

Neo4J - Graphdatenbanken

Dipl.-Ing. Msc. Paul Panhofer Bsc.

① Graphentheorie Grundlagen

② CQL - Cypher Query Language Cypher Grundlagen Cypher - DDL Commands

③ APOC - Funktionen APOC - Cypher Funktions

④ Graphen Algorithmen Graphusage CALL Klausel Path finding Centrality Community

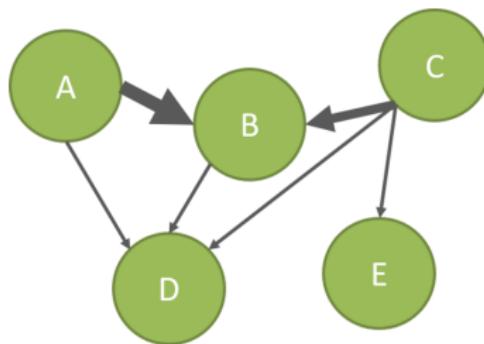
Graphentheorie

Graph

Ein **Graph** besteht aus einer Menge an Knoten V und einer Menge von Kanten E .

- Die Knoten werden mit Kanten verbunden, wobei eine Kante immer genau 2 Knoten miteinander verknüpft.
- Ein Graph wird dargestellt als $G = (V,E)$

Graphentheorie



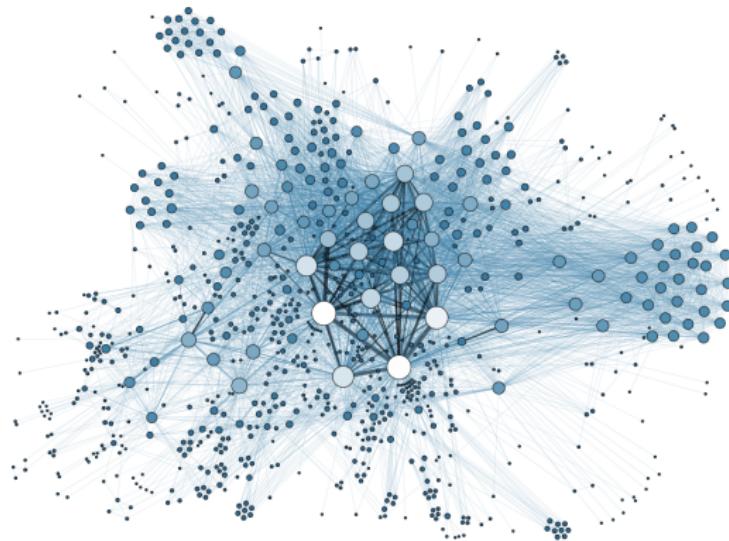
Graphentheorie

Graphen: U-Bahn Netz



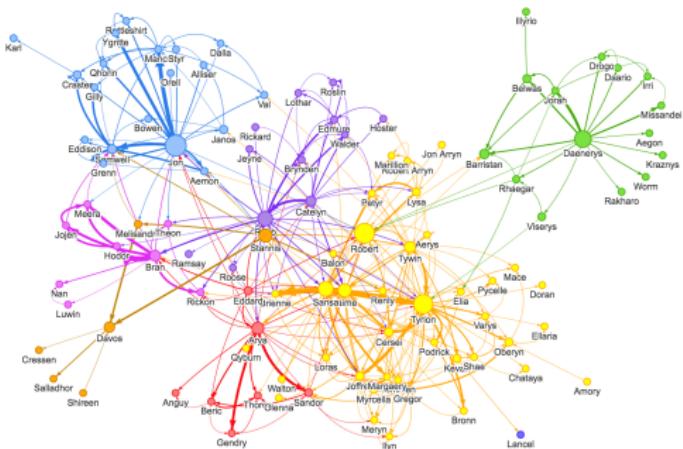
Graphentheorie

Graphen: Soziale Netzwerkanalyse



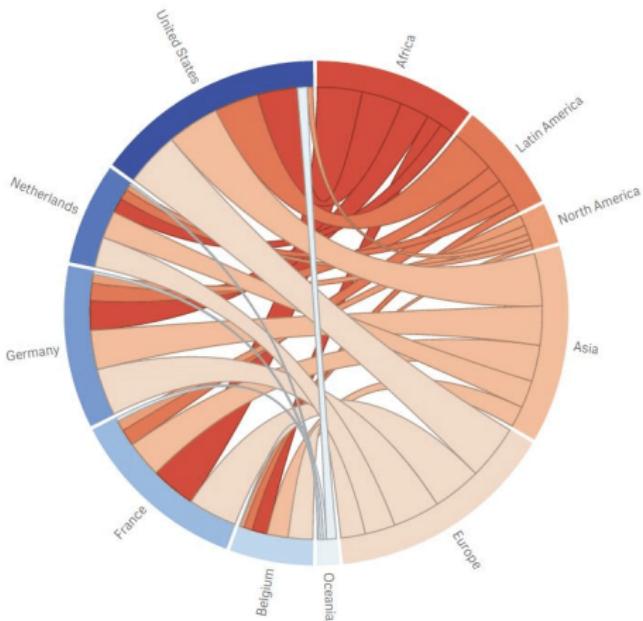
Graphentheorie

Graphen: Interaktionsanalyse



Graphentheorie

Graphen: Handelsbeziehungen



Graphentheorie

Gerichtete und ungerichtete Graphen

- In gerichteten Graphen werden Kanten durch Pfeile dargestellt.
Ist die Verbindung zweier Knoten ein Pfeil, darf die Kante nur in der Richtung des Pfeils genutzt werden.
- Wird eine Kante im Graphen als einfache Verbindung zwischen zwei Konten dargestellt, ist der Graph ungerichtet und es muss nicht auf die Richtung geachtet werden.

Graphentheorie

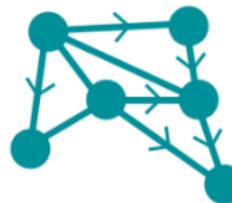
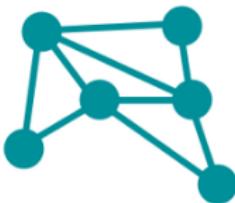
Undirected Edges



Directed Edges



indegree outdegree



Graphentheorie

```
1 -- Relationship Creating is always directed
2 MATCH (A {id : 1})
3 MATCH (B {id : 2})
4 CREATE (A)-[r:type]->(B);

5
6 -- Patternmatching can be undirected
7 MATCH (A)-[r]->(B) ...
8 MATCH (a)-[r]-(B) ...
```

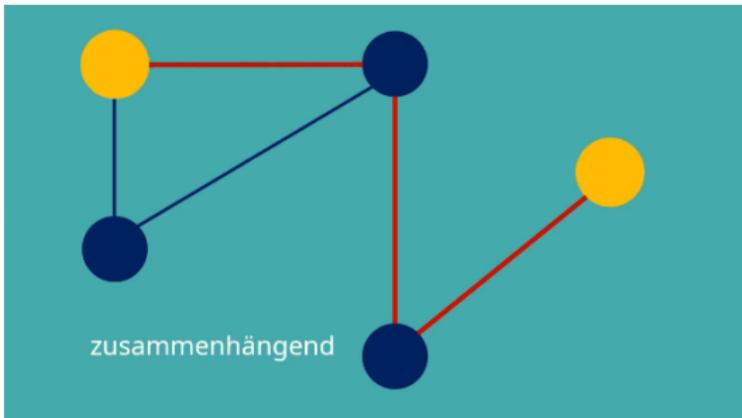
Graphentheorie

Zusammenhang von Graphen

- Ein ungerichteter Graph gilt als **zusammenhängend**, wenn es zu jedem beliebigen Knotenpaar einen Weg von einem zum anderen Knoten gibt. Jeder Knoten ist erreichbar.
- Nicht zusammenhängende Graphen erkennt man an **isolierten Knoten**.

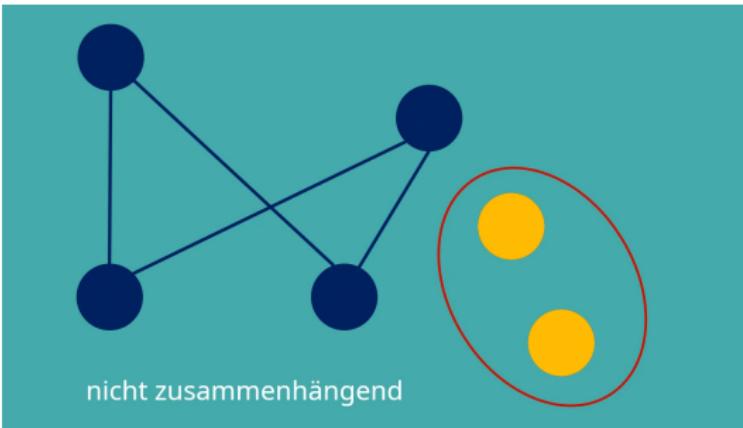
Graphentheorie

Zusammenhängende Graphen



Graphentheorie

Nicht zusammenhängende Graphen



Graphentheorie

Gewichtete Graphen

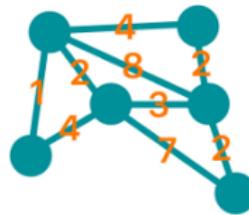
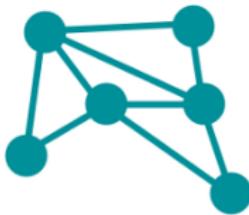
Gewichtete Graphen sind Graphen, deren Kanten ein **Kantengewicht** zugeordnet wird.

Graphentheorie

Unweighted Edge



Weighted Edge



Graphentheorie

Gewichteter vs ungewichteter Graph

```
1 -- Create Weighted Graph
2 MATCH (A {id : 1})
3 MATCH (B {id : 2})
4 CREATE (A)-[r:type {length: 5}]->(B);
```

Graphentheorie

Knotengrad

Knotengrad

Der **Knotengrad** eines Knoten beschreibt, wie viele **Kanten** in einen Knoten ein- bzw. von einem Knoten ausgehen.

Graphentheorie

Knotengrad: ungerichtete Graphen

Bei einem ungerichteten Graphen ist der Grad eines Knotens die **Anzahl der Kanten**, die mit den Knoten verbunden sind.

