Neo4J

Bei einer Graph Datenbank werden die Relationen zwischen den Daten als genau so wichtig wie die Daten selber angesehen. Die Daten selber werden ähnlich einer Dokument Datenbank ohne vorher definierte Schemas oder Modelle gespeichert.

Die Vorteile einer Graph Datenbank liegen darin, dass man zusammenhängende Daten sehr schnell bekommt und man somit von Node zu Node springen kann. Dabei ist die Zugriffszeit zwischen zwei Nodes immer konstant. Es werden keine zeitaufwendige JOIN Operationen benötigt, da die Relationen direkt mit den Daten abgespeichert werden.

Die Vorgehensweise beim holen von Daten ist, dass man bei einem Start Node startet und von dieser Node aus sich immer mehr Daten entlang des Graphen holt. Um nachher APIs zu für Graph basierte Daten zu erstellen bietet sich `GraphQL` an.

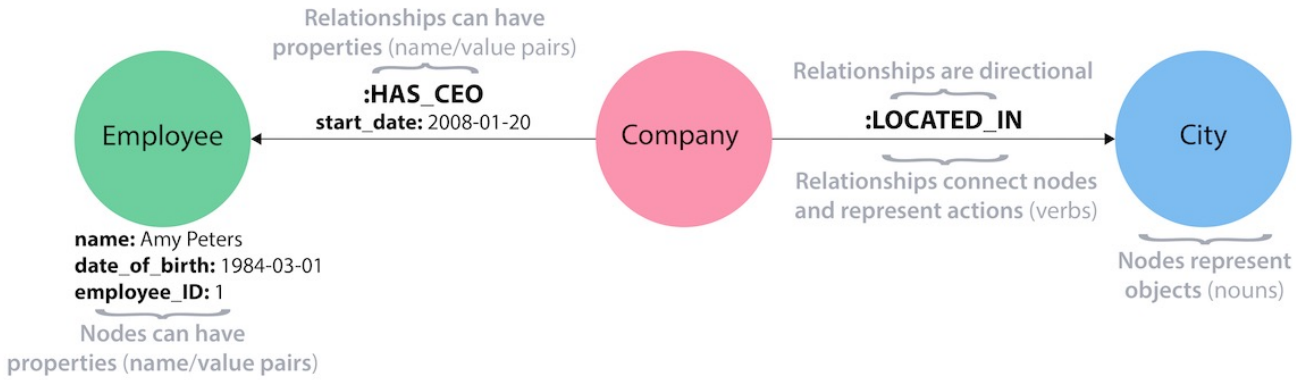
Property Graph Model

Neo4J basiert auf dem `Property Graph Model`. Das bedeutet, dass es `Nodes`, `Relationships` und `Properties` gibt. Bei `Nodes` handelt es sich um die Entitäten in einem Graph. Sie enthalten eine beliebige Anzahl an `Attributes` (key-value Paare) die sich `Properties` nennen. `Nodes` können mit `Labels` versehen werden um ihnen verschiedene Rollen innerhalb des Datensatzes zu geben. Außerdem werden `Labels` genutzt um `Nodes` mit metadata zu versehen (`Index` oder `Contrainst`).

`Relationships` sind dabei Verbindungen zwischen zwei `Nodes`. Dabei sind `Relations` immer gerichtet und mit einem Namen versehen.
Beispiel für eine Relation:

Employee `WORKS_FOR` Company

`Relationships` haben *IMMER* eine `Direction`, einen `Type` eine `Start Node` und eine `End Node`.
Auch `Relationships` können `Properties` haben, genau wie bei `Nodes`. Dies sind meistens quantitative `Properties`, wie Entfernung, Kosten, Wichtungen... Zwei gleiche `Nodes` können beliebig viele `Relationships` teilen ohne das dabei die Performance verändert wird.
Auch wenn `Relationships` immer eine `Direction` haben, kann man in beide Richtungen wandern.




```
docker run -p 7474:7474 -p 7687:7687 -d --env=NEO4J_AUTH=none neo4j
```

commands start with `:` cypher commands do not

Node -> Propperties (Key-Value) Grouping Nodes with Labels

```
CREATE (ee:Person { name: "Emil", from: "Sweden", age: 20})
```

Label: `Person` Data: `{...}`

```
MATCH (emil:Person) WHERE emil.name="Emil" RETURN emil;
```

```
MATCH (ee:Person) WHERE ee.name = "Emil"

CREATE (js:Person { name: "Johan", from: "Sweden", learn: "surfing" }),
(ir:Person { name: "Ian", from: "England", title: "author" }),
(rvb:Person { name: "Rik", from: "Belgium", pet: "Orval" }),
(ally:Person { name: "Allison", from: "California", hobby: "surfing" }),

(ee)-[:KNOWS {since: 2001}]->(js),
(ee)-[:KNOWS {rating: 5}]->(ir),
(js)-[:KNOWS]->(ir),(js)-[:KNOWS]->(rvb),
(ir)-[:KNOWS]->(js),(ir)-[:KNOWS]->(ally),
(rvb)-[:KNOWS]->(ally)
```

