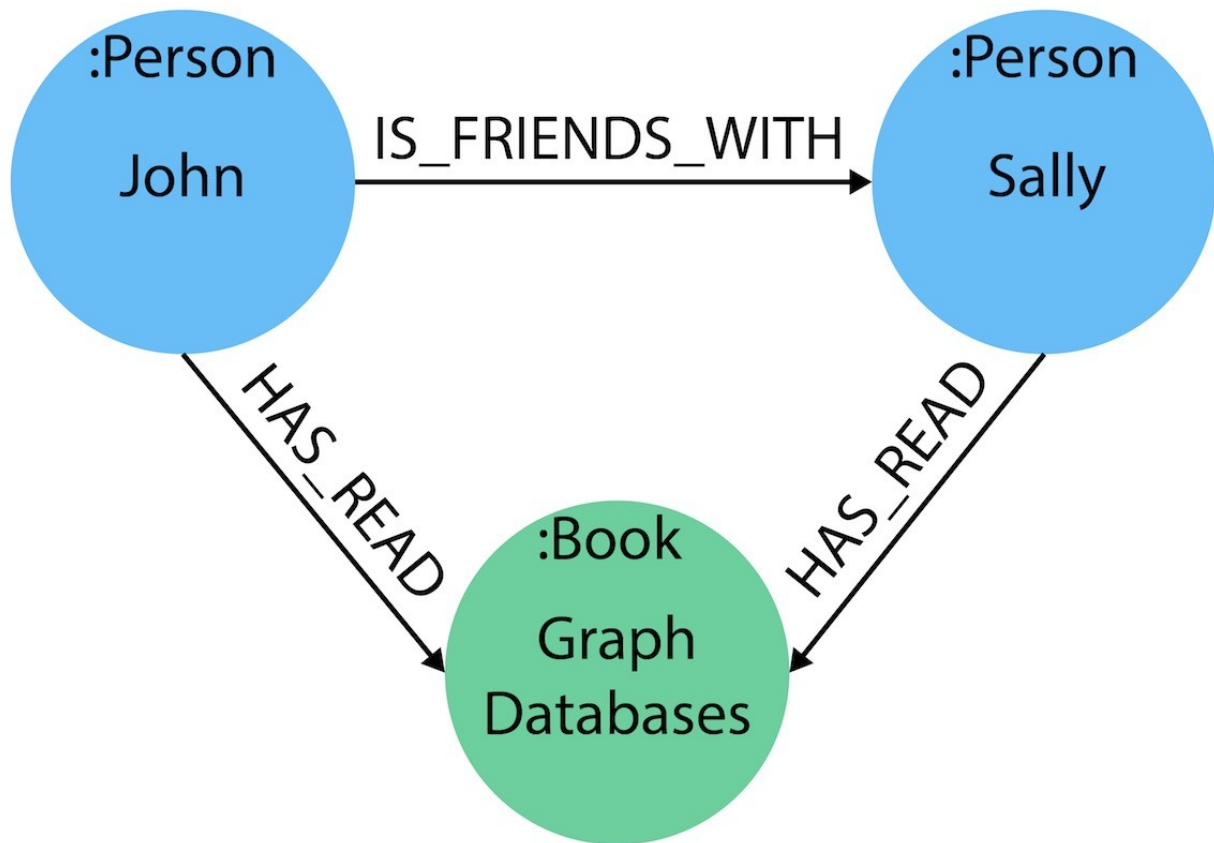


Schaubild 4: Nodes mit Relationships

1.1.3 Properties

Properties (Eigenschaften) können nur in Kombination von entweder Nodes oder Relationships erstellt werden. Sie beschreiben das jeweilige Element mit Hilfe von Pair-Value Einträgen.

Wenn Nodes die Reihen einer SQL-Tabelle wären, wären die Properties die Spalten mit zugehörigen Wert der Reihe.

Nodes und Relationships können beliebig viele Properties besitzen, aber es ist ratsam, diese nicht zu überladen, sondern stattdessen neue Nodes anzulegen und sie mit Relationships zu verbinden.

In Schaubild 5 sind sowohl den Nodes, als auch den Relationships Properties angehängt worden. Hierbei sieht man nun auch, dass nun das Beispiel aus Schaubild 1 zu Beginn dieses Dokuments vervollständigt ist. Alle Basisbausteine des Property-Graph-Models sind nun also vorgestellt.