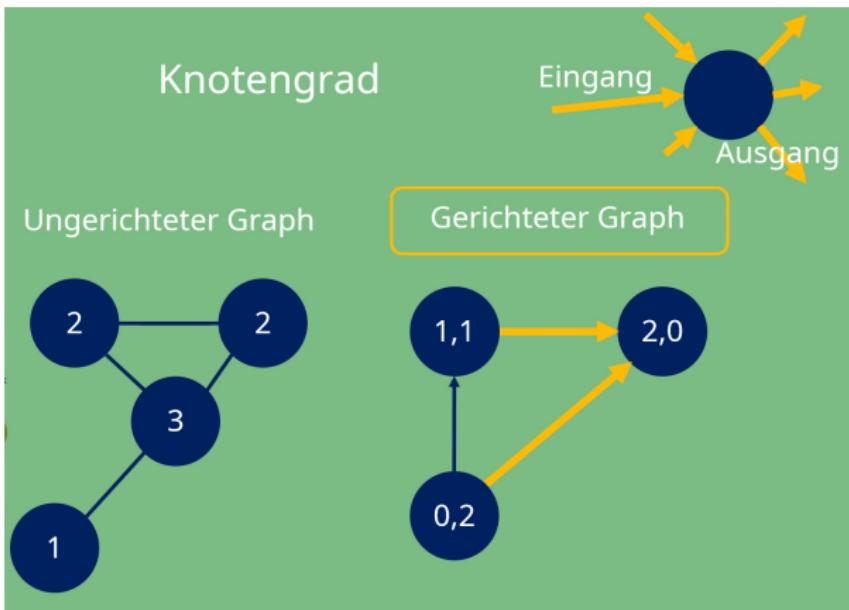
Graphentheorie

Knotengrad: gerichtete Graphen

- Bei einem gerichteten Graphen wird für jeden Knoten zwischen **Eingangsgrad** und **Ausgangsgrad** unterschieden.
- Der Eingangsgrad beschreibt wieviele **Kanten** in den Knoten hineinführen. Der Ausgangspunkt beschreibt wieviele Kanten aus einem Knoten hinausführen.

Graphentheorie

Knotengrad



Graphentheorie

Multigraphen vs. Simplegraph

- In einfache Graphen sind 2 Knoten maximal über eine Kante verbunden.
- In Multigraphen können zwischen 2 Knoten mehrere Kanten existieren.

Graphenalgorithmen

Eigenschaften von Graphen: Sparse- vs Dense Graph

Simple Graph



Multigraph

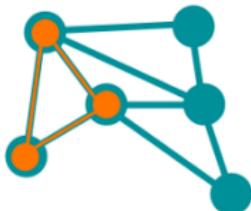


Graphenalgorithmen

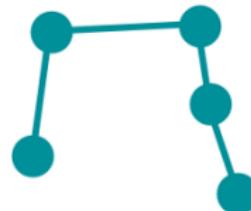
Eigenschaften von Graphen: Cyclic- vs Acyclic Graph

Hinweis: Das Ausführen mancher Algoithmen führt bei zyklischen Graphen zu keiner Termination der Routine.

Cyclic Graph



Acyclic Graph



Graphenalgorithmen

Eigenschaften von Graphen: Cyclic- vs Acyclic Graph

```
1 -- Loop Detection
2 MATCH (n)-[:TYPE*]->(n)
3 RETURN n;
4
5 -- Limited Loop Length
6 MATCH (n)-[:TYPE*..5]->(n)
7 RETURN n;
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

Centrality

Community

Cypher

Cypher ist eine **deklarative Abfragesprache**. Die Sprache ist an SQL angelehnt und für menschliche Leser gut verständlich.

Cypher

Cypher als **Abfragesprache** beruht auf 2 Prinzipien:
Klauseln und **Mustern**.

- Muster werden durch AcsiiArt beschrieben. Sie beschreiben die Struktur von Graphen.

Cypher

MATCH Klausel

Mit der **MATCH** Klausel wird definiert, welcher Teil des Graphen verarbeitet werden soll.

```
1 MATCH <AsciiArt ...>
2 ...
3 RETURN ...;
```

Cypher

MATCH Klausel - Knoten

```
1 -- Alle Knoten eines Graphen auslesen
2 MATCH (s) RETURN s;
3
4 -- Alle User Knoten eines Graphen auslesen
5 MATCH (s:Station) RETURN s;
6
7 -- Knoten mit mehreren Labeln ansprechen
8 MATCH (s:Station:Stop) RETURN s;
9
10 -- User Knoten mit bestimmten Eigenschaften
11 -- auslesen
12 MATCH (s:Station {name:"Heiligenstadt"}) RETURN s;
```

Cypher

MATCH Klausel - Knoten

```
1 -- Alle Eigenschaften eines Knotens ausgeben
2 MATCH (s:Station {name:"Heiligenstadt"})
3 RETURN properties(s);
4
5 -- Bestimmte Eigenschaften von User Knoten auslesen
6 MATCH(u:User {age:21}) RETURN u.name, u.age;
7 MATCH(u:User) RETURN u.nicknames[0];
```

Cypher

MATCH Klausel - Relationen

```
1 -- Alle Personen Knoten die eine Relation haben zu
2 -- einem Movie Knoten
3 MATCH (:PERSON)--(m:Movie)
4 RETURN m.title;

5
6 -- Definition einer gerichteten Relation
7 MATCH (:PERSON)-->(m:Movie)
8 RETURN m.title;

9
10 -- Definition einer typisierten gerichteten
11 -- Relation.
12 MATCH (:Person)-[a:ACTED_ID]->(m:Movie)
13 RETURN m.title;
```

Cypher

MATCH Klausel - Relationen

```
1 -- Definition der Richtung der Relation
2 MATCH(p1)-[:HIRED]->(p2)
3 --      (p1)<-[:HIRED]-(p2)
4 RETURN p1, p2;
5
6 -- gewichtete Relationen
7 MATCH(p)-[:HIRED {type : "fulltime"}]->(m:MOVIE)
8 return m;
9
10 -- Relationen mit mehreren Labeln
11 MATCH(p)-[:ACTED_ID|:DIRECTED]->(m:MOVIE)
12 return p.name;
```

Cypher

MATCH Klausel - Relationen

```
1 -- Mehrere Relationen definieren
2 MATCH(a:Person{name:"Charlie Sheen"})
3     - [:ACTED_IN] ->(m:Movie)<- [:DIRECTED] -
4         (d:Person)
5 RETURN d.name;
```

Cypher

MATCH Klausel - Relationen

```
1 -- Alle Knoten mit einer bestimmten Relation  
2 -- abfragen  
3 MATCH (n)-[r:Friend]->(p) RETURN n,r,p;
```

Cypher

MATCH Klausel - Abfragen

```
1 -- Geben Sie alle Properties von Tom Cruise aus.  
2 MATCH (p:Person {name:"Tom Cruise"})  
3 RETURN properties(p);  
  
4  
5 -- Finden Sie alle Filme in denen Tom Cruise als  
6 -- Schauspieler mitgewirkt hat.  
7 MATCH (p:Person {name:"Tom Cruise"})  
     -[:ACTED_IN]->  
     (m:Movie)  
10 RETURN m;
```

Cypher

Match Klausel - Abfragen

```
1 -- Finden Sie alle Filme die von derselben Person  
2 -- gedreht und produziert wurden.  
3 MATCH (p:Person)-[:PRODUCED]->(m:Movie)  
4 MATCH (p)-[:DIRECTED]->(m)  
5 RETURN p;
```

Cypher

MATCH Klausel: OPTIONAL

Konzeptionell kann der OPTIONAL MATCH mit einem LEFT JOIN in Relationalen Datenbanken verglichen werden.

Cypher

MATCH Klausel: OPTIONAL

```
1 -- Finden Sie alle Personen die in Filmen als
2 -- Schauspieler mitgewirkt haben.
3 -- Hat die Person Filme produziert geben
4 -- Sie auch die entsprechenden Filme mit aus.

5
6 MATCH (p:Person)-[:ACTED]->(:Movie)
7 OPTIONAL MATCH (p)-[:PRODUCED]->(m:Movie)
8 RETURN p,m;
```

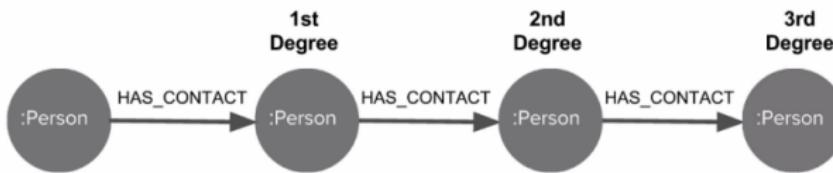
Cypher

MATCH Klausel - Pfade

Als Pfad wird eine Reihe von **Knoten** bezeichnet die über Relationen miteinander verbunden sind.

Cypher

MATCH Klausel - Pfade



Cypher

MATCH Klausel - Pfade

```
1 -- Geben Sie alle Pfade mit einer 2nd Degree Follow  
2 -- Beziehung aus.  
3 MATCH path=  
4     ((:Person)-[:FOLLOW]->(:Person)-[:FOLLOW]->(:Person))  
5 RETURN path;  
6  
7 MATCH path=((:Person)-[:FOLLOW*2]->(:Person))  
8 RETURN path;
```

Cypher

MATCH Klausel - Pfade

```
1 -- Geben Sie alle Pfade mit einer 2nd Degree FOLLOW  
2 -- Beziehung aus. Geben Sie nur die Knoten des  
3 -- Pfades an.  
4 MATCH path=((:Person)-[:FOLLOW*2]->(:Person))  
5 RETURN nodes(path);
```

Cypher

MATCH Klausel - Pfade

```
1 -- Geben Sie alle Pfade mit einer 2nd oder 3rd  
2 -- Degree FOLLOW Beziehung aus.  
3 MATCH path=((:Person)-[:FOLLOW*2..3]->(:Person))  
4 RETURN path;  
5  
6 -- Geben Sie alle Pfade mit einer 2nd oder 3rd  
7 -- Degree FOLLOW Beziehung aus. Geben Sie die  
8 -- Laenge des Pfades an.  
9 MATCH path=((:Person)-[:FOLLOW*2..3]->(:Person))  
10 RETURN length(path) AS PATH_LENGTH;
```

Cypher

MATCH Klausel - Pfade

```
1 -- Finden Sie alle Filme die von derselben  
2 -- Person produziert und gedreht wurden.  
3 -- Geben Sie die Relationen der Abfrage aus.  
4 MATCH p =  
      ((p:Person)-[:ACTED_IN] -> (m:Movie) <- [:PRODUCED] - (p))  
5 RETURN relationships(p)
```

Cypher

MATCH Klausel - Pfade

```
1 -- Geben Sie alle Pfade mit einer 2nd oder 3rd  
2 -- Degree FOLLOW Beziehung aus.  
3 MATCH  
4   (charlie:Person {name: 'Charlie Sheen'}),  
5   (martin:Person {name: 'Martin Sheen'}),  
6   p = shortestPath((charlie)-[*]-(martin))  
7 WHERE none(r IN relationships(p) WHERE type(r) =  
8   'FATHER')  
9 RETURN p
```

Cypher

WHERE Klausel

Mit der **WHERE** Klausel erfolgt eine **Filterung** der Datenbasis der Abfrage.