Find all friends of emil

```

```

```
MATCH (ee:Person)-[:KNOWS]-(friends)
WHERE ee.name = "Emil" RETURN ee, friends
```

Find someone to surf together with Johan

```
MATCH (js:Person)-[:KNOWS]-()-[:KNOWS]-(surfer)
WHERE js.name = "Johan" AND surfer.hobby = "surfing"
RETURN DISTINCT surfer
```

Updating:

First find with match, then Set

```
MATCH (p:Person {name: 'Jennifer'})
SET p.birthdate = date('1980-01-01')
RETURN p
```

Relations:

```
-[<label>:<Relation_label>]->(<result_label>:<Node_Label>)
-[<label>:<Relation_label>]-(<result_label>:<Node_Label>)
-[<label>:<Relation_label>]<->(<result_label>:<Node_Label>)
```

Movie example

Holen uns Tom Hanks

```
:play movie-graph

MATCH (tom:Person {name: 'Tom Hanks'})
RETURN tom
```

Alle seine Filme

```
MATCH (tom:Person {name: 'Tom Hanks'})-[r:ACTED_IN]->(movie:Movie)
RETURN tom, r, movie
```

Alle seine Kollegen

```

```

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-
(coActor:Person)
RETURN coActor.name
```

Alle Kollegen seiner Kollegen

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-
(coActor:Person)-[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(coCoActor:Person)
WHERE tom <> coCoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coCoActor)
RETURN coCoActor.name
```

`WHERE tom <> coCoActor` verhindert, dass Tom selber wieder gefunden wird.
Es tauchen welche doppelt auf, da es teilweise mehrere Pfade zu den Kollegen der Kollegen gibt.
Also sortieren wir mal nach der häufigkeit (Meisten gemeinsamen Kollegen) und zeigen uns die ersten 5 an.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-
(coActor:Person)-[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(coCoActor:Person)
WHERE tom <> coCoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coCoActor)
RETURN coCoActor.name, count(coCoActor) as frequency
ORDER BY frequency DESC
LIMIT 5
```

Tom Cruise ist zb sehr häufig da. Also suchen wir jetzt mal nach einem Mittelmann, der vermitteln könnte.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-
(coActor:Person)-[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(cruise:Person {name:
'Tom Cruise'})
RETURN tom, movie1, coActor, movie2, cruise
```

Indexes

Ein `Index` ist eine redundante Kopie von Daten um schneller suchen zu können. Dabei wird die Schreibgeschwindigkeit verringert und es müssen mehr Daten gespeichert werden. Also muss gut überlegt werden, auf welche `Properties` man wirklich einen `Index` braucht. Sobald ein `Index` angelegt wurde, kümmert sich Neo4J automatisch darum, diesen aktuell zu halten.

Es gibt 2 Arten von `Indexes`:

- `single-property Index`
- `composite Index`

Ein `single-property Index` ist dabei nur auf eine `Property` angelegt. Der `composite Index` hingegen auf mehr als eine `Property`.

Wenn man einen `Index` anlegt, sollte man diesem einen `index name` geben. Sonst wird er automatisch generiert. Der `index name` muss `UNIQUE` sein. Er darf nicht für einen andere `Index` oder `Constraint` genutzt werden.

Syntax:

Für einen `single-property Index`

```
CREATE INDEX [index_name]
FOR (n:LabelName)
ON (n.propertyName)
```

Oder einen `composite Index`

```
CREATE INDEX [index_name]
FOR (n:LabelName)
ON (n.propertyName_1,
    n.propertyName_2,
    …
    n.propertyName_n)
```

Index löschen:

Mit

```
DROP INDEX index_name
```

können wir einen `Index` wieder löschen.

Alle `Indexes` anzeigen lassen:

```
CALL db.indexes;
```

Constraints

Es gibt 4 `Constraints`:

- Unique node property constraints
- Node property existence constraints (Enterprise Edition only)
- Relationships property existence constraints (Enterprise Edition only)
- Node key constraints (Enterprise Edition only)

Unique node property constraint:

Eine `Property` aller `Nodes` eines `Labels` müssen `UNIQUE` sein. Das heißt zwei `Nodes` mit gleichem `Label` dürfen bei der selben `Property` nicht den selben Wert haben. `Nodes` ohne die `Property` sind von dieser Regel ausgenommen.

Syntax

```
CREATE CONSTRAINT [CONSTRAINT NAME]
ON (n:LabelName)
ASSERT n.propertyName IS UNIQUE
```

Der `Constraint Name` ist optional, sollte aber gesetzt werden. Falls dieser nicht gesetzt wird, wird er automatisch generiert. Der `Constraint` Name muss UNIQUE sein. Er darf nicht von einem bereits existierenden `Index` oder `Constraint` genutzt werden.

Um einen `Constraint` wieder zu löschen:

```
DROP CONSTRAINT constraint_name
```

Alle Constraints anzeigen lassen

Analog zu den `Indexes`

```
CALL db.constraints;
```

Hinweis zu Constraints und Indexes

Wenn wir einen Constraint anlegen, dann wird automatisch ein `single-property Index` auf die `Property` erstellt.