Masterseminar: Neo4j

- Wintersemester 2020/2021

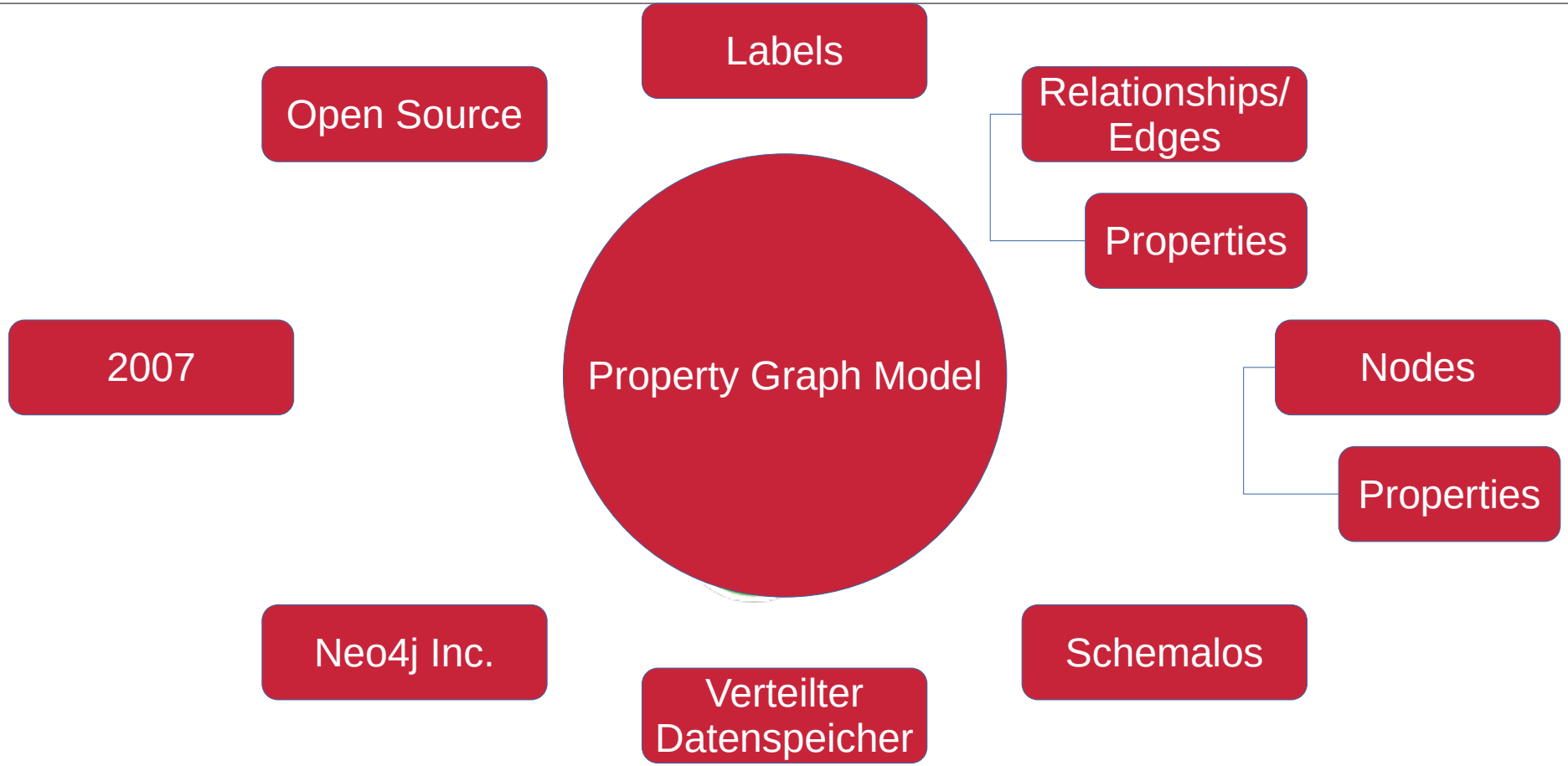
Leander Féret
Matrikelnummer 631926

Inhalt

1. Was ist Neo4j?
   - Nodes
   - Relations/Edges
   - Properties
2. Architektur
   - Storages
   - Storage Format
3. Sprachen
   - Cypher
   - Syntax
   - Beispiele
   - Unterstützte Sprachen
4. Wann Neo4j?
   - Weitere Features Neo4j
   - Use-Cases
   - Pitfalls
5. Demo
   - Systemvoraussetzungen
   - Aussicht