```
1 MATCH <ASCI Art>
2 WHERE ...
3 RETURN ...;
```

Cypher

WHERE Klausel

```
1 -- Finden Sie die Person mit dem Namen Tom Cruise
2 MATCH (p:Person)
3 WHERE p.name = "Tom Cruise"
4 RETURN p;
```

Cypher

WHERE Klausel: and, or, not

```
1 -- Finden Sie die Person mit dem Namen Tom Cruise.  
2 -- Finden Sie alle Person die mit Tom Cruise am  
3 -- Film Top Gun mitgearbeitet haben.  
4  
5 MATCH (p:Person)-[:ACTED_ID]->(m:Movie)  
6 MATCH (a:Person)-->(m)  
7 WHERE p.name = "Tom Cruise" and  
8       m.title = "Top Gun"  
9 RETURN a;
```

Cypher

WHERE Klausel: Reguläre Ausdrücke

```
1 -- Finden Sie alle Filme die in ihrem Namen den
2 -- Begriff Gun enthalten
3 MATCH (m:Movie)
4 WHERE m.title =~ ".*Gun.*"
5 RETURN m;
```

Cypher

WHERE Klausel: Label

```
1 -- Finden Sie alle Darsteller des Films Top Gun
2 MATCH (p:Person)-[r]->(m:Movie)
3 WHERE r:ACTED_IN and m.title ="Top Gun"
4 RETURN p;
```

CYPHER

RETURN Klausel

In der RETURN Klausel wird definiert welche Daten von der Abfrage zurückgegeben werden sollen.

```
1 MATCH <AsciiArt ...>
2 ...
3 RETURN ...;
```

CYPHER

RETURN Klausel - Knoten

```
1 -- Alle Knoten eines Graphen zurckgeben  
2 MATCH (n) RETURN n;  
  
3  
4 -- Die Properties eines Knoten zurckgeben  
5 MATCH (n) RETURN properties(n);  
  
6  
7 -- Properties ausgeben  
8 MATCH (n:Person)  
9 RETURN properties(n.name);
```

CYPHER

RETURN Klausel - Knoten

```
1 -- Jegliche Information ausgeben
2 MATCH (n) RETURN *;
3
4 -- Disjunkte Knoten ausgeben
5 MATCH (n) RETURN DISTINCT n;
6
7 -- Alias definieren
8 MATCH (n:Person) RETURN n as PERSON;
```

CYPHER

RETURN Klausel - Relation

```
1 -- Relation ausgeben
2 MATCH (a:Person)-[r]->(m:Movie)
3 RETURN a, r, m;
4
5 -- Typ einer Relation ausgeben
6 MATCH (a:Person)-[r]->(m:Movie)
7 RETURN a, type(r), m;
```

CYPHER

RETURN Klausel - PATH

```
1 MATCH path=( p:Person)-[:ACTED_IN]->(m:Movie) )  
2 RETURN nodes(path), relationships(path);
```

CYPHER

RETURN Klausel - reduce

```
1 -- Berechnen Sie fuer das folgende Array die Summe
2 -- der Werte.
3 RETURN reduce(i=0, age in [43,2,34,3,32] | i + age)
4 AS ageSum;
5
6 -- Berechnen Sie fuer das folgende Array das
7 -- Produkt der Werte.
8 RETURN reduce(i=1, p in [1,2,3,4,5,6] | i * p)
9 AS facult;
```

CYPHER

RETURN Klausel - collect

```
1 -- Sammeln Sie die Namen der Schauspieler von Top
2 -- Gun in einem Array.
3 MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
4 WHERE m.title = "Top Gun"
5 RETURN collect(p.name) as Actors
```

CYPHER

RETURN Klausel - Listen

```
1 -- Erhöhen Sie die Werte folgenden Arrays um 2
2 RETURN [i in [1,2,3,4,5,6] | i + 2] as data;
3
4 -- Finden Sie alle geraden Werte eines Arrays
5 RETURN [i IN [1,3,5,7,4,10] WHERE i%2 = 0]
6 AS evenValue;
```

Cypher

RETURN Klausel - Aggregationsfunktionen

```
1 -- Wieviele Darsteller spielen im Film Top Gun mit?  
2 MATCH  
3   (m:Movie {title:"Top  
4       Gun"})->[:ACTED_IN]-(a:Person)  
5 RETURN m, count(a);
```

Cypher

WITH Klausel

Die WITH Klausel ermöglicht es die Ausgabe einer Abfrage vor Ihrer Ausgabe zu verändern. Sie ermöglicht gleichzeitig die Definition neuer Ausdrücke.

- 1 MATCH ...
- 2 WHERE ...
- 3 WITH ...
- 4 RETURN ...

Cypher

WITH Klausel

```
1 MATCH(g{name:"George"})<--(p)
2 WITH p, toUpper(p.name) AS upperCaseName
3 WHERE upperCaseName STARTS WITH 'C'
4 RETURN p.name;
```

Cypher

WITH Klausel

```
1 -- Alle Werte der letzten Stufe mitnehmen  
2 MATCH (person)-[r]->(otherPerson)  
3 WITH *, type(r) AS type  
4 RETURN person.name, otherPerson.name, type
```

DQL Befehl WITH Klausel

```
1 -- Berechnen Sie fuer jede Kategorie den Wert der
2 -- gelagerten Waren.
3 MATCH (p:Product)-[:PART_OF]->(c:Category)
4 WITH c, collect(p) as products
5 RETURN c.categoryName,
6 REDUCE(price = 0,
7         product IN products |
8         price + product.unitPrice *
9         product.unitInStock);
```

Cypher

UNWIND Klausel

Mit der UNWIND Klausel können Listen auf ihre einzelnen Werte transformiert werden

```
1 UNWIND ...  
2 RETURN ...;
```

Cypher

UNWIND Klausel

```
1 -- Geben Sie die Werte folgender Liste aus:  
2 UNWIND [1, 3, 5, 6] as x  
3 RETURN x;  
4  
5 -- Create distinct list  
6 WITH [1, 1, 2, 2] AS coll  
7 UNWIND coll AS x  
8 WITH DISTINCT x  
9 RETURN collect(x) AS setOfVals
```

Cypher

UNWIND Klausel

```
1 -- Unwind Union of 2 Lists
2 WITH [1, 3, 5, 6] as a,
3      [4, 21] as b
4 UNWIND (a + b) as x
5 RETURN x;
```

Cypher

UNWIND Klausel

```
1 -- Unwind List of Lists
2 WITH [[1,2], [3,4], 5] as nested,
3 UNWIND nested as x
4 unwind x as y
5 RETURN y;
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

Centrality

Community

DDL Befehle - CREATE

```
1 -- Einen leeren Knoten anlegen
2 CREATE ();
3
4 -- Einen Knoten des Typs User anlegen und
5 -- den Knoten zurueckgeben.
6 CREATE (n :User) RETURN n;
7
8 -- Einen Knoten mit Eigenschaften anlegen
9 CREATE (s :Student {name: "Kalchert Daniel",
10 age:21} )
11 RETURN s;
```

DDL Befehle - CREATE

Constraints definieren

```
1 -- Array Properties anlegen
2 CREATE (s :Student {
3         name:"Kalchert Daniel",
4         nickname=['Kalchi', 'the beast']
5     }
6 );
7
8 -- unique Constraint fuer bestimmte Eigenschaften
9 -- von Knoten festlegen
10 CREATE CONSTRAINT ON (p:Person)
11 ASSERT p.name IS UNIQUE;
```

DDL Befehle - CREATE

```
1 -- Relationen anlegen
2 MATCH (he:Student {name:"Hannes"}) , (dk:Student
      {name:"Daniel"})
3 CREATE (he)-[f:friend]->(dk)
4 RETURN he, dk, f;
5
6 -- Relation mit Eigenschaften anlegen
7 MATCH (he:Student {name:"Hannes"}) ,
      (dk:Student {name:"Daniel"})
8 CREATE (he)-[f:friend {nickname:"Kalchi"}]->(dk)
9
10 RETURN he, dk, f;
```

DDL Befehle - CREATE

```
1 -- mehrere Relationen definieren
2 MATCH (he:Student {name:"Hannes"}) ,
3       (dk:Student {name:"Daniel"})
4 CREATE (he)-[f:friend {nickname:"Kalchi"}]->(dk)
5           -[f:friend]->(he)
6 RETURN he, dk, f;
```

DDL Befehle - SET, REMOVE

Knotendaten ändern

```
1 -- Eigenschaften zu Knoten hinzufuegen
2 MATCH (s:Student) SET s.legal=true RETURN s;
3
4 MATCH (p {name:"Hannes Ettenauer"})
5 SET p:Teacher
6 RETURN p;
7
8 -- Typen von Knoten entfernen
9 MATCH (s:Student {name:"Hannes Ettenauer"})
10 REMOVE s:Student;
```

DDL Befehle - DELETE

Artefakte löschen

```
1 -- Knoten loeschen
2 MATCH (u:User) DELETE u;
3
4 -- Relationen zwischen Knoten loeschen
5 MATCH (dk:Student {name:"Daniel"})
6      -[r:Friend]->
7      (he:Student {name:"Hannes"})
8 DELETE r;
```