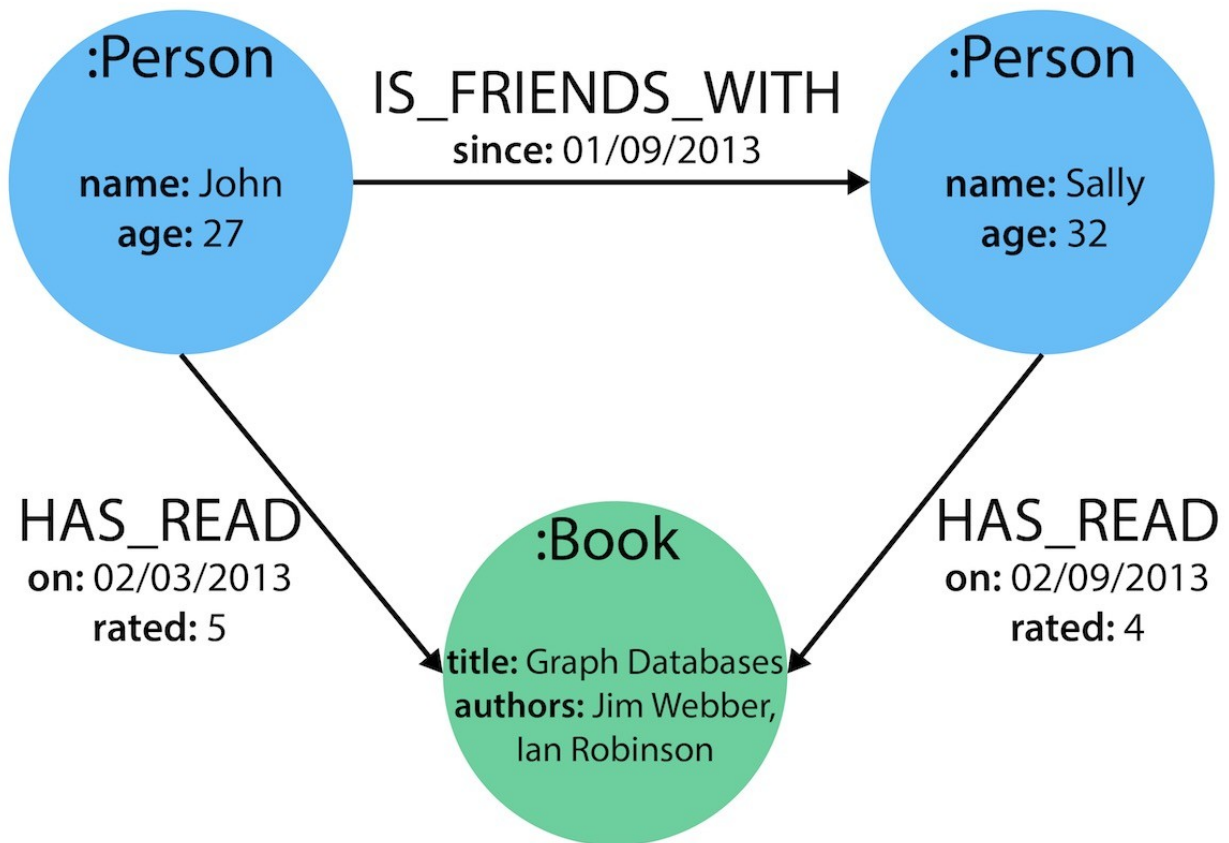


Schaubild 5: Property Graph Model mit Properties an Nodes und Relationships

Wie auch für die Nodes und die Relationships haben Properties ihren eigenen Storage namens **propertystore**.

