

we
focus
on
students



Moderne Datenbanken

Graph-Datenbanken

**Fachhochschule
Dortmund**

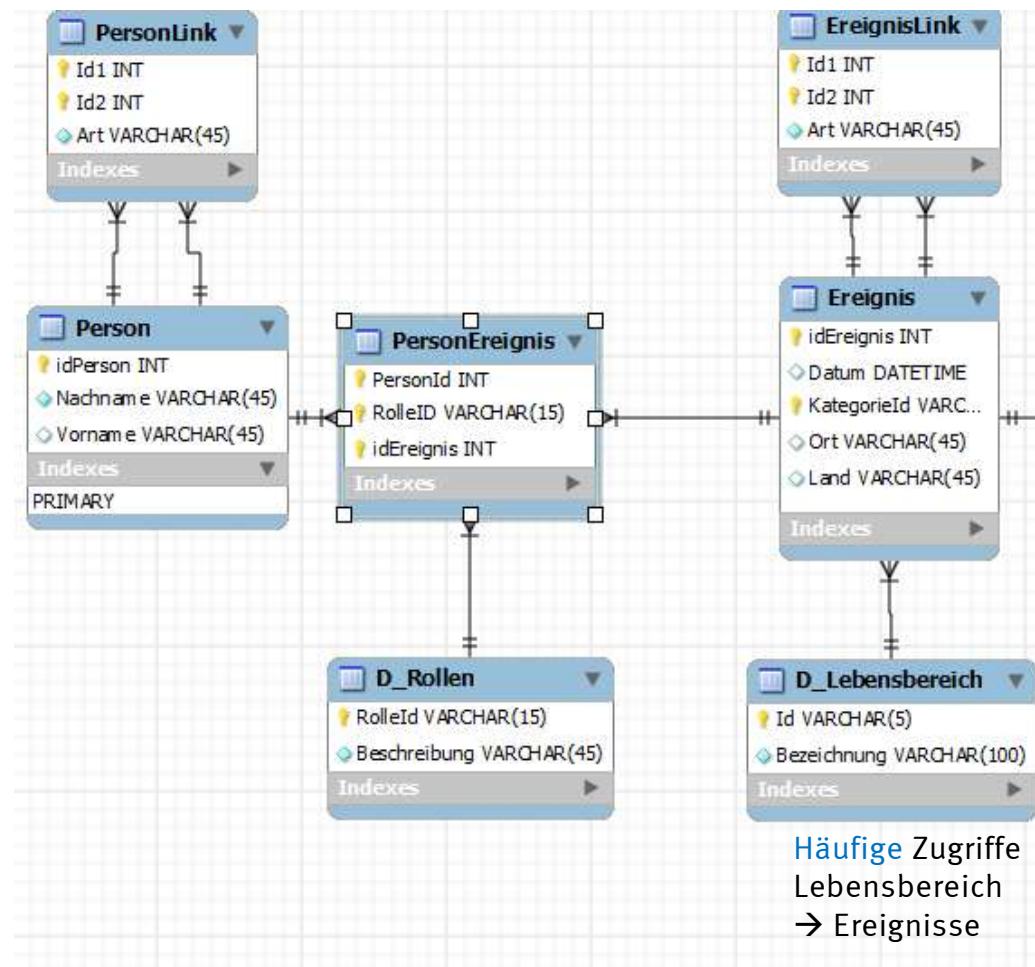
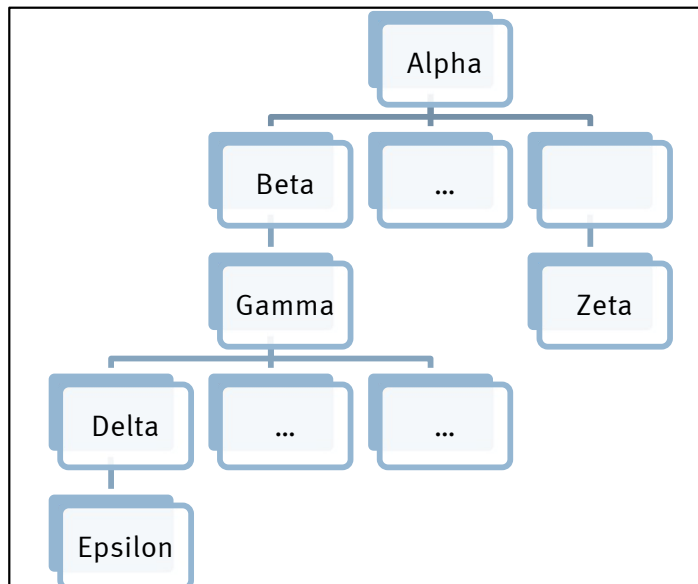
University of Applied Sciences