DDL Befehle - MERGE

Knotendaten konditional ändern

```
1 -- Knoten anlegen. Falls der Knoten schon  
2 -- vorhanden ist soll nichts gemacht werden.  
3 MERGE (s:Student {name:"Martin", age:21})  
4 RETURN s;
```

DDL Befehle - MERGE

Knotendaten konditional ändern

```
1 -- Relationen zwischen Knoten anlegen. Falls die
2 -- Relationen bereits existieren soll nichts ge-
3 -- macht werden.
4 MATCH (m:Student {name:"Martin"}) ,
5      (j:Student {name:"Julia"})
6 MERGE (m)-[r1:Classmate]->(j)
7 MERGE (j)-[r2:Classmate]->(m)
8 RETURN m,j, type(r1);
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

Centrality

Community

APOC - Cypher Funktions

Die Cypher Spezifikation definiert eine Reihe von Funktionen zur Verarbeitung von Daten.

- **create lib:** Bibliothek die das Anlegen und Ändern von Strukturen im Graph unterstützt.
- **collections lib:** Bibliothek zur Verarbeitung von Listen von Daten. (siehe 83)
- **node lib:** Bibliothek zum Arbeiten mit Knotenmetadaten. (siehe 130)
- **aggregate lib:** Zusätzliche Aggregatfunktionen zur Verarbeitung von Listen.

APOC - Cypher Funktions

CREATE LIB - setProperty

```
1 -- Funktion zum Hinzufuegen von Properties zu
2 -- Knoten
3 MATCH (p:Person)
4 CALL apoc.create.setProperty(p, "degree", 2)
5 YIELD node
6 RETURN node;
```

APOC - Cypher Funktions

CREATE LIB - setProperties

```
1 -- Funktion zum Hinzufuegen von Properties zu
2 -- Knoten
3 MATCH (p:Person)
4 ...
5 CALL apoc.create.setProperty(
6     p, [..names..], [..values..]
7 )
8 YIELD node
9 RETURN node;
```

APOC - Cypher Functions

CREATE LIB - setProperties

```
1 -- Funktion zum Loeschen von Properties von
2 -- Knoten
3 MATCH (p:Person)
4 ...
5 CALL apoc.create.removeProperties(
6     p, ["degree"]
7 )
8 YIELD node
9 RETURN node;
```

APOC - Cypher Funktions COLLECTIONS LIB

Funktionsbibliothek mit Funktionen zur Verarbeitung von Daten in Listenform.

APOC - Cypher Funktions COLLECTIONS LIB

- `List<T> insert(List<T> data, int index, T value)` -
siehe 93
- `List<T> set(List<T> data, int index, T values)` -
siehe 94
- `List<T> insertAll(List<T> data, int index, List<T> values)` - siehe 95
- `List<T> remove(List<T> data, int index, int count)`
- siehe 96
- `List<T> removeAll(List<T> data, List<T> values)` -
siehe 97

APOC - Cypher Funktions COLLECTIONS LIB

- int `indexOf(List<T> data, T value)` - **siehe 98**
- bool `contains(List<T> data, T value)` - **siehe 99**
- bool `containsAll(List<T> data, List<T> values)` -
siehe 100

APOC - Cypher Funktions COLLECTIONS LIB

- `bool containsAll(List<T> data, List<T> values)` - siehe 100
- `bool containsAllSorted(List<T> data, List<T> values)` - siehe 101
- `bool containsDuplicates(List<T> data)` - siehe 103
- `bool different(List<T> data)` - siehe 104

APOC - Cypher Funktions COLLECTIONS LIB

- `List<T> disjunction(List<T> data, List<T> data2)` -
siehe 105
- `List<T> intersection(List<T> data, List<T> data2)` -
siehe 106
- `List<T> subtract(List<T> data, List<T> data2)` -
siehe 107
- `List<T> unionAll(List<T> data, List<T> data2)` -
siehe 108

APOC - Cypher Funktions COLLECTIONS LIB

- `List<T> toSet(List<T> data)` - **siehe 109**
- `List<T> flatten(List<T> data)` - **siehe 110**

APOC - Cypher Funktions COLLECTIONS LIB

- `List<List<T>` `partition(List<T> data, int size)` - siehe 112
- `List<List<T>` `pairs(List<T> data)` - siehe 113
- `List<List<T>` `split(List<T> data, T discriminator)` - siehe 114
- `List<List<T>` `combinations(List<T> data, int tupelSize)` - siehe 116
- `List<U,V>` `zipToRow(List<U> data, List<V>)` - siehe 118
- `List<T>` `dropDuplicateNeighbors(List<T> data)` - siehe 119

APOC - Cypher Funktions COLLECTIONS LIB

- `List<T> duplicates(List<T> data)` - **siehe 120**
- `List<U> duplicatesWithCount(List<T> data)` - **siehe 121**
- `List<U> frequencies(List<T> data)` - **siehe 122**

APOC - Cypher Funktions COLLECTIONS LIB

- `List<T> sort(List<T> data)` - **siehe 123**
- `List<T> sortNodes(List<T> data)` - **siehe 124**

APOC - Cypher Funktions COLLECTIONS LIB

- T `min(List<T> data)` - **siehe 125**
- T `max(List<T> data)` - **siehe 126**
- T `avg(List<T> data)` - **siehe 127**
- T `sum(List<T> data)` - **siehe 128**
- T `count(List<T> data)` - **siehe 129**

APOC - Cypher Funktions

COLLECTIONS LIB - insert

```
1 -- Funktion zum Einfuegen eines Elements in
2 -- eine Liste.
3 RETURN apoc.coll.insert (
4     [1,2,1,4],
5     2, -- Index
6     34 -- Wert
7 );
8
9 -- Result:
10 -- [1,2,34,1,4]
```

APOC - Cypher Funktions

COLLECTIONS LIB - set

```
1 -- Funktion zum Setzen eines Elements
2 -- in der Liste
3 RETURN apoc.coll.set (
4     [1,2,1,4],
5     2, -- Index
6     34 -- Value
7 );
8
9 -- Result:
10 -- [1,2,34,4]
```

APOC - Cypher Funktions

COLLECTIONS LIB - insertAll

```
1 -- Funktion zum Einfuegen von Elementen in
2 -- eine Liste.
3 RETURN apoc.coll.insertAll (
4     [1,2,1,4],
5     2, -- Index
6     [25,18]
7 );
8
9 -- Result:
10 -- [1,2,25,18,1,4]
```

APOC - Cypher Funktions

COLLECTIONS LIB - remove

```
1 -- Funktion zum Loeschen von Elementen ab einem
2 -- bestimmten Index.
3 RETURN apoc.coll.remove (
4     [1,4,2,2],
5     1, -- Index
6     1 -- Amount
7 );
8
9 -- Result:
10 -- [1,4,2]
```

APOC - Cypher Funktions

COLLECTIONS LIB - removeAll

```
1 -- Funktion zum Loeschen von Elementen
2 RETURN apoc.coll.removeAll (
3     [1,4,2,2] ,
4     [1,2]
5 );
6
7 -- Result:
8 -- [4]
```

APOC - Cypher Funktions

COLLECTIONS LIB - indexOf

```
1 -- Funktion zum Ermitteln des ersten Index eines
2 -- Elements der Liste
3 RETURN apoc.coll.indexOf (
4     [1, 4, 2, 4], 4
5 );
6
7 -- Result:
8 -- 1
```

APOC - Cypher Funktions

COLLECTIONS LIB - contains

```
1 -- Funktion zum Pruefen ob ein Wert in einer
2 -- Liste von Werten enthalten ist.
3 RETURN apoc.coll.contains([2,3,1,1], 3);
4
5 -- Result:
6 -- true
```

APOC - Cypher Funktions

COLLECTIONS LIB - containsAll

```
1 -- Funktion zum Pruefen ob eine Liste, Teil-
2 -- liste der gegebenen Collection ist.
3 RETURN apoc.coll.containsAll(
4     [2,3,1,1], -- Collection
5     [2,3]       -- Teilliste
6 );
7
8 -- Result:
9 -- true
```

APOC - Cypher Funktions

COLLECTIONS LIB - containsAllSorted

```
1 -- Funktion zum Pruefen ob eine Liste, Teil-
2 -- liste der gegebenen Collection ist.
3 RETURN apoc.coll.containsAllSorted(
4     [2,3,1,1], -- Collection
5     [2,3,1]      -- Teilliste
6 );
7
8 -- Result:
9 -- true
```

APOC - Cypher Funktions

COLLECTIONS LIB - containsAllSorted

```
1 -- Funktion zum Pruefen ob eine Liste, Teil-
2 -- liste der gegebenen Collection ist.
3 RETURN apoc.coll.containsAllSorted(
4     [2,3,1,1], -- Collection
5     [2,1]      -- Teilliste
6 );
7
8 -- Result:
9 -- false
```

APOC - Cypher Funktions

COLLECTIONS LIB - containsDuplicates

```
1 -- Funktion zum Pruefen ob eine Liste Duplikate
2 -- enthaelt.
3 RETURN apoc.coll.containsDuplicates(
4     [2,3,1,1]
5 );
6
7 -- Result:
8 -- true
```

APOC - Cypher Funktions

COLLECTIONS LIB - different

```
1 -- Funktion zum Pruefen ob alle Elemente einer
2 -- Liste unterschiedlich sind.
3 RETURN apoc.coll.different(
4     [2,3,1,1]
5 );
6
7 -- Result:
8 -- false
```

APOC - Cypher Funktions

COLLECTIONS LIB - disjunction

```
1 -- Funktion zum Bilden der Disjunktion 2er
2 -- Listen. Die Ergebnisliste ist dabei eine echte
3 -- Menge
4 RETURN apoc.coll.disjunktion(
5     [2,3,1,1,7],
6     [3,4]
7 );
8
9 -- Result:
10 -- [2,1,7]
```

APOC - Cypher Funktions

COLLECTIONS LIB - intersection

```
1 -- Funktion zum Berechnen der Schnittmenge 2er
2 -- Listen.
3 -- Menge
4 RETURN apoc.coll.intersection(
5     [2,3,1,1,7],
6     [3,4,1,1]
7 );
8
9 -- Result:
10 -- [3,1]
```

APOC - Cypher Funktions

COLLECTIONS LIB - subtract