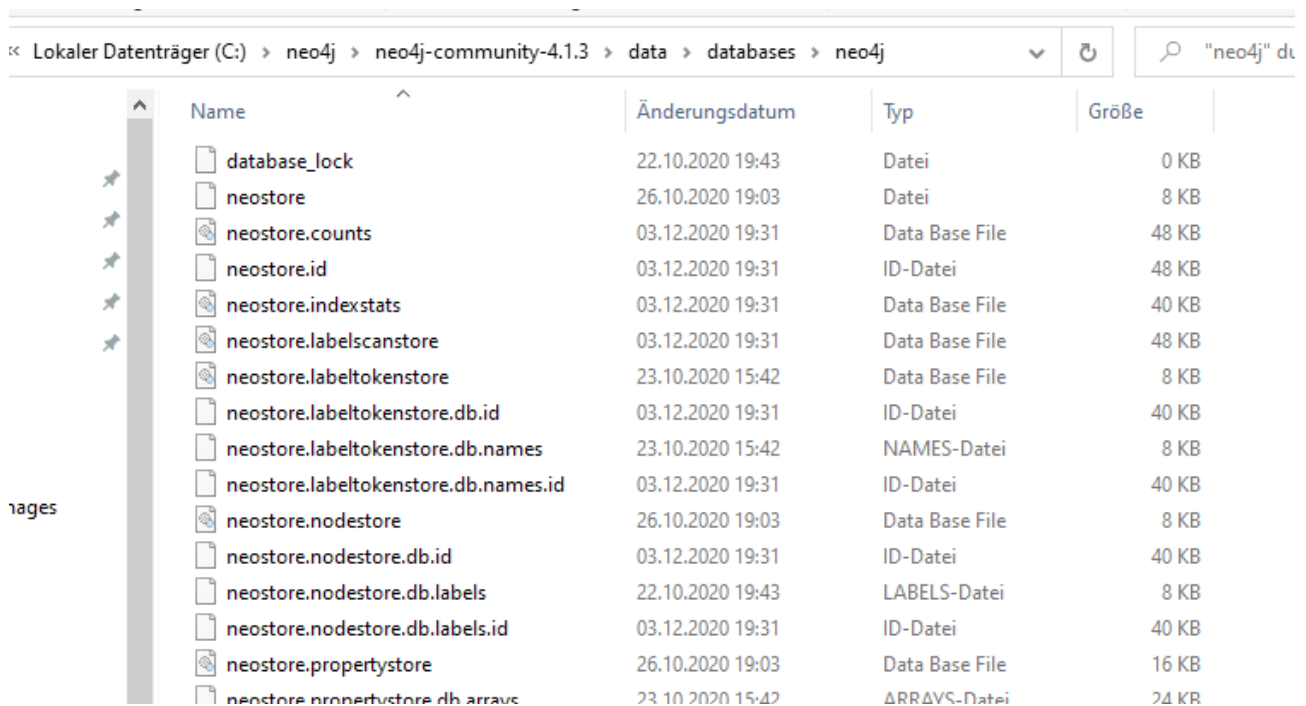
Einige dieser Storages werden im nächsten Abschnitt näher betrachtet.

2. Storage

Neo4j ist eine NoSQL-Datenbank und hat einen Datei-basierten Speichermechanismus. Es wird keine einzelne Datei zum Abspeichern aller Daten erzeugt, sondern viele verschiedene Dateien.

In Schaubild 6 werden einige Dateien angezeigt, die im Neo4j Installationsverzeichnis unter „data“ und „databases“ angelegt werden.

Diese Daten enthalten in Binary-Data die Einträge für die Java-Objekte der Neo4j Engine.



Name	Änderungsdatum	Typ	Größe
database_lock	22.10.2020 19:43	Datei	0 KB
neostore	26.10.2020 19:03	Datei	8 KB
neostore.counts	03.12.2020 19:31	Data Base File	48 KB
neostore.id	03.12.2020 19:31	ID-Datei	48 KB
neostore.indexstats	03.12.2020 19:31	Data Base File	40 KB
neostore.labelscestore	03.12.2020 19:31	Data Base File	48 KB
neostore.labeltokenstore	23.10.2020 15:42	Data Base File	8 KB
neostore.labeltokenstore.db.id	03.12.2020 19:31	ID-Datei	40 KB
neostore.labeltokenstore.db.names	23.10.2020 15:42	NAMES-Datei	8 KB
neostore.labeltokenstore.db.names.id	03.12.2020 19:31	ID-Datei	40 KB
neostore.nodestore	26.10.2020 19:03	Data Base File	8 KB
neostore.nodestore.db.id	03.12.2020 19:31	ID-Datei	40 KB
neostore.nodestore.db.labels	22.10.2020 19:43	LABELS-Datei	8 KB
neostore.nodestore.db.labels.id	03.12.2020 19:31	ID-Datei	40 KB
neostore.propertystore	26.10.2020 19:03	Data Base File	16 KB
neostore.propertystore.db.arrays	23.10.2020 15:42	ARRAYS-Datei	24 KB

Schaubild 6: Ausschnitt aus Neo4j gespeicherten Daten im Windows Explorer

Sämtliche Dateien zu behandeln, würden den Rahmen dieses Dokumentes sprengen, daher wird nur auf eine Auswahl näher eingegangen.

1. Nodestore (neostore.nodestore) - Data Base File
2. Relationshipstore (neostore.relationshipstore) – Data Base File
3. Id-Datei (allgemein)

2.1 Nodestore

Alle Nodes einer Neo4j-Datenbank werden im Nodestore gespeichert. Jeder Eintrag ist ein Binary-Eintrag, der von Neo4j in ein Java-Objekt umgewandelt werden kann. Die Größe eines Eintrags ist fix 15 Bytes (Versionsunterschiede können auftauchen).