Bildquelle: https://www.bmc.com/blogs/neo4j-graph-database/

1. Was ist Neo4j?

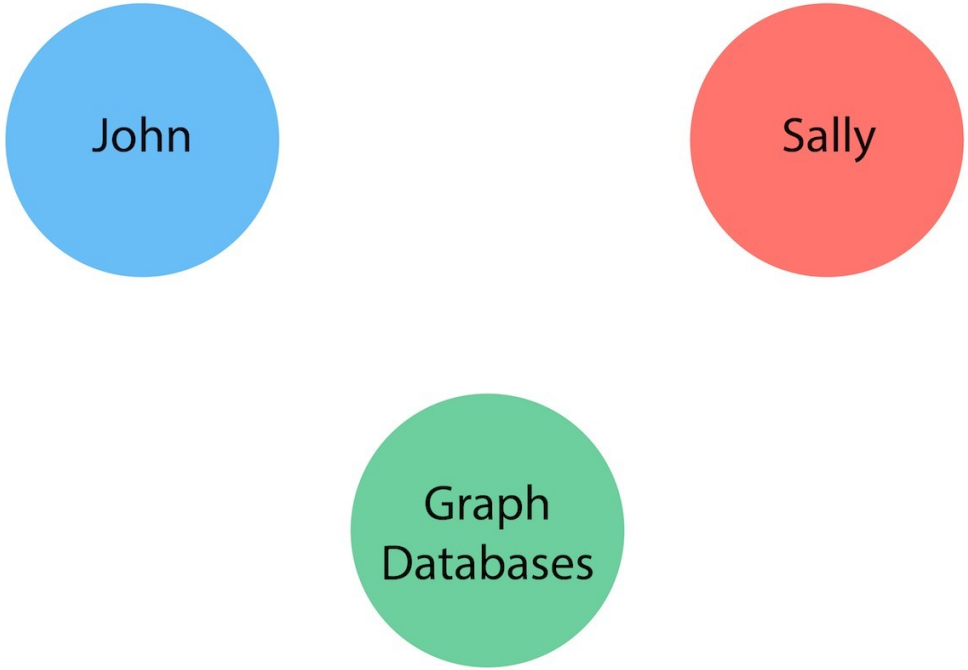


1. Was ist Neo4j?

Bildquelle: https://neo4j.com/company/

1.1 Nodes

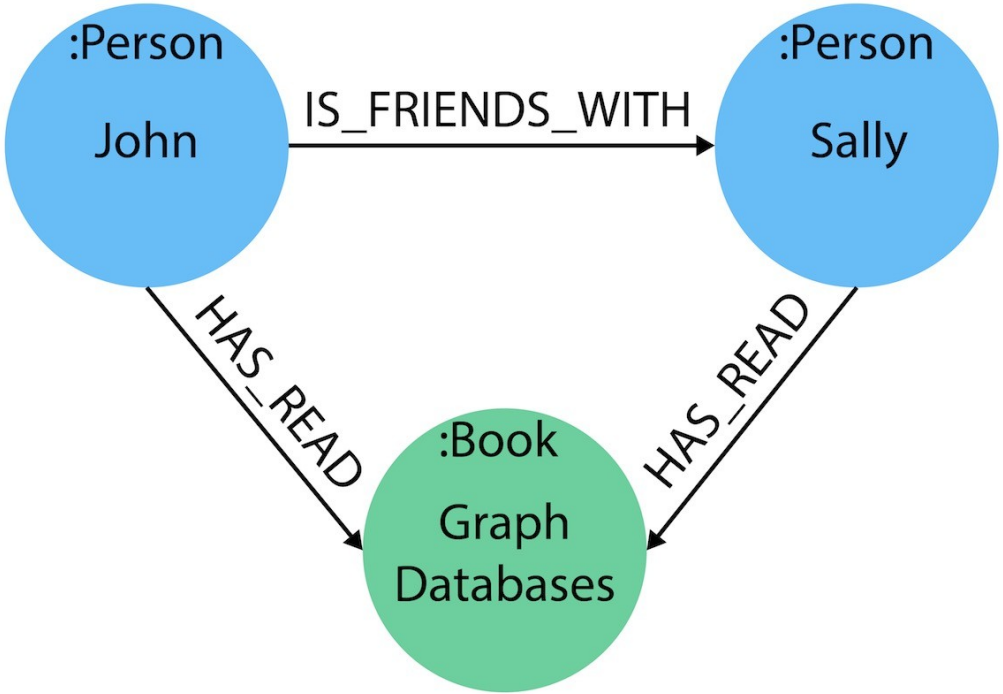
- **Entitäten**
  - wie Objekte
  - Wie Reihen
- **Label**
  - Gruppierung
  - Wie Klassen
  - Wie Tabellen
  - Können Metadaten erhalten
    - Index
    - Constraints
- **Properties**
  - Zusatzinformationen
  - Key-Value-Pairs
  - Wie Spalten
- **Relationships**
- **Node-Store**