```
1 -- Funktion zum Berechnen der Differenzmenge 2er
2 -- Listen.
3 RETURN apoc.coll.subtract(
4     [2,3,1,1,7],
5     [3,4,1]
6 );
7
8 -- Result:
9 -- [2,7]
```

APOC - Cypher Funktions

COLLECTIONS LIB - unionAll

```
1 -- Funktion zum Berechnen der Vereinigungsmenge
2 -- 2er Listen.
3 RETURN apoc.coll.unionAll(
4     [2,3,1,7],
5     [3,4,1]
6 );
7
8 -- Result:
9 -- [2,3,1,7,3,4,1]
```

APOC - Cypher Funktions

COLLECTIONS LIB - toSet

```
1 -- Funktion zum Transformieren der Liste
2 -- in eine Menge
3 RETURN apoc.coll.toSet (
4   [2,3,3,1,4,2]
5 );
6
7 -- Result:
8 -- [2,3,1,4]
```

APOC - Cypher Funktions

COLLECTIONS LIB - flatten

```
1 -- Funktion zum Begradiigen von Listen.  
2 RETURN apoc.coll.flatten (  
3     [2,3, [3,2,5], [1]]  
4 );  
5  
6 -- Result:  
7 -- [2,3,3,2,5,1]
```

APOC - Cypher Funktions

COLLECTIONS LIB - flatten

```
1 RETURN apoc.coll.flatten (            
2     [2,3, [3,2,5], [1, [2,3]]]        
3 );  
4  
5 -- Result:  
6 -- [2,3,3,2,5,1, [2,3]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - partition

```
1 -- Funktion zum Partitionieren von Listen in Teil-
2 -- listen. Die Groesse der Teillisten wird als
3 -- Parameter festgelegt
4 RETURN apoc.coll.partition([1,4,2,5,3,2,1], 3);
5
6 -- Result:
7 -- [[1,4,2], [5,3,2], [1]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - pairs

```
1 -- Funktion zum Partitionieren von Listen in Teil-
2 -- listen. Die Groesse der Teillisten wird als
3 -- Parameter festgelegt
4 RETURN apoc.coll.pairs([1,4,2,5,3,2,1]);
5
6 -- Result:
7 -- [[1,4],[4,2],[2,5],[5,3],[3,2],[2,1]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - split

```
1 -- Funktion zum Aufspalten von Listen in Teil-
2 -- listen.
3 CALL apoc.coll.split([1,4,2,5,3,2,1], 5)
4 YIELD value
5 RETURN value;
6
7 -- Result:
8 -- [[1,4,2], [3,2,1]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - split

```
1 CALL apoc.coll.split([1,4,2,5,3,2,1], 2)
2 YIELD value
3 RETURN value;
4
5 -- Result:
6 -- [[1,4], [5,3], [1]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - combinations

```
1 -- Funktion zum Generieren von Tupeln
2 -- aus Listen. Die Anzahl der Sublisten
3 -- entspricht dabei: 3!/2!
4 RETURN apoc.coll.combinations(
5     [1,3,5], 2
6 );
7 -- Result:
8 -- [[1,3], [1,5], [3, 5]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - combinations

```
1 RETURN apoc.coll.combinations([2,3,1,1], 2);  
2  
3 -- Result:  
4 -- [[2,3], [2,1], [2,1], [3,1], [3,1], [1,1]]
```

APOC - Cypher Funktions

COLLECTIONS LIB - zipToRow

```
1 -- Funktion zum Verknuepfen der Werte 2er
2 -- Listen
3 CALL apoc.coll.zipToRow(
4     [1,2,3],
5     ["a", "b", "c"]
6 )
7 YIELD value
8 RETURN value;
9
10 -- Result:
11 -- [1, "a"] \n [2, "b"]\n [3, "c"]
```

APOC - Cypher Funktions

COLLECTIONS LIB - dropDuplicateNeighbors

```
1 -- Funktion zum Entfernen von identischen Nachbarn
2 -- einer Liste
3 RETURN apoc.coll.dropDuplicateNeighbors(
4     [2,2,2,3,2,2,1,1,7,7,1],
5 );
6
7 -- Result:
8 -- [2,3,2,1,7,1]
```

APOC - Cypher Funktions

COLLECTIONS LIB - duplicates

```
1 -- Funktion zum Ermitteln aller Elemente die mehr-
2 -- mals in der Liste enthalten sind.
3 RETURN apoc.coll.duplicates(
4     [2,2,2,3,2,2,1,1,7,7,1],
5 );
6
7 -- Result:
8 -- [2,3,1,7]
```

APOC - Cypher Funktions

COLLECTIONS LIB - duplicatesWithCount

```
1 -- Funktion zum Ermitteln der Anzahl der
2 -- Vorkommen der einzelnen Duplikate
3 RETURN apoc.coll.duplicatesWithCount(
4     [2,2,2,3,2,2,1,1,7,7,1],
5 );
6
7 -- Result:
8 -- [{"count":5 , "item":2}, {"count":3 , "item":1},
9 -- {"count":2, "item":7}]
```

APOC - Cypher Funktions

COLLECTIONS LIB - frequencies

```
1 -- Funktion zum Ermitteln der Anzahl der
2 -- Vorkommen der einzelnen Elemente
3 RETURN apoc.coll.frequencies (
4     [2,2,2,3,2,2,1,1,7,7,1],
5 );
6
7 -- Result:
8 -- [{"count":5 , "item":2}, {"count":3 , "item":1},
9 -- {"count":2, "item":7}, {"count":1 , "item":3}]
```

APOC - Cypher Funktions

COLLECTIONS LIB - sort

```
1 -- Funktion zum Sortieren der Elemente einer Liste
2 RETURN apoc.coll.sort (
3     [56,7,2,1,78],
4 );
5
6 -- Result:
7 -- [1,2,7,56,78]
```

APOC - Cypher Funktions

COLLECTIONS LIB - sortNodes

```
1 -- Funktion zum Sortieren von Knoten nach
2 -- einer Property
3 MATCH (p:Person)
4 WITH collect(p) as people
5 RETURN CALL apoc.coll.sortNodes (people, "name");
```

APOC - Cypher Funktions

COLLECTIONS LIB - min

```
1 -- Funktion zum  
2 RETURN apoc.coll.min([1,5,32]);  
3  
4 -- Result  
5 -- 1
```

APOC - Cypher Funktions

COLLECTIONS LIB - max

```
1 -- Funktion zum  
2 RETURN apoc.coll.max([1,5,32]);  
3  
4 -- Result  
5 -- 32
```

APOC - Cypher Funktions

COLLECTIONS LIB - avg

```
1 -- Funktion zum  
2 RETURN apoc.coll.avg([7,10,4]);  
3  
4 -- Result  
5 -- 7
```

APOC - Cypher Funktions

COLLECTIONS LIB - sum

```
1 -- Funktion zum
2 RETURN apoc.coll.sum([7,10,4]);
3
4 -- Result
5 -- 21
```

APOC - Cypher Funktions

COLLECTIONS LIB - count

```
1 -- Funktion zum  
2 RETURN apoc.coll.count([1,5,32]);  
3  
4 -- Result  
5 -- 3
```

APOC - Cypher Funktions COLLECTIONS LIB

Funktionsbibliothek mit Funktionen zur Verarbeitung von
Knoten.

APOC - Cypher Funktions

COLLECTIONS LIB - degree

```
1 -- Funktion zum Berechnen des Knotengrades
2 MATCH (s:Station)
3 RETURN s, apoc.node.degree(s);
4
5 MATCH (p:Person{Name:"Migel"})
```

APOC - Cypher Funktions COLLECTIONS LIB -

```
1 -- Funktion zum Einfuegen eines Elements in
2 -- eine Liste.
3 RETURN apoc.node (
4 );
5
6 -- Result:
7 -- [1,2,34,1,4]
```

APOC - Cypher Funktions COLLECTIONS LIB -

```
1 -- Funktion zum Einfuegen eines Elements in
2 -- eine Liste.
3 RETURN apoc.node (
4 );
5
6 -- Result:
7 -- [1,2,34,1,4]
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

Centrality

Community

Graphenalgorithmen

Graphen kommen in diversen Bereichen des Alltags zum Einsatz.



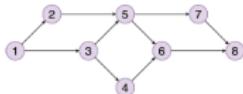
Graphenalgorithmen

Graphusage

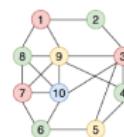
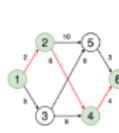
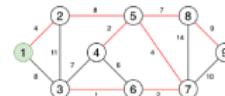
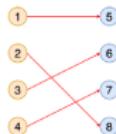
- Biology: Epidemy Progagation
- Games: State Transition Graph
- Traffic: Interstate Highway
- ...

Graphalgorithmen

Zur Verarbeitung von Graphen werden Graphalgorithmen verwendet.



Graph Algorithms



Graphalgorithmen

Klassifizierung

Graphenalgorithmen werden je nach ihrer Funktionsweise in **Kategorien** klassifiziert.

- Shortest Path: Path finding algorithms find the **shortest path** between two or more nodes or evaluate the availability and quality of paths.
- Centrality: Centrality algorithms are used to determine the **importance** of distinct nodes in a network.

Graphalgorithmen

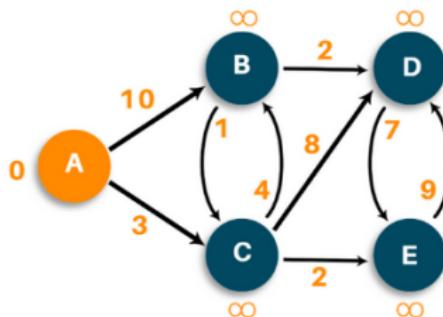
Klassifizierung

- **Communities:** Community detection algorithms are used to evaluate how groups of nodes are **clustered or partitioned**, as well as their tendency to strengthen or break apart.
- **Similarity:** Similarity algorithms compute the similarity of pairs of nodes using different vector-based metrics.