1 Byte	4 Bytes	1 Byte	1 Byte	5 Bytes	3 Bytes
IsInUse	Node ID	First Relationship ID	First Property ID	Label Store	For Future Use

Die Größe wird dadurch ermöglicht, dass ein Node nur eine Art Pointer zu seinen anderen Daten darstellt. So sind alle zugehörigen Daten in anderen Dateien gespeichert und der Node verweist nur auf diese. Ein Vorteil einer solchen Speicherung ist, dass ein Heraussuchen eines Nodes innerhalb der Datei sehr einfach ist. Wenn man zum Beispiel auf den neunten Eintrag der Datei zugreifen will, muss man nur $8 * 15$ Bytes berechnen und kommt zum gewünschten Node. Eine solche Indexierung innerhalb der Datei ist extrem schnell und einfach. Dadurch wird der Zugriff auf bestimmte Nodes kaum von der Größe der Datei beeinflusst (also von der Anzahl der Nodes).

Auch eine Relationship hat eine feste Größe, wodurch oben beschriebenes Indexing schnell und einfach möglich ist. Dadurch ermöglicht das Speichern der ID der ersten Relationship einen sehr schnellen Zugriff auf die entsprechende Relationship.

2.2 Relationshipstore

Ein Eintrag im Relationshipstore hat eine feste Größe mit 34 Bytes (Versionenunterschiede können auftauchen). Auch eine Relationship ist unterteilt in feste Segmente unter anderen die folgenden.

Start-Node ID	End-Node ID	Pointer: Type	Pointer zur nächsten und vorigen Relationship des Start- und auch des Endnodes!
---------------	-------------	---------------	---

Jede Relationship besitzt einen Start- und einen Endnode und somit eine intrinsische Richtung. Dennoch sind Relationships bidirektional abfragbar.

Zu beachten ist auch, dass der Relationship-Type (ähnlich dem Node-Label) als Pointer gespeichert wird. Dadurch ist der Zugriff extrem schnell über Pointer-Following. Noch wichtiger zu beachten ist, dass jede Relationship einer Doppelverketteten-Liste ähnelt, insofern, als dass sie zur vorigen und nächsten Relationship zeigt. Dies gilt für den Start- als auch den Endnode. Hierbei sollte auch

verstanden werden, dass diese jeweils Pointer sind, denen über Pointer-Following extrem schnell gefolgt werden kann. Dadurch ist das Abfragen über mehrere Relationships hinweg sehr schnell und Neo4j den SQL-Datenbanken beim Abfragen von Relationships in der Geschwindigkeit deutlich überlegen.

Neo4j besitzt obendrein noch einen LRU k-Page Cache, der einen großen Anteil an Daten im Cache hält und immer die am wenigsten genutzten Einträge entfernt.

2.3 ID-Dateien

Zu beachten sind hier noch die ID-Dateien. Diese speichern die IDs von gelöschten Objekten (beim Node zum Beispiel über „IsInUse“-Byte zu Beginn gekennzeichnet). Das Objekt wird nicht sofort aus der Datei entfernt, sondern nur als „gelöscht“ markiert und die ID wird als frei gekennzeichnet, indem sie in die zugehörige ID-Datei gespeichert wird. Wenn nun ein neues Objekt, zum Beispiel ein neuer Node erzeugt wird, wird zunächst in der ID-Datei geschaut, ob dort eine ID frei ist. Ist dies der Fall, so wird eine der freien genommen und aus der ID-Datei entfernt. Der entsprechende Eintrag im Store wird mit dem neuen überschrieben und das „IsInUse“-Byte wird entsprechend wieder auf „InUse“ gesetzt.

Dies beschleunigt das Löschen von Datensätzen und vereinfacht die Verwaltung von IDs.

Zu beachten ist dadurch aber, dass beim Löschen von Datensätzen, die Größe der Datenbank nicht schrumpft, sondern sogar zunimmt, da ein neuer Eintrag in der ID-Datei erstellt wird.

3. Cypher

Neo4j kommt mit seiner eigenen Abfrage-Sprache namens „Cypher“. Cypher ist eine deklarative Sprache, die sich an der Syntax von SQL, Python, Haskell und der englischen Sprache orientiert. Sie wird genutzt um die typischen CRUD-Anweisungen zu bieten, also Erstellen, Bearbeiten, Löschen und Abfragen von Daten der Datenbank.

Die Sprache ist vom Menschen lesbar und intuitiv verständlich (auch wenn ohne Vorwissen es etwas dauern kann, bis man die grundlegende Syntax extrahiert hat).

Neben der grundlegenden Syntax, wie Nodes, Relationships und Properties (alle nachfolgend erläutert) und vieler weiterer Syntax (nachzulesen unter: <https://neo4j.com/docs/cypher-manual/current/syntax/>), bietet Cypher sogenannte Clauses (Klauseln). Diese werden mit der grundlegenden Syntax kombiniert um Abfragen zu erzeugen.