Bildquelle: https://neo4j.com/developer/guide-data-modeling/ && https://neo4j.com/blog/data-profiling-holistic-view-neo4j/

1.2 Relationships/Edges

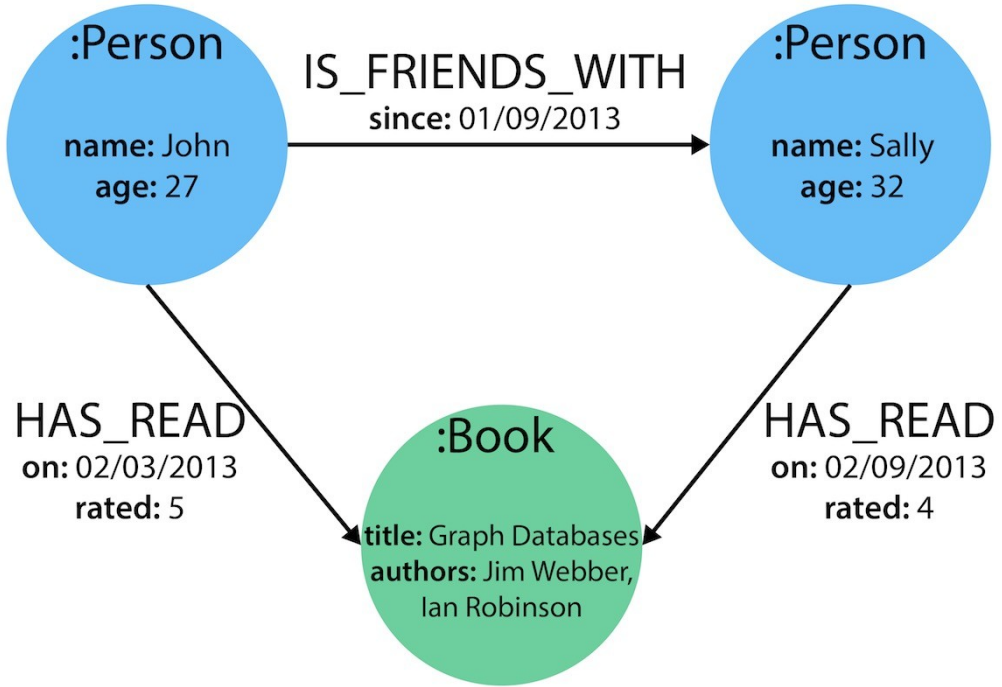
- **Verbindung zwischen zwei Nodes**
  - Haben eine Richtung
  - Haben einen Namen
  - Haben einen Startnode
  - Haben einen Endnode
  - Können in beide Richtungen navigiert werden
  - Können in beliebiger Anzahl und in unterschiedlichen Typen die selben Nodes verbinden
- **Properties**
  - Zusatzinformationen
    - meist quantitativ, wie: „Kosten“, „Zeitintervalle“ etc.
- **Relationship-Store**



Bildquelle: https://neo4j.com/developer/guide-data-modeling/ && https://neo4j.com/blog/data-profiling-holistic-view-neo4j/

1.3 Properties

- **Name-Value-Pairs**
  - Können mit Nodes verknüpft werden
  - Können mit Relationships verknüpft werden
  - Aufgabe wie Spalten einer Tabelle
- **Poperty-Types**
  - Number
    - Integer
    - Float
  - String
  - Boolean
  - Point (Spatial)
  - Temporal Types:
    - Date
    - Time
    - LocalTime
    - DateTime
    - LocalDateTime
    - Duration
- **Property-Store**



Bildquelle: https://neo4j.com/developer/guide-data-modeling/ && https://neo4j.com/blog/data-profiling-holistic-view-neo4j/