© 2026 - Prof. Dr. Inga Marina Saatz

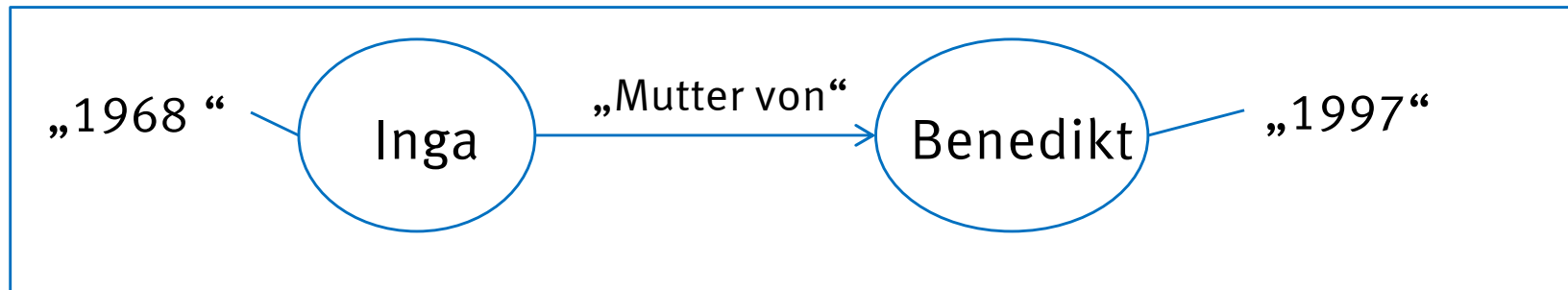
1	Datenbankmodell	2
2	Datenmodellierung	6
3	Graphendatenbank neo4j	18
4	Traversierung von Graphen	35
5	Vergleich mit anderen Datenmodellen	43

Eine **Aggregation** erfordert das Zusammenführen von Objekten, welches durch das relationale Modell **nicht** direkt unterstützt wird. Für die Abbildung von häufigen N:M-Beziehungen, Rekursion und Aggregationsbeziehungen ist das Graph-Datenbankmodell geeignet.

Mitarbeiterhierarchie



- Datenmodell:
 - Verwaltung von Graph- oder Baumstrukturen mit denen die Elemente miteinander verknüpft sind
 - Knoten und Kanten des Graphen werden mit Eigenschaften versehen oder gewichtet.
- Beispiel: „Inga(1968) ist Mutter von Benedikt(1997)“



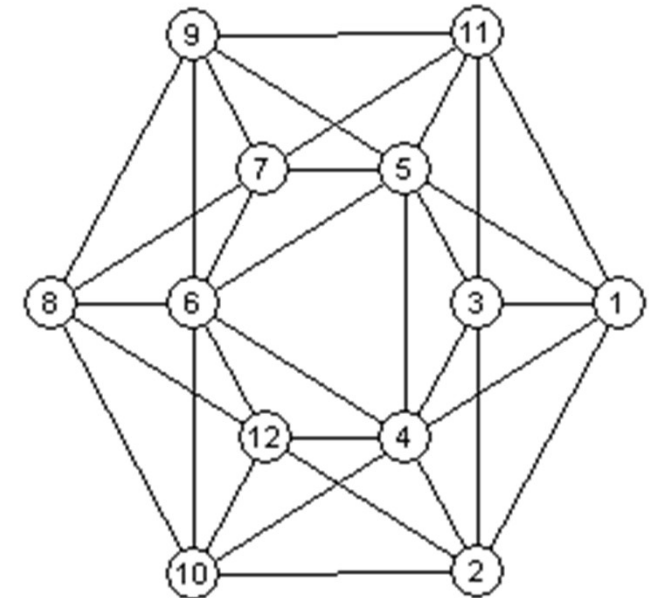
- Anwendungen:
 - Location-Based-Services zur Verknüpfung von sozialen Netzen mit Geodaten
 - Pfadsuche
 - Molekülmodellierung, ...
- Datenbank-Beispiele:
 - Neo4j, SonesDB

■ Typische Fragestellungen

- Wie können Beziehungen zwischen Knoten abgebildet werden?
- Abfragen:
 - Welche Knoten sind von Knoten 10 aus erreichbar?
 - Welche Knoten sind die k. nächsten Nachbarn vom Knoten 10?
 - Welches ist die kürzeste Verbindung zwischen zwei Knoten?
 - Welches ist der kürzeste Weg, um alle Knoten genau 1x zu durchlaufen?