Graphalgorithmen

Shortest Path

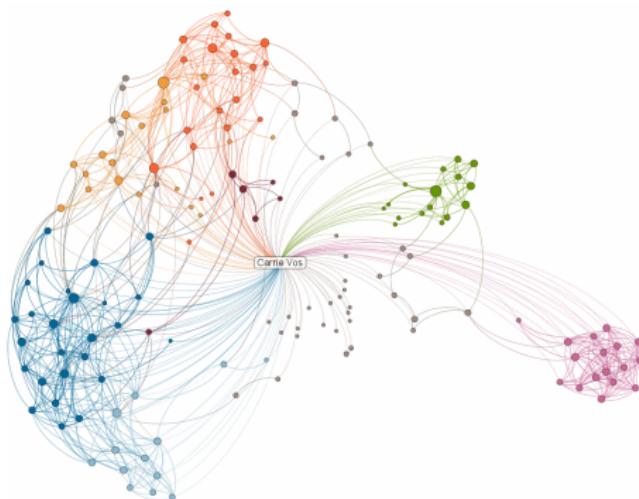
Shortest Path: Kürzesten Pfad zwischen 2 Knoten eines Graphen berechnen.



Graphalgorithmen

Centrality beschreibt die Suche nach den wichtigsten Knoten eines Graphen.

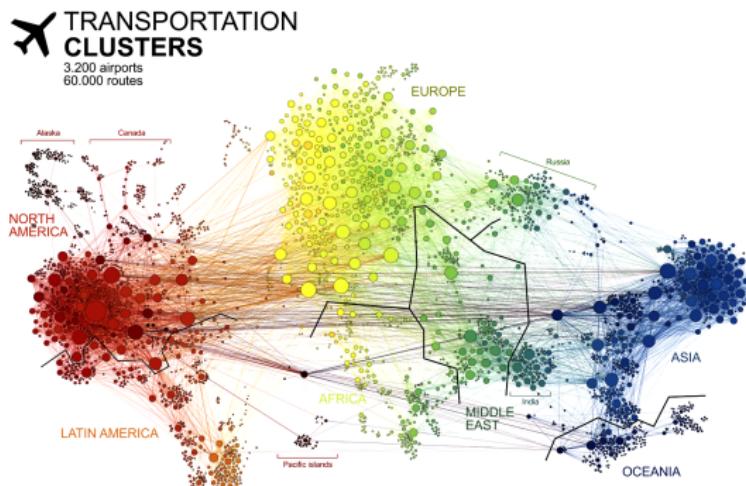
- Influencias
- Critical Nodes



Graphalgorithmen

Communities: Finden aller Knoten mit ähnlichen Eigen-schaften.

- Clusterbildung
- Recommendationengine



① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

Centrality

Community

Graphalgorithmen

CALL Klausel

Graphalgorithmen werden nicht auf dem gespeicherten Graphen, sondern für eine **Kopie** des Graphen ausgeführt.

- **1.Schritt:** Kopie der gewünschten Knoten und Kanten in den Speicher laden.
- **2.Schritt:** Algorithmus ausführen.

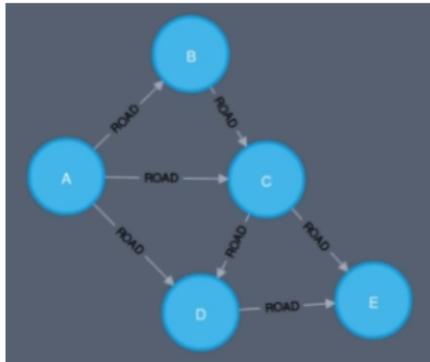
Graphalgorithmen

CALL Klausel: Beispielgraphen anlegen

```
1 -- Datenbasis
2 WITH [
3     // START, END, DISTANCE
4     ["A", "B", 10], ["A", "C", 33], ["A", "D", 35],
5     ["C", "D", 28], ["B", "C", 20], ["C", "E", 6],
6     ["D", "E", 40]
7 ] AS nested
8 UNWIND nested AS row
9 MERGE (n:City {name: row[0]})
10 MERGE (m:City {name: row[1]})
11 MERGE (n)-[r:ROAD {distance: row[2]}]->(m);
```

Graphalgorithmen

CALL Klausel: Beispiel



Graphalgorithmen

CALL Klausel: Graph in den Speicher laden

```
1 CALL gds.graph.create (
2     "mygraph",
3     "City",
4     ["ROAD", "TRAIN"],
5     {
6         relationshipProperties: "distance"
7     }
8 );
```

Graphalgorithmen

CALL Klausel: Graph in den Speicher laden

```
1 CALL gds.graph.create.cypher (
2   "mygraph",
3   "MATCH (c:City) RETURN id(c) as id",
4   "MATCH (n:City)-[r:ROAD|TRAIN]->(m:City)
5   RETURN id(n) as source,
6           id(m) as target,
7           type(r) as type,
8           coalesce(r.distance, 0) as distance"
9 );
```

Graphalgorithmen

CALL Klausel: Virtuelle Graphen löschen

```
1 CALL gds.graph.drop('my-fictive-graph', false)
2 YIELD graphName;
```

Graphalgorithmen

CALL Klausel

Die CALL Klausel wird verwendet um Algorithmen in einer Cypher Abfrage aufzurufen.

```
1 -- show installed algorithms
2 CALL algo.list();
3
4 -- show db labels
5 CALL db.labels();
```

Graphalgorithmen

YIELD Klausel

Mit der YIELD Klausel kann das Ergebnis eines Methodenaufrufs zur weiteren **Verarbeitung** bereitgestellt werden.

```
1 -- show installed algorithms
2 CALL db.labels() YIELD label
3 RETURN count(label) AS numLabels;
4
5 CALL db.propertyKeys() YIELD propertyKey AS prop
6 MATCH (n)
7 WHERE n[prop] IN NOT NULL
8 RETURN prop, count(n) AS numNodes;
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

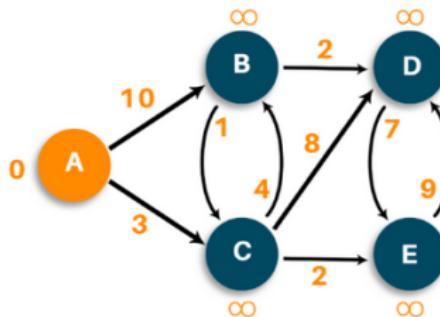
Centrality

Community

Graphalgorithmen

Path finding

Path finding algorithms find the **shortest path** between two or more nodes or evaluate the availability and quality of paths.



Graphalgorithmen

Path finding

- Dijkstra Source-Target: The Dijkstra Shortest Path algorithm computes the shortest path between nodes.
The algorithm supports weighted graphs with positive relationship weights.
- Dijkstra Single-Source: The Dijkstra Single-Source algorithm computes the shortest paths between a source node and all nodes reachable from that node.

Graphalgorithmen

Dijkstra Source Target

Dijkstra Source-Target: The Dijkstra Shortest Path algorithm computes the **shortest path** between nodes.

The algorithm supports weighted graphs with positive relationship weights.

Graphalgorithmen

Dijkstra Source Target

```
1 -- Berechne den kuerzesten Weg von A nach E.  
2 -- Beruecksichtigen Sie die Anzahl der Hops.  
3 MATCH (A:City {name: "A"}) // <-- Startknoten  
4 MATCH (E:City {name: "E"}) // <-- Endknoten  
5 CALL gds.shortestPath.dijkstra.stream(  
6   'myGraph', {  
7     sourceNode: A,  
8     targetNode: E,  
9     relationshipWeightProperty: null  
10    }  
11  )  
12 YIELD path  
13 RETURN nodes(path) as Path;
```

Graphalgorithmen

Dijkstra Source Target: Pfadausgabe

```
1 -- Berechne den kuerzesten Weg von A nach E.  
2 -- Beruecksichtigen Sie die Distanz.  
3 MATCH (A:City {name: "A"}) // <-- Startknoten  
4 MATCH (E:City {name: "E"}) // <-- Endknoten  
5 CALL gds.shortestPath.dijkstra.stream(  
6   "myGraph", {  
7     sourceNode: A,  
8     targetNode: E,  
9     relationshipWeightProperty: "distance"  
10    }  
11  )  
12 YIELD path  
13 RETURN nodes(path) as Path;
```

Graphalgorithmen

Dijkstra Source Target: Pfadausgabe

```
1 -- Berechne den kuerzesten Weg von A nach E.  
2 -- Beruecksichtigen Sie die Distanz. Geben Sie  
3 -- die Namen der Knoten gesammelt in einem  
4 -- Array aus.  
5 MATCH (A:City {name: "A"})  
6     ...  
7 YIELD nodeIds  
8 RETURN [id IN nodeIds | gds.util.asNode(id).name ];
```

Graphalgorithmen

Dijkstra Source Target: Rückgabewerte

```
1 CALL gds.shortestPath.dijkstra.stream(...)  
2 YIELD nodeIds, costs, path ...  
3 RETURN ...  
4  
5 nodeIds -- die IDs der Knoten des Ergebnispfades in  
6 -- Form eines Arrays  
7 costs -- Die stufenweisen iterierten Kosten ent-  
8 -- lang des Ergebnispfades  
9 path -- Ergebnispfad
```

Graphalgorithmen

Dijkstra Source Target: Rückgabewerte

```
1 CALL gds.shortestPath.dijkstra.stream(...)  
2 YIELD sourceNode, targetNode, totalCost ...  
3 RETURN ...  
4  
5 sourceNode -- Ausgangsknoten  
6 targetNode -- Zielknoten  
7  
8 totalCost -- Die gesammelten Kosten des Ergebnis-  
9 -- pfades
```

Graphalgorithmen

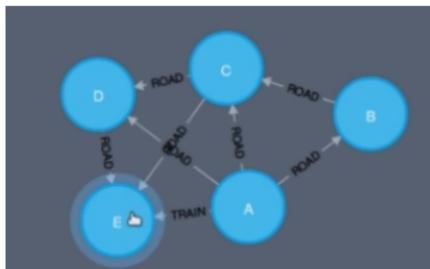
Dijkstra Source Target: Beispiel

```
1 -- Es wird eine Zugverbindung zwischen A und E an-
2 -- gelegt
3
4 MATCH (A:City {name:"A"})
5 MATCH (E:City {name:"E"})
6 CREATE (A)-[:TRAIN {distance: 20}]->(E)
```

Graphalgorithmen

Dijkstra Source Target: Beispiel

Das Ergebnis der Abfrage wird nun die neu eingefügte Verbindung sein.



Graphalgorithmen

Dijkstra Source Target: Beispiel