2. Architektur

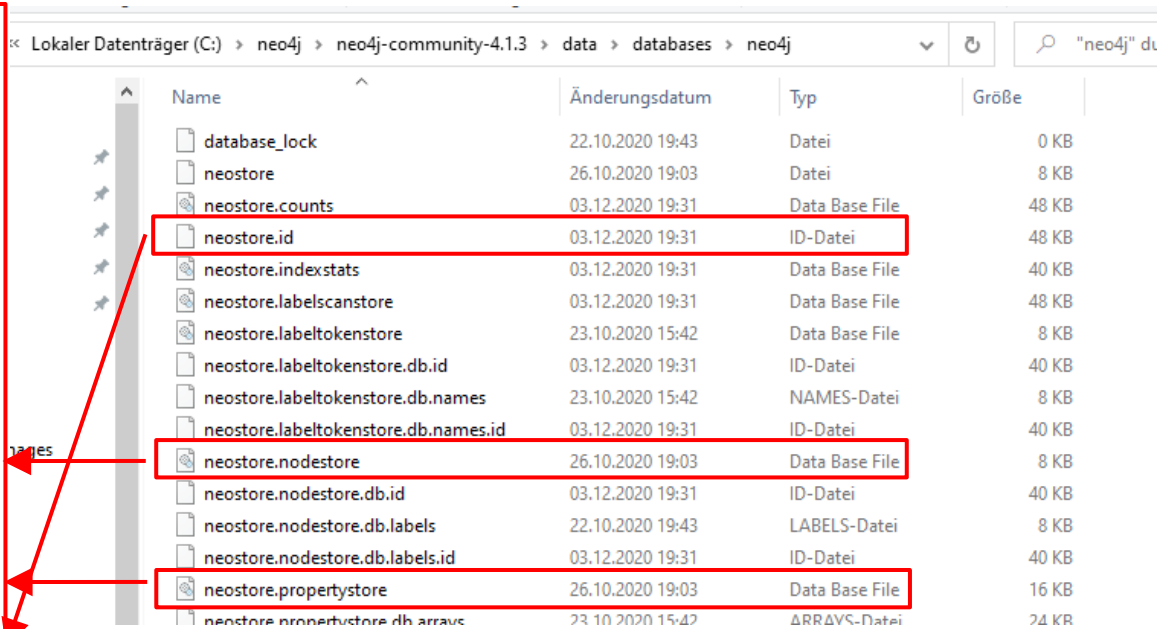
- Storages
- Storage Format

Bildquelle: https://www.bmc.com/blogs/neo4j-graph-database/

2.1 Storages

Neo4j-Installationsverzeichnis: data\\databases\\

- **Data-Base-Files**
  - nodestore*
    - Node zugehörige Daten
  - propertystore*
    - Property zugehörige Daten
    - (Key-Value properties)
  - relationshipstore*
    - Releationship zugehörige Daten
  - label*
    - Label zugehörige Daten

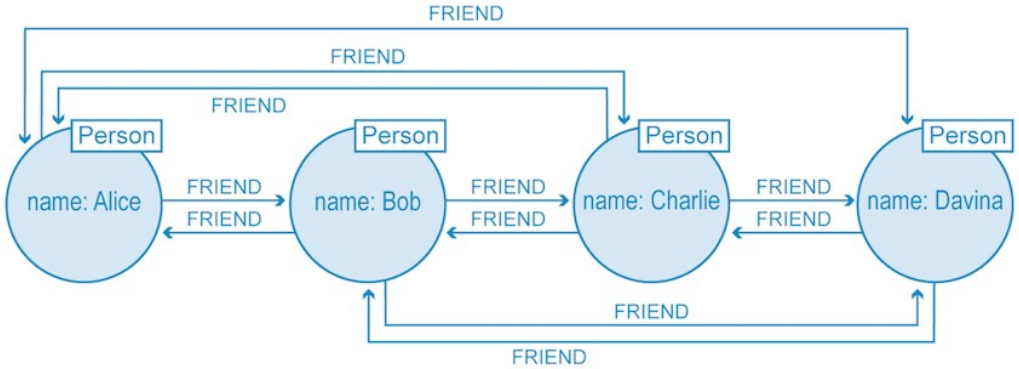


- **ID-Files**
  - Speichert ID gelöschter Objekte
  - Neue Objekte entnehmen ID aus ID-Files