■ Beispiele typischer Anwendungsfälle

- Soziale Netzwerke (social graph)
- Kaufereignisse (consumption graph)
- Interessen, z.B. besuchte Webseiten (interest graph)
- Empfehlungssysteme (intent graph)
- Logistische Fragestellungen und Location-Based Services (location graph)



1	Datenbankmodell	2
2	Datenmodellierung	6
3	Graphendatenbank neo4j	18
4	Traversierung von Graphen	35
5	Vergleich mit anderen Datenmodellen	43

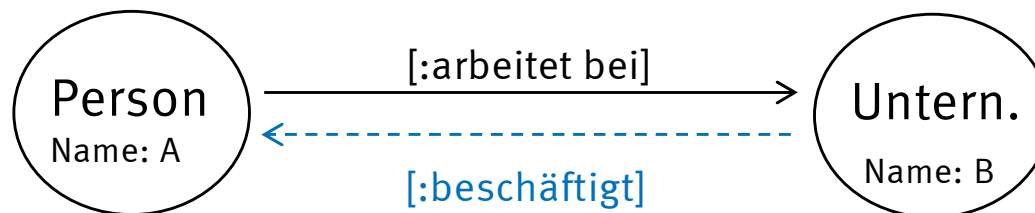
■ Begriffe

ER-Modell	Relationales Modell	Graphmodell
Entität	Relation	Knoten (node)
Beziehung	Fremdschlüssel	Kante (relationship)
Attribut	Attribut	Eigenschaft (property)

■ Kantenarten

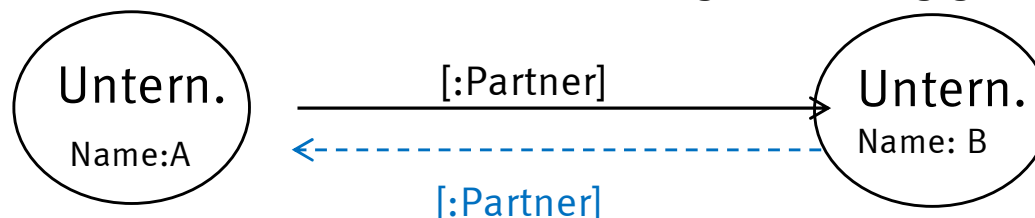
● Unidirektionale Beziehung

- Wird durch eine gerichtete Kante auch **implizit** die umgekehrte (**blaue**) Beziehung modelliert, so genügt es, nur eine der beiden gerichteten Kanten im Graphmodell zu modellieren.

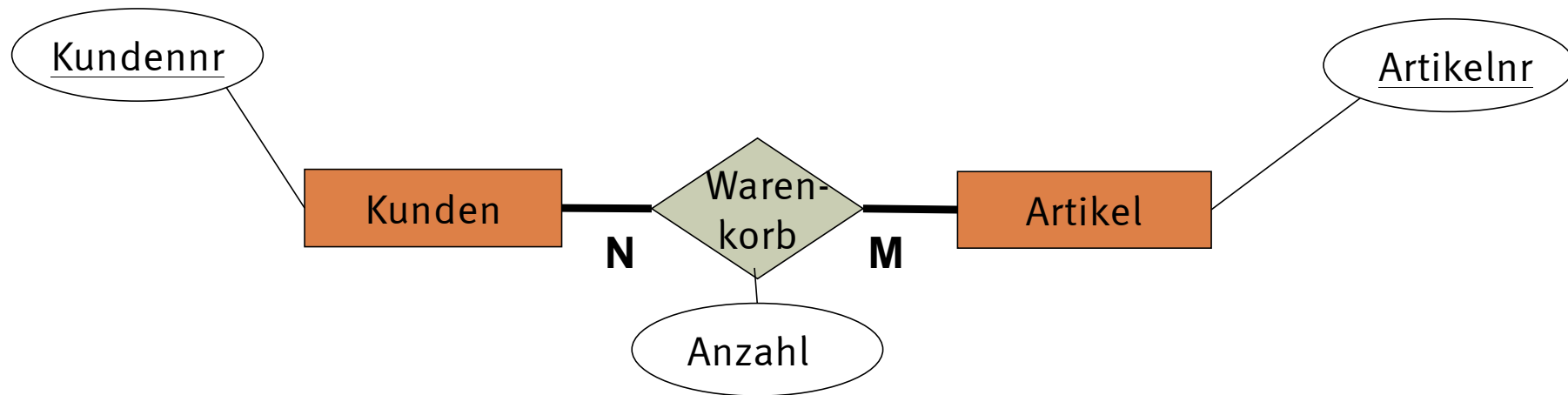


● Bidirektionale Beziehung

- Bei einer bidirektionalen Beziehung kann die Richtung nicht eindeutig festgelegt werden. In diesem Fall wird eine beliebige Richtung gewählt.