3.1 Basic Syntax

Die basic Syntax von Cypher ist weitläufig. Hier wird sich auf eine Auswahl an „Pattern“-Syntax beschränkt, die wohl am wichtigsten ist, zu verstehen.



Schaubild 7: Auswahl an Beispielen der Basic Syntax

<code>()</code> → Node	<code>{name:"value"}</code> → Property	<code>- ></code> → Relationship
<code>-[]-</code> → Relationship-Data	<code>(:Label)</code> → Node-Label	<code>[:Type]</code> → Relationship-Type

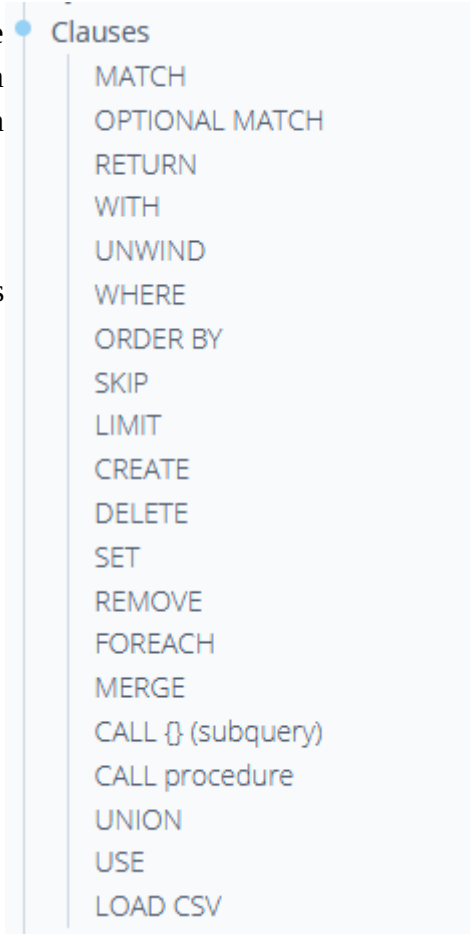
3.2 Clauses

Auf dem ersten Blick sieht man, dass Neo4j wirklich viele Clauses anbietet, zu viele für dieses Dokument. Daher wird sich auf eine Auswahl der für dieses Dokument relevanten beschränkt. Alle Clause können bei Bedarf unter:

<https://neo4j.com/docs/cypher-manual/current/clauses/>

nachgeschlagen werden. Wir betrachten die folgenden Clauses näher mit einigen Beispielen und Erläuterungen:

1. CREATE
2. DELETE
3. MATCH



Clauses

- MATCH
- OPTIONAL MATCH
- RETURN
- WITH
- UNWIND
- WHERE
- ORDER BY
- SKIP
- LIMIT
- CREATE
- DELETE
- SET
- REMOVE
- FOREACH
- MERGE
- CALL {} (subquery)
- CALL procedure
- UNION
- USE
- LOAD CSV

Schaubild 8: Neo4j Clauses

3.2.1 CREATE

Zum Erstellen von Nodes und Relationships, sowie Properties und Label von diesen wird die CREATE-Clause verwendet. Wichtig zu verstehen ist, das CREATE alle Objekte erzeugt, die ihm übergeben werden. Wenn also ein „Path“, also eine Verknüpfung von Nodes über mehrere Relationships übergeben wird, so werden alle noch nicht vorhandenen Elemente des Paths von CREATE erstellt. Dies beinhaltet auch potentielle Properties, Labels und Relationship-Types.

Die erzeugten Daten können über die RETURN-Clause ausgegeben werden.

<code>CREATE (n)</code>	→	Erzeugt einzelnen <u>Node</u>
<code>CREATE (n),(m)</code>	→	Erzeugt zwei <u>Nodes</u>
<code>CREATE (n:Person)</code>	→	Erzeugt <u>Node</u> mit Label „Person“
<code>CREATE (n:Person:Swedish)</code>	→	Erzeugt <u>Node</u> mit Label „Person“ und „Swedish“
<code>CREATE (n:Person { name: 'Andy', title: 'Developer' })</code>	→	Erzeugt <u>Node</u> mit Label und zwei <u>Properties</u>
<code>CREATE (a { name: 'Andy' }) RETURN a.name</code>	→	Erzeugt <u>Node</u> und gibt <u>Property</u> „name“ zurück

Schaubild 9: CREATE Nodes Beispiele

<pre>MATCH (a:Person),(b:Person) WHERE a.name = 'A' AND b.name = 'B' CREATE (a)-[r:RELTYPE]->(b) RETURN type(r)</pre>	→	Erzeugt <u>Relationship</u> zwischen <u>Nodes</u>
<pre>MATCH (a:Person),(b:Person) WHERE a.name = 'A' AND b.name = 'B' CREATE (a)-[r:RELTYPE { name: a.name + '<->' + b.name }]->(b) RETURN type(r), r.name</pre>	→	Erzeugt <u>Relationship</u> mit <u>Property</u>

CREATE erzeugt alle fehlende Bestandteile eines Patterns!