2.1 Storages

Zusammenfassung

- Daten werden in Dateien gespeichert, die jeweils für den jeweiligen Graphenbereich spezifisch sind (nodestore etc.)
- Jeder Node-Eintrag zeigt auf seine Label, Relationships, Properties, speichert diese aber nicht und ist somit sehr klein
- Index-Free Adjacency: Jeder Node referenziert direkt seine verbundenen Nodes über die Pointer der Relationships
  - Jeder Node ist eine Art Micro-Index für alle verknüpften Nodes
- Query-Zeit ist proportional zur Anzahl der Nodes zu suchen und nicht der totalen Anzahl an Daten und eignet sich dadurch für große Datenmengen
- Die folgende Query (Wer ist mit wem verbunden?) lässt sich über ein Index-Einstiegspunkt auslesen!



Bildquelle: https://neo4j.com/blog/native-vs-non-native-graph-technology/

2.2 Storage Format

Node-Eintrag (15 bytes)

1 byte	isInUse
4 bytes	Node ID
1 byte	Erste Relationship ID
1 byte	Erste Property ID
5 bytes	Label Store
3 bytes	Für Zukunft reserviert

Wird der Eintrag genutzt oder ist er gelöscht?

- Wenn er als gelöscht markiert ist, wird er für neue Einträge genutzt

ID des Nodes

ID der ersten Relationship des Nodes

ID der ersten Property des Nodes

ID des Label-Stores des Nodes

- Manche der Labels sind direkt im Node gespeichert, für weniger Jumps

Quelle: https://medium.com/@gauravsarma1992/neo4j-storage-internals-be8d150028db

2.2 Storage Format

Relationship-Eintrag (34 bytes)

Start-Node ID	End-Node ID	Pointer zum Relationship-Type	Pointer zur nächster und voriger Relationship sowohl für den Start-, als auch End-Node (4 Pointer)

Der Property- und Labelstores sind simple Speicher, ähnlich dem Node-Store.

LRU k-Page Cache:

Fasst den Cache in Segmente zusammen, abhängig der unterschiedlichen Typen der Store-Dateien. Eine fixe Anzahl der Einträge werden im Cache gehalten, wobei immer die am längsten nicht mehr genutzten Einträge entfernt werden.

3. Sprache

- Cypher
- Syntax
- Beispiele

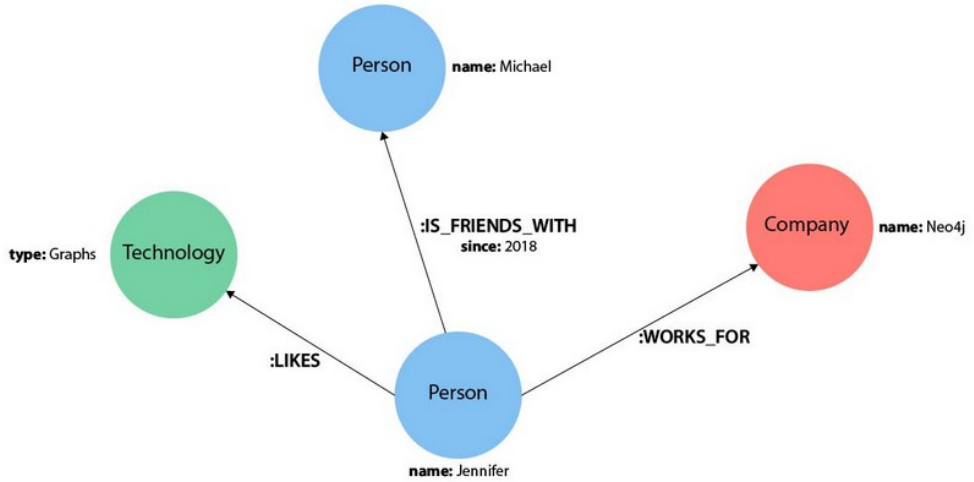
Bildquelle: https://www.bmc.com/blogs/neo4j-graph-database/