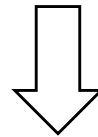


Beispiel 1: Interest Graph

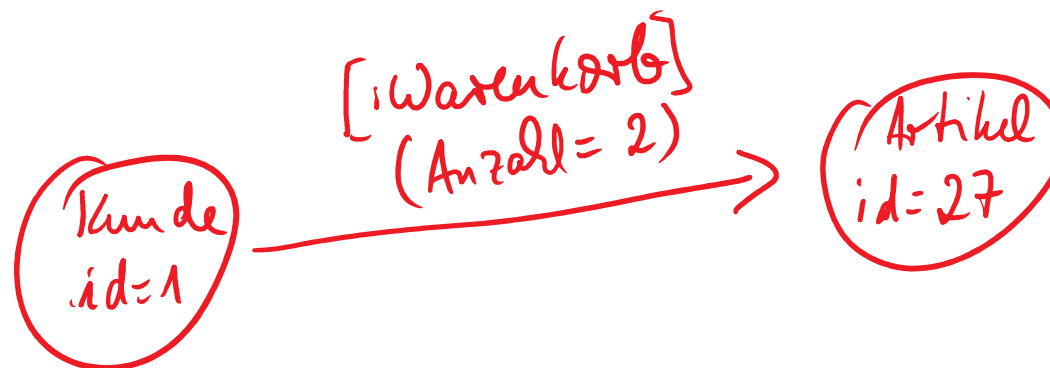


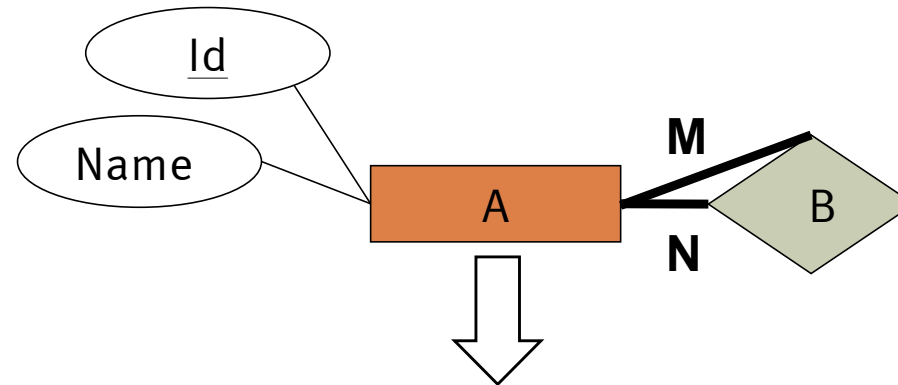
- Art der Beziehung:

unidirectional

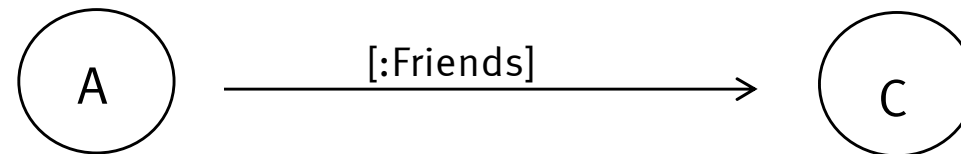


- Graph-Modellierung:

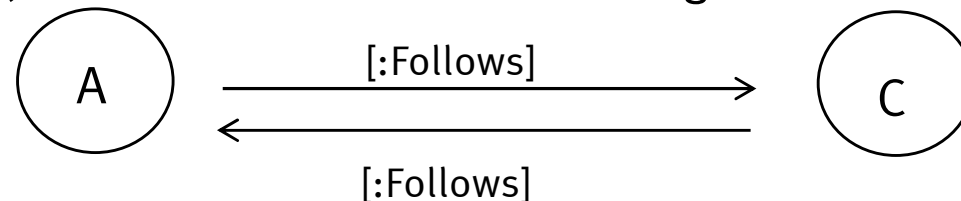




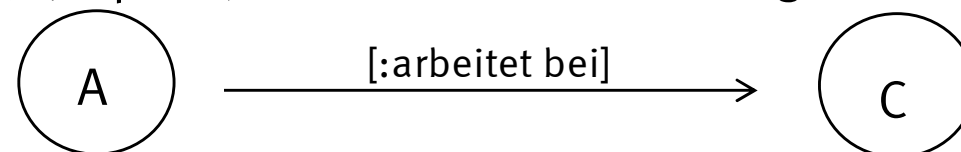
1. Bidirektionale (gleichberechtigte) Beziehung



2. Unidirektionale, nicht umkehrbare Beziehung