```
1 -- Berechnen Sie den kuerzesten Weg von A nach E.  
2 -- Beruecksichtigen Sie die Laenge der Strecke.  
3 -- Die Strecke soll auf der Strasse zurueckgelegt  
4 -- werden.  
5 MATCH(A:City{name:"A"})  
6 MATCH(E:City{name:"E"})  
7 CALL gds.shortestPath.dijkstra.stream(  
8     "mygraph", {  
9         sourceNode : A,  
10        targetNode : E,  
11        relationshipWeightProperty: "distance",  
12        relationshipTypes : ["ROAD"]  
13    }  
14 )...
```

Graphalgorithmen

Dijkstra Source Target: Beispiel

```
1 -- Berechnen Sie den kuerzesten Weg von A nach E.  
2 -- Beruecksichtigen Sie die Laenge der Strecke.  
3 -- Die Strecke soll auf der Strasse zurueckgelegt  
4 -- werden.  
5 MATCH(A:City{name:"A"})  
6 ...  
7 YIELD nodeIds, totalCost  
8 RETURN [id in nodeIds| gds.util.asNode(id).name],  
9       totalCost;
```

Graphalgorithmen

Dijkstra Source Target: Beispiel

```
1 -- ... Beruecksichtigen Sie nur City Knoten
2
3 MATCH(A:City{name:"A"})
4 MATCH(E:City{name:"E"})
5 CALL gds.shortestPath.dijkstra.stream(
6   "mygraph", {
7     sourceNode : A,
8     targetNode : E,
9     relationshipWeightProperty: "distance",
10    relationshipTypes : ["ROAD", "TRAIN"],
11    nodeLabels : ["City"]
12  }
13 )
14 YIELD nodeIds, totalCost ...
```

Graphalgorithmen

Dijkstra Single Source

Dijkstra Source-Source: The Dijkstra Single-Source algorithm computes the shortest paths between a source node and all nodes reachable from that node.

Graphalgorithmen

Dijkstra Single Source: Beispiel

```
1 WITH [  
2     ["A", "B", 50], ["A", "C", 50],  
3     ["A", "D", 100], ["B", "D", 40]  
4     ["C", "D", 40], ["C", "E", 40]  
5     ["D", "E", 30], ["D", "F", 80]  
6     ["E", "F", 40]  
7 ] AS data  
8 UNWIND data AS row  
9 MERGE (n:City {name: row[0]})  
10 MERGE (m:City {name: row[1]})  
11 MERGE (n)-[r:ROAD {distance:row[2]}]->(m);
```

Graphalgorithmen

Dijkstra Single Source: Beispiel

```
1 CALL gds.graph.create(  
2     'myGraph',  
3     'Location',  
4     'ROAD',  
5     {  
6         relationshipProperties: 'distance'  
7     }  
8 )
```

Graphalgorithmen

Dijkstra Single Source: Beispiel

```
1 MATCH (source:Location {name: 'A'})  
2 CALL  
    gds.allShortestPaths.dijkstra.stream('myGraph',  
    {  
        3     sourceNode: source,  
        4     relationshipWeightProperty: 'cost'  
    })  
    6 YIELD index, sourceNode, targetNode, totalCost,  
    7         nodeIds, costs, path  
8 RETURN ...
```

Graphalgorithmen

Dijkstra Single Source: Beispiel

```
1 MATCH (source:Location {name: 'A'})  
2 ...  
3 RETURN  
4     index,  
5     gds.util.asNode(sourceNode).name AS source,  
6     gds.util.asNode(targetNode).name AS target,  
7     totalCost,  
8     [nodeId IN nodeIds |  
9         gds.util.asNode(nodeId).name] AS nodeNames,  
10    costs,  
11    nodes(path) as path;  
12 ORDER BY index
```

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

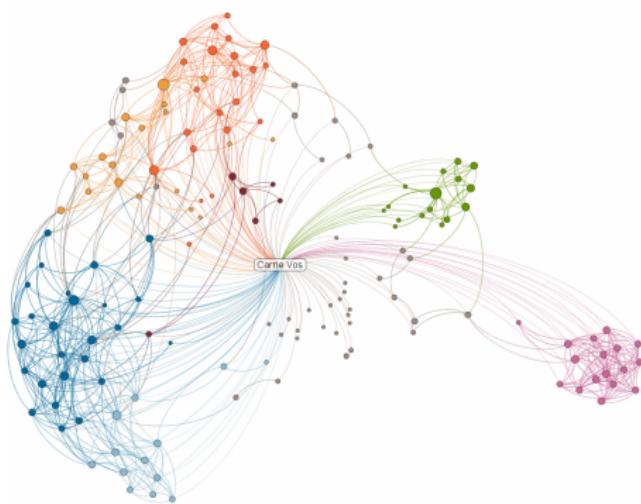
Centrality

Community

Graphalgorithmen

Centrality

Centrality algorithms are used to determine the **importance** of distinct nodes in a network.



Graphalgorithmen

Centrality - Page Rank

Page Rank: The PageRank algorithm measures the **importance** of each node within the graph, based on the number incoming relationships and the importance of the corresponding source nodes.

Hint: The underlying assumption roughly speaking is that a page is only as important as the pages that link to it.

Graphalgorithmen

Page Rank: Beispiel

```
1 CREATE
2   (home:Page {name:'Home'}) ,
3   (about:Page {name:'About'}) ,
4   (product:Page {name:'Product'}) ,
5   (links:Page {name:'Links'}) ,
6   (a:Page {name:'Site A'}) ,
7   (b:Page {name:'Site B'}) ,
8   (c:Page {name:'Site C'}) ,
9   (d:Page {name:'Site D'}) ,
10
11  (home)-[:LINKS {weight: 0.2}]->(about) ,
12  (home)-[:LINKS {weight: 0.2}]->(links) ,
13  (home)-[:LINKS {weight: 0.6}]->(product) ,
14  (about)-[:LINKS {weight: 1.0}]->(home) ,
```

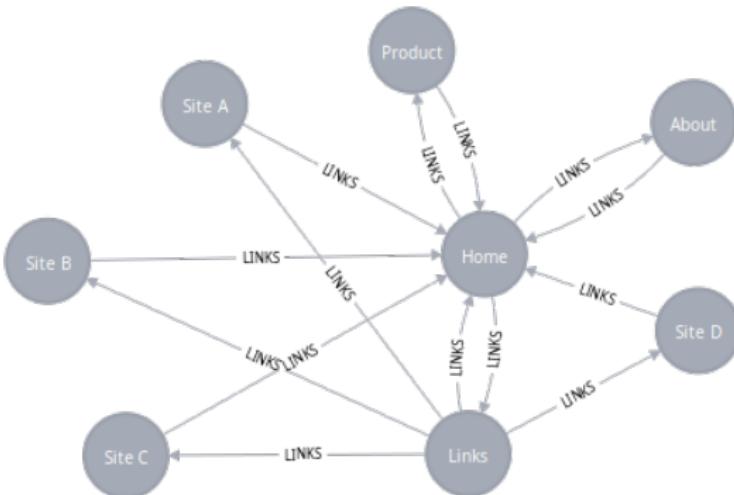
Graphalgorithmen

Page Rank: Beispiel

```
1 CREATE
2 ...
3 (product)-[:LINKS {weight: 1.0}]->(home) ,
4 (a)-[:LINKS {weight: 1.0}]->(home) ,
5 (b)-[:LINKS {weight: 1.0}]->(home) ,
6 (c)-[:LINKS {weight: 1.0}]->(home) ,
7 (d)-[:LINKS {weight: 1.0}]->(home) ,
8 (links)-[:LINKS {weight: 0.8}]->(home) ,
9 (links)-[:LINKS {weight: 0.05}]->(a) ,
10 (links)-[:LINKS {weight: 0.05}]->(b) ,
11 (links)-[:LINKS {weight: 0.05}]->(c) ,
12 (links)-[:LINKS {weight: 0.05}]->(d) ;
```

Graphalgorithmen

Page Rank: Beispiel



Graphalgorithmen

Page Rank: Beispiel

```
1 CALL gds.graph.create(  
2     "myGraph",  
3     "Page",  
4     "LINKS",  
5     {  
6         relationshipProperties: 'weight'  
7     }  
8 )
```

Graphalgorithmen

Page Rank: Beispiel

```
1 -- Berechnen Sie fur jeden Knoten des Graphen
2 -- seinen Pagerank.
3 CALL gds.pageRank.stream('myGraph')
4 YIELD nodeId, score
5 RETURN gds.util.asNode(nodeId).name AS name, score
6 ORDER BY score DESC, name ASC
```

Graphalgorithmen

Centrality - Betweenness Centrality

Betweenness Centrality: Betweenness centrality is a way of detecting the amount of influence a node has over the flow of information in a graph. It is often used to find nodes that serve as a bridge from one part of a graph to another.

Hint: The underlying assumption roughly speaking is that a page is only as important as the pages that link to it.

Graphalgorithmen

Centrality - Betweenness Centrality

The algorithm calculates unweighted shortest paths between all pairs of nodes in a graph.

Each node receives a score, based on the number of shortest paths that pass through the node. Nodes that more **frequently** lie on shortest paths between other nodes will have higher betweenness centrality scores.

Graphalgorithmen

Centrality - Betweenness Centrality