3.1 Cypher

Cypher ist eine deklarative Graphen- Abfrage-Sprache

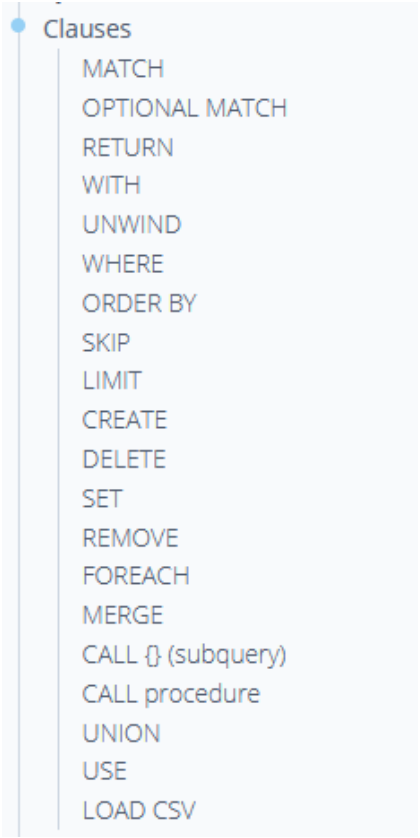
- Orientiert sich an SQL-Syntax
  - aber auch in Aspekten an Python und Haskell
- Wird genutzt zum
  - Erstellen
  - Bearbeiten
  - Löschen
  - Abfragen
- Menschen Lesbar
  - Basiert auf Englisch
- Setzt sich aus mehreren Clauses zusammen
  - Clause: CREATE, DELETE, MATCH, RETURN ...

```
(p:Person {name: "Jennifer"})-[rel:LIKES]->(g:Technology {type: "Graphs"})
```



Jennifer likes Graphs. Jennifer is friends with Michael. Jennifer works for Neo4j.

3.2 Syntax



Ein Cypher-Befehl baut sich zusammen aus n beliebigen Clauses

Nur die nachfolgende Auswahl an Clauses werden weiter betrachtet!

- CREATE
- DELETE
- MATCH

Bildquelle: https://neo4j.com/docs/cypher-manual/current/syntax/

3.2 Syntax

Auswahl wichtiger Syntax!

`(a)`	Einzelne Nodes
`(a)-->(b)`	Verknüpfte Nodes
`(a)-->(b)<--(c)`	Serie verknüpfter Nodes (Path)
`(a:User)-->(b)`	Mit Label
`(a:User:Admin)-->(b)`	Mit mehreren Label
`(a {name: 'Andy', sport: 'Brazilian Ju-Jitsu'})`	Node mit Properties
`(a)-[{blocked: false}]->(b)`	Relationship mit Property
`(a)-[r]->(b)`	Relationship mit Name (r)
`(a)-[r:TYPE1|TYPE2]->(b)`	Relationship mit Typ

3.3 Beispiele

```
CREATE (n)
```
→ Erzeugt einzelnen Node

```
CREATE (n),(m)
```
→ Erzeugt zwei Nodes

```
CREATE (n:Person)
```
→ Erzeugt Node mit Label „Person“

```
CREATE (n:Person:Swedish)
```
→ Erzeugt Node mit Label „Person“ und „Swedish“

```
CREATE (n:Person { name: 'Andy', title: 'Developer' })
```
→ Erzeugt Node mit Label und zwei Properties

```
CREATE (a { name: 'Andy' })
RETURN a.name
```
→ Erzeugt Node und gibt Property „name“ zurück

3.3 Beispiele

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'A' AND b.name = 'B'
CREATE (a)-[r:RELTYPE]->(b)
RETURN type(r)
```

→ Erzeugt Relationship zwischen Nodes

```
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
```

→ Erzeugt Relationship mit Property

CREATE erzeugt alle fehlende Bestandteile eines Patterns!

```
CREATE p =(andy { name:'Andy' })-[:WORKS_AT]->(neo)<-[:WORKS_AT]-(michael { name: 'Michael' })
RETURN p
```

→

p
(2)-[WORKS_AT,0]->(3)<-[WORKS_AT,1]-(4)

1 row, Nodes created: 3
Relationships created: 2
Properties set: 2

Quelle: https://neo4j.com/docs/cypher-manual/current/clauses/create/

3.3 Beispiele