<pre>CREATE p =(andy { name: 'Andy' })-[:WORKS_AT]->(neo)-[:WORKS_AT]->(michael { name: 'Michael' }) RETURN p</pre>	→	<p>p</p> <pre>(2)-[:WORKS_AT,0]->(3)-[:WORKS_AT,1]->(4)</pre> <p>1 row, Nodes created: 3 Relationships created: 2 Properties set: 2</p>
---	---	---

Schaubild 10: CREATE Relationship & Path Beispiele

3.2.2 DELETE

Es gibt neben DELETE noch die Clause REMOVE, die zum gezielten Entfernen von zum Beispiel Labeln, oder Relationship-Typen genutzt wird. DELETE dient dem Löschen von Nodes und Relationships.

Hierbei ist zwingend zu beachten, dass wenn ein Node entfernt wird, auch alle Relationships die auf den Node verweisen und vom Node ausgehen entfernt werden müssen. Dies kann manuell über DELETE gemacht werden, oder aber mit Hilfe von DETACH, welche automatisch alle Relationships des Nodes entfernt.

```
MATCH (n:Person { name: 'UNKNOWN' })
DELETE n
```

→ Löscht einzelnen Node

```
MATCH (n)
DETACH DELETE n
```

→ Löscht alle Nodes und Relationships

```
MATCH (n { name: 'Andy' })
DETACH DELETE n
```

→ Löscht einen Node mit allen Relationships

```
MATCH (n { name: 'Andy' })-[r:KNOWS]->()
DELETE r
```

→ Löscht eine Relationship

Schaubild 11: DELETE Clause Beispiele

3.2.3 MATCH

Für die meisten Abfragen auf die Daten, um diese auf bestimmte Dinge zu Filtern wird MATCH verwendet. MATCH vergleicht das übergebene Pattern (siehe basic Syntax) und prüft diesen auf mögliche Treffer. MATCH wird in aller Regel mit RETURN zusammen genutzt, um die gefundenen Daten auch wieder auszugeben.

```
MATCH (n)
RETURN n
```

→ Hole alle Nodes

```
MATCH (movie:Movie)
RETURN movie.title
```

→ Hole alle Nodes mit Label „Movie“

```
MATCH (director { name: 'Oliver Stone' })--(movie)
RETURN movie.title
```

→ Hole alle Nodes verknüpft zu Nodes mit Namen „Oliver Stone“

```
MATCH (:Person { name: 'Oliver Stone' })--(movie:Movie)
RETURN movie.title
```

→ Hole alle Nodes mit Label „Movie“ verknüpft mit Nodes mit Label „Person“ mit Namen „Oliver Stone“

```
MATCH (:Person { name: 'Oliver Stone' })-[r]->(movie)
RETURN type(r)
```

→ Hole alle ausgehende Verknüpfungen von Nodes mit dem Namen „Oliver Stone“

Schaubild 12: MATCH Clause Beispiele

4. Wann Neo4j?

Nachfolgend werden vom Property-Graph-Model unabhängige Eigenschaften von Neo4j behandelt und der Frage nachgegangen, wann man Neo4j einsetzen sollte und was die häufigsten Fehlritte in der Arbeit mit Neo4j sind.

4.1 Neo4j Features

Die nachfolgenden vier Eigenschaften sind unabhängig vom Property-Graph-Model, welches zuvor erläutert wurde, Bestandteil von Neo4j. Diese können als Gründe für die Nutzung von Neo4j dienen.

4.1.1 Clustering

Die Enterprise Edition von Neo4j bietet horizontale Skalierung über zwei Methoden des Clusterings.

1. High-Availability Clustering

Mehrere Slaves gehorchen einem ausgesuchten Master. Im Fall des Ausfalls eines Masters, wird ein neuer Master gewählt.

2. Causal Clustering

Diese Variante bietet noch weitere Optionen, wie wegwerfbare nur lesbare Replikationen der Daten. Ebenso bietet sie ein eingebautes load-balancing. Dadurch wird die Verteilte Natur des Clusterings abstrahiert und für den Entwickler verborgen. Der Entwickler bekommt hierbei von der Verteilung nichts mit und muss nur so wie immer seine Eingaben tätigen.