```
1 CREATE
2   (a:User {name: 'Alice'}),  

3   (b:User {name: 'Bob'}),  

4   (c:User {name: 'Carol'}),  

5   (d:User {name: 'Dan'}),  

6   (e:User {name: 'Eve'}),  

7   (f:User {name: 'Frank'}),  

8   (g:User {name: 'Gale'}),  

9   ...
```

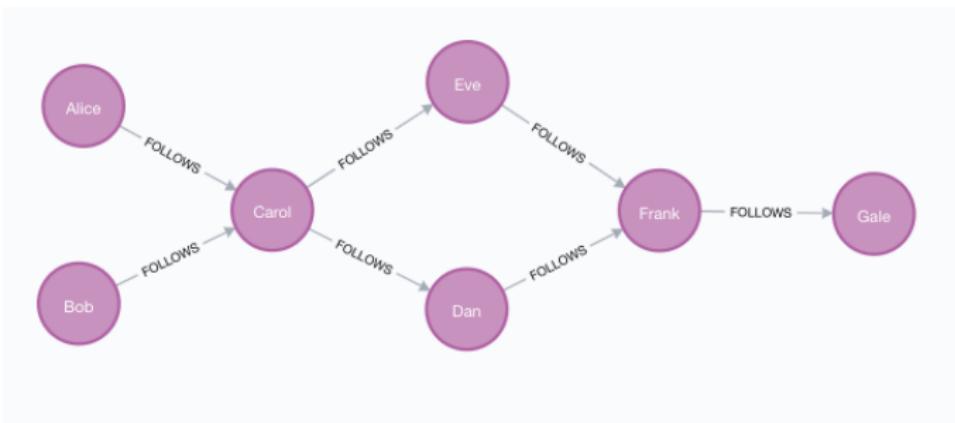
Graphalgorithmen

Centrality - Betweenness Centrality

```
1 CREATE
2 ...
3 (a)-[:FOLLOWS]->(c),
4 (b)-[:FOLLOWS]->(c),
5 (c)-[:FOLLOWS]->(d),
6 (c)-[:FOLLOWS]->(e),
7 (d)-[:FOLLOWS]->(f),
8 (e)-[:FOLLOWS]->(f),
9 (f)-[:FOLLOWS]->(g);
```

Graphalgorithmen

Centrality - Betweenness Centrality



Graphalgorithmen

Centrality - Betweenness Centrality

```
1 CALL gds.graph.create(  
2     "myGraph", "User", "FOLLOWS"  
3 );
```

Graphalgorithmen

Centrality - Betweenness Centrality

```
1 CALL gds.betweenness.stream("myGraph")
2 YIELD nodeId, score
3 RETURN gds.util.asNode(nodeId).name AS name, score
4 ORDER BY name ASC
```

Graphalgorithmen

Centrality - Betweenness Centrality

```
1 "Alice" 0.0
2 "Bob" 0.0
3 "Carol" 8.0
4 "Dan" 3.0
5 "Eve" 3.0
6 "Frank" 5.0
7 "Gale" 0.0
```

Graphalgorithmen

Centrality - Degree Centrality

Degree Centrality: The Degree Centrality algorithm can be used to find **popular nodes** within a graph.

Degree centrality measures the number of incoming or outgoing (or both) relationships from a node, depending on the orientation of a relationship projection.

Graphalgorithmen

Centrality - Degree Centrality

It can be applied to either **weighted** or **unweighted** graphs.

In the weighted case the algorithm computes the sum of all positive weights of adjacent relationships of a node, for each node in the graph.

Graphalgorithmen

Centrality - Degree Centrality

```
1 CREATE
2   (a:User {name: 'Alice'}) ,
3   (b:User {name: 'Bridget'}) ,
4   (c:User {name: 'Charles'}) ,
5   (d:User {name: 'Doug'}) ,
6   (m:User {name: 'Mark'}) ,
7   (m:User {name: 'Michael'}) ,
8   ...
```

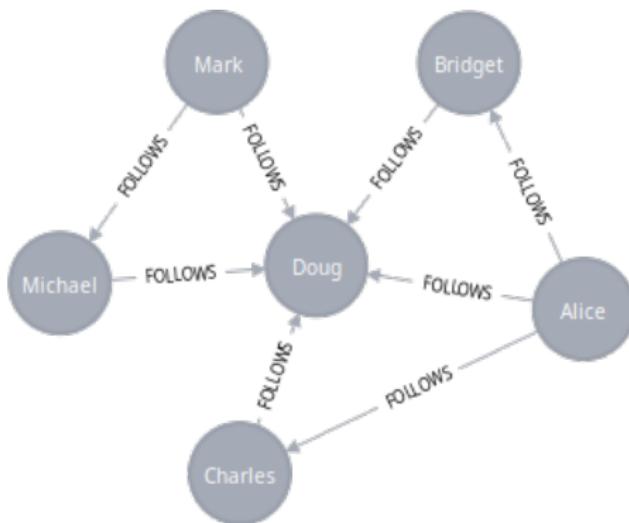
Graphalgorithmen

Centrality - Degree Centrality

```
1 CREATE
2   (a)-[:FOLLOWS {score: 1}]->(d),
3   (a)-[:FOLLOWS {score: -2}]->(b),
4   (a)-[:FOLLOWS {score: 5}]->(c),
5   (m)-[:FOLLOWS {score: 1.5}]->(d),
6   (m)-[:FOLLOWS {score: 4.5}]->(m),
7   (b)-[:FOLLOWS {score: 1.5}]->(d),
8   (c)-[:FOLLOWS {score: 2}]->(d),
9   (m)-[:FOLLOWS {score: 1.5}]->(d);
```

Graphalgorithmen

Centrality - Degree Centrality



Graphalgorithmen

Centrality - Degree Centrality

```
1 CALL gds.graph.create(  
2     "myGraph",  
3     "User",  
4     {  
5         FOLLOWS: {  
6             orientation: "REVERSE",  
7             properties: ["score"]  
8         }  
9     }  
10 )
```

Graphalgorithmen

Centrality - Degree Centrality

```
1 CALL gds.degree.stream('myGraph')
2 YIELD nodeId, score
3 RETURN gds.util.asNode(nodeId).name AS name,
4       score AS followers
5 ORDER BY followers DESC, name DESC
```

Graphalgorithmen

Centrality - Degree Centrality

1	"Doug"	5.0
2	"Michael"	1.0
3	"Charles"	1.0
4	"Bridget"	1.0
5	"Mark"	0.0
6	"Alice"	0.0

① Graphentheorie

Grundlagen

② CQL - Cypher Query Language

Cypher Grundlagen

Cypher - DDL Commands

③ APOC - Funktionen

APOC - Cypher Funktions

④ Graphen Algorithmen

Graphusage

CALL Klausel

Path finding

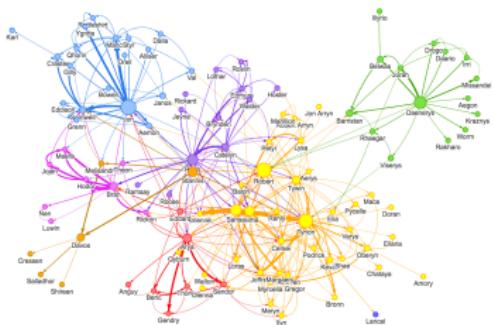
Centrality

Community

Graphalgorithmen

Community

Community detection algorithms are used to evaluate how groups of nodes are **clustered** or **partitioned**, as well as their tendency to strengthen or break apart.



Graphalgorithmen

Community - Label Propagation

Label Propagation: The Label Propagation algorithm is a fast algorithm for finding **communities** in a graph.

It detects these communities using network structure alone as its guide, and doesn't require a pre-defined objective function or prior information about the communities.

Graphalgorithmen

Community - Label Propagation

Label Propagation Algorithm:

- 1.Step: Every node is initialized with a unique community label (an **identifier**).
- 2.Step: These labels **propagate** through the network.
- 3.Step: At every iteration of propagation, each node updates its label to the one that the maximum numbers of its neighbours belongs to.

Ties are broken arbitrarily but deterministically.

Graphalgorithmen

Community - Label Propagation

- 4.Step: LPA reaches convergence when each node has the majority label of its neighbours.
- 5.Step: LPA stops if either convergence, or the user-defined maximum number of iterations is achieved.

Graphalgorithmen

Community - Label Propagation

1 CREATE

```
2   (a:User {name: 'Alice', seed_label: 52}),  
3   (b:User {name: 'Bridget', seed_label: 21}),  
4   (c:User {name: 'Charles', seed_label: 43}),  
5   (d:User {name: 'Doug', seed_label: 21}),  
6   (m:User {name: 'Mark', seed_label: 19}),  
7   (mi:User {name: 'Michael', seed_label: 52}),  
8   ...
```

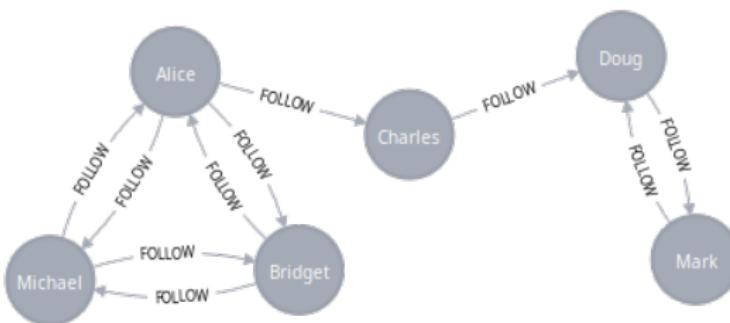
Graphalgorithmen

Community - Label Propagation

```
1 CREATE
2 ...
3 (a)-[:FOLLOW {weight: 1}]->(b),
4 (a)-[:FOLLOW {weight: 10}]->(c),
5 (m)-[:FOLLOW {weight: 1}]->(d),
6 (b)-[:FOLLOW {weight: 1}]->(mi),
7 (d)-[:FOLLOW {weight: 1}]->(m),
8 (a)-[:FOLLOW {weight: 1}]->(mi),
9 (b)-[:FOLLOW {weight: 1}]->(a),
10 (c)-[:FOLLOW {weight: 1}]->(d)
11 (mi)-[:FOLLOW {weight: 1}]->(b),
12 (mi)-[:FOLLOW {weight: 1}]->(a),
```

Graphalgorithmen

Community - Label Propagation



Graphalgorithmen

Community - Label Propagation

```
1 CALL gds.graph.create(  
2     "myGraph",  
3     "User",  
4     "FOLLOW",  
5     {  
6         nodeProperties: "seed_label",  
7         relationshipProperties: "weight"  
8     }  
9 )
```

Graphalgorithmen

Community - Label Propagation

```
1 CALL gds.labelPropagation.stream('myGraph')
2 YIELD nodeId, communityId AS Community
3 RETURN gds.util.asNode(nodeId).name AS Name,
4       Community
5 ORDER BY Community, Name
```

Graphalgorithmen

Community - Label Propagation

```
1 "Alice"    1
2 "Bridget"  1
3 "Michael"  1
4 "Charles"  4
5 "Doug"     4
6 "Mark"     4
```