```
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

Quelle: https://neo4j.com/docs/cypher-manual/current/clauses/delete/

3.3 Beispiele

```
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“

3.4 Unterstützte Sprachen

✓	Offizielle Treiber
	Community Treiber

✓ Java	✓ Spring	✓ Neo4j-OGM	✓ .NET	✓ JavaScript
✓ Python	✓ Go	Ruby	PHP	Erlang/Elixir
Perl	C/C++	Clojure	Haskell	R

In der Demo verwendet!

Bildquelle: https://neo4j.com/developer/language-guides/

4. Wann Neo4j?

- Weitere Features Neo4j
- Use-Cases
- Pitfalls

Bildquelle: https://www.bmc.com/blogs/neo4j-graph-database/

4.1 Weitere Features Neo4j

Clustering

- High-Avaiability Clustering
  - Mehrere Slaves verarbeiten Daten eines Master
  - Wenn Master ausfällt, wird Ersatzmaster gewählt
- Causal Clustering

Neo4j Browser

- Web-Based Applikation, zur Arbeit mit Neo4j
- Bietet Lernangebote für den schnellen Einstieg
- Überwacht die Instanz, auf welcher er läuft

Cache Sharding

- Abfragen werden auf Bereiche im vorgeladenen Cache ausgeführt

Leichter Einstieg

- Eine sehr große Menge an Online-Hilfe und Tutorials
- Ein im Neo4j-Browser eingebautes Mini-Tutorial

4.2 Use-Cases

Social Networks

- Speichern von Relationships zwischen Nodes und Informationsaustausch zwischen diesen auf Tiefe n.

Matchmaking

- Testen und Prüfen auf gleiche Relationships auf Nodes zwischen n Nodes.
- Verbieten einzelner Matches und geblockter Matches

Network Managment

- Komplexe verknüpfte Netzwerke in einem Unternehmen
  - Zum Beispiel: Bei Ausfall eines Gerätes, kann über den Graph sehr schnell abhängige Geräte und Instanzen gefunden werden.

Recommendation Engines

- Vergleiche der selben Relationships von Personen und bei genügend Übereinstimmungen, Vorschlagen der Relationships der anderen Person (Amazon „andere Kunden kauften auch“ Sektion)

4.3 Pitfalls

Relationale Model-Technik

- Nicht überlegen, wie die Daten gespeichert werden, sondern wie sie abgefragt werden!

First-Time Nutzung an etwas kritischem

- Starte mit etwas kleinem, bis die Grundlagen sitzen und weite dies dann aus!

Speichern von Entitäten und Relationships innerhalb einer Entität

- Speichere zum Beispiel das Auto eines Fahrers nicht in dem Node des Fahrers als Property,
  da zukünftige Abfragen darunter leiden können.

Falsche Nutzung von Relationship-Types

- Achte auf korrekte Relationship-Types und nicht zum Beispiel immer nur „Connected_To“.

Speichern großer binärer Objekte

- Speichern von BLOB Daten führt zu großen Propterty-Werten, die in einer einzelnen Datei gespeichert werden.
  Dadurch wird diese Datei langsamer lesbar und die gesamte Abfragen verlängern sich.

Alles mit Index versehen

- Man sollte vorsichtig sein beim Indexing von jedem Property eines Nodes oder Labels. Dies führt schnell
  zu stärkerer Auslastung auf der Festplatte auf Grund weiteren Schreibeaktivitäten.

5. Demo

- Systemvoraussetzungen
- Aussicht

Bildquelle: https://www.bmc.com/blogs/neo4j-graph-database/

5.1 Systemvoraussetzungen

RAM

- Minimal: ~2 GB
- Produktion: ~32 GB

CPU

- Minimal: Intel Core i3 oder analoge Alternativen
- Produktion: Intel Core i7 oder analoge Alternativen

Disk

- Nodes: 15 bytes
- Edges: 33 bytes
- Properties: 41 bytes
- 400.000 Nodes * 15 bytes = 6 MB + 1.200.000 Edges * 33 bytes = 39,6 MB + 3.000.000 Properties * 41 bytes = 123 MB
- Ein Größe von etwa 168,6 MB für obiges Beispiel

OS

- Entwicklung: Windows & Linux
- Produktion: Linux (Stand Neo4j Version 2.1)

Network/Firewall

- Die folgenden (TCP) Ports müssen offen sein:
  - 2003: Graphite outbound
  - 3637: JMX (not enabled by default)
  - 6362: Backups
  - 7687: Neo4j's binary Bold protocol
  - 7474: HTTP Neo4j Browser and Bolt
  - 7473: REST API
  - 5001, 6001: High-availability cluster communication
  - 5000, 6000, 7000: Causal cluster communication via the Raft protocol

5.2 Aussicht

1. Installation
2. Start
3. Basic Tutorial
4. Python3 und Neo4j
5. Simple Fraud-Detection example