Anzumerken ist, dass Clustering für dieses Seminar nicht in der Tiefe behandelt wurde (auf Grund des zu breiten Themen-Gebietes und der Tatsache, dass der Kern von Neo4j, das Property-Graph-Model darstellt und nicht das Clustering. Dieses ist noch einmal eine ganz eigene Welt.). Die obigen Informationen sind aus dem empfohlenen Lehrbuch der Seminars gewonnen und wurden nicht auf Aktualität geprüft.

4.1.2 Neo4j Browser

Neo4j kommt mit dem eigenen Neo4j Browser. Dieser bietet eine sehr angenehme Möglichkeit mit Hilfe von Cypher auf den vorhandenen Datenbanken zu arbeiten. Results werden sogar direkt grafisch dargestellt.

Obendrein bietet der Neo4j-Browser ein integriertes Tutorial für den Neueinsteiger und sonstige Verwaltungsmöglichkeiten.

4.1.3 Cache Sharding

Die Neo4j-Engine hat ein eingebautes Cache-Sharding, wie zuvor einmal erwähnt. Dieses ermöglicht die vollautomatisierte Zugriffsoptimierung, durch das Halten von häufig genutzten Daten im Cache

.

4.1.4 Leichter Einstieg

Neo4j bietet eine sehr große Auswahl an Tutorials und Dokumentation für den Einsteiger und dem Fortgeschrittenen. Dies geht soweit, dass ein Beginner-Tutorial bereits im Neo4j-Browser mit geliefert wird.

Neo4j bemüht sich eindeutig entwicklerfreundlich zu sein.

4.2 Use-Cases

Neo4j ist eine Datenbank, die nur für sehr spezifische Anwendungsfälle anderen Datenbanken überlegen ist, für diese aber deutlich. Im allgemeinen kann man die Behauptung aufstellen, dass man Neo4j dann einsetzen sollten, wenn die Verbindung zwischen den Objekten von deutlich größerer Wichtigkeit ist, als die Objekte selbst. Nachfolgend sind einige typische Anwendungsfälle von Neo4j aufgelistet, aber diese sind natürlich nicht alle und es kommen auch immer noch welche dazu.

4.2.1 Social Networks

Soziale Netzwerke waren mitunter die ersten Anwendungen, die auf Neo4j gesetzt haben, um ihre Angebote ordentlich umsetzen zu können. In sozialen Netzwerken haben Individuen Freunde, die Events besuchen, deren geographische Daten erfasst werden, die Posts erstellen und Nachrichten senden. All dies kann über Neo4j überwacht werden. Daraus entstehen sehr schnell, sehr große und komplizierte Verknüpfungen, wenn zum Beispiel an einem Event 1000 Leute teilgenommen haben, von welchen jeder durchschnittlich 500 Freunde besitzt. Die Interaktion und Verteilung von Nachrichten über diese wird bei herkömmlichen Datenbanken sehr schnell sehr langsam, während Neo4j dies mühelos bewältigt.

4.2.2 Matchmaking

Den sozialen Netzwerken sehr ähnlich ist das Matchmaking. Bei diesen werden die Relationships von Individuen zu bestimmten Dingen untersucht und den Relationships anderer Individuen verglichen. Auch hierbei kommt schnell eine Suche auf Tiefe n Zustände auf n Teilnehmer, wobei n sehr große Zahlen annehmen kann. Auch hier sind herkömmliche Datenbanken sehr schnell ausgelastet, bis sogar überlastet, während Neo4j durchgehen stabil skaliert.

4.2.3 Network Management

Die Nutzung von Neo4j im Network-Management ist noch ein relativ neuer Anwendungsfall. Dennoch gibt es auch hier bereits Ansätze, in welcher Neo4j durchaus attraktiv wird. Grundsätzlich besteht in einem komplexeren Firmennetzwerk eine Struktur, die über ein Graph dargestellt werden kann. Hierbei sind Netzwerkgeräte zum Beispiel Nodes und die Verbindungen untereinander die Relationships. In einem Ausfall eines Gerätes, kann über Neo4j sehr schnell ermittelt werden, welche abhängigen Geräte davon betroffen sind und welche Geräte wiederum von diesem abhängigen Geräten betroffen sind, usw.. Dies macht erst ab einer bestimmten Größe Sinn, aber da Neo4j leicht mit der Zeit wachsen kann, da es schemalos ist, kann man es durchaus bereits bei kleinen Netzwerken einsetzen, um für die Zukunft gewappnet zu sein.

4.2.4 Recommendation Engines

Als letztes wird hier noch der Einsatz von Neo4j in Recommendation Engines vorgestellt. Auch hier ist das Matchen von Relationships von Nutzern zu Objekten, zum Beispiel Produkten von größerer Bedeutung, als die Nutzer und Objekte an sich. Über das prüfen von gleichen Relationships von Nutzern, kann man anschließend nicht überschneidende Objekte zwischen diesen, den jeweils

anderen empfehlen, wenn genügend weitere Kriterien getroffen sind. Ähnlich wie Amazons „Andere kauften auch“-Empfehlungen.

4.3 Häufige Fehltritte

Nachfolgend werden häufig gemachte Fehler aufgelistet, die Neulingen in der Arbeit mit Neo4j unterlaufen.

4.3.1 Relationale Model-Technik

Häufig kommen Datenbank-Ingenieure aus der SQL-Welt und haben dadurch eine relational geprägte Design-Philosophie. Hier besteht die große Gefahr, dass zu viel Wert auf die gespeicherten Daten gelegt wird, wie zum Beispiel viel zu viele Properties und zu wenige Relationships.

Stattdessen benötigt Neo4j den Focus auf die Art, wie die Daten durchlaufen werden sollen und die Optimierung in diese Richtung. Das bedeutet korrekte und häufige Relationships, anstatt Node-Properties.

4.3.2 First-Time Nutzung an etwas kritischem

Neo4j nutzt ein (noch) nicht sehr weit verbreitetes Datenbankmodell. Häufig wird SQL und somit relationale oder alterantive datenfokusierte NoSQL Datenbanken benutzt. Die Graphen-Datenbank ist an sich noch nicht weit verbreitet und recht speziell in der Denkweise und der Anwendungsbereiche.

Dadurch sollte Noe4j nicht direkt von unerfahrenen Entwickler an etwas System-kritischen ausprobiert werden. Sie sollte lieber in einem kleinen Projekt ausprobiert und nach und nach großgezogen werden, was dank des schemalosen Designs kein Problem darstellt.

4.3.3 Speichern von Entitäten und Relationen innerhalb einer Entität

Dieses Problem schließt ein wenig an Punkt 4.3.1 Relationale Model-Technik an. Viele Entwickler, die nicht mit Property-Graph-Datenbanken vertraut sind, überladen die einzelnen Nodes mit viel zu vielen Properties, da sie das aus Spalten in Tabellen gewohnt sind. Hierbei ist zu beachten, dass man, wenn möglich, lieber aus einem Property ein Relationship zu einem Node bauen sollte, wenn

dies Sinn ergibt. Das heißt nicht, dass Properties komplett verboten sind, oder dass alle Properties immer in Nodes und Relationships umgebaut werden müssen. Es muss hierbei eine rationale und ordentliche Abschätzung vorgenommen werden, wobei Nodes und Relationships bevorzugt behandelt werden sollten.

4.3.4 Falsche Nutzung von Relationship-Types

Viele neue Nutzer von Graphen-Datenbanken arbeiten mit ungünstigen Relationship-Types, indem sie zum Beispiel alle Nodes einfach nur mit Relationships vom Typ „IS_CONNECTED_TO“ verknüpfen. Relationship Types sollten differenziert und aussagekräftig sein, aber gleichzeitig gut gruppieren. Hier gilt es wieder, die Wage zu halten.

4.3.5 Speichern großer binärer Objekte

Es wird abgeraten große binäre Objekte (BLOB) in Neo4j Properties zu speichern. Dies liegt daran, dass die Property-Datei dadurch unnötig aufgeblasen und entsprechend langsamer zu durchlaufen wird. Für Anwendungsfälle mit großen binären Objekten, sollte lieber nach dafür ausgerichteten Alternativen gesucht werden, oder Workarounds gebaut werden.

7. Bildquellen

1	https://neo4j.com/developer/guide-data-modeling/
2	https://neo4j.com/developer/guide-data-modeling/
3	https://neo4j.com/developer/guide-data-modeling/
4	https://neo4j.com/developer/guide-data-modeling/
5	https://neo4j.com/developer/guide-data-modeling/
6	Eigener Screenshot
7	https://neo4j.com/docs/cypher-manual/current/syntax/
8	https://neo4j.com/docs/cypher-manual/current/syntax/
9	https://neo4j.com/docs/cypher-manual/current/clauses/create/
10	https://neo4j.com/docs/cypher-manual/current/clauses/create/
11	https://neo4j.com/docs/cypher-manual/current/clauses/delete/
12	https://neo4j.com/docs/cypher-manual/current/clauses/match/

6. Anhang

Im Anhang befinden sich die folgenden Dateien, direkt an diesem Dokument und als ZIP-Archiv bei Abgabe!

- **Installation: Neo4j (Windows)**
- **Installation: Python3 (Windows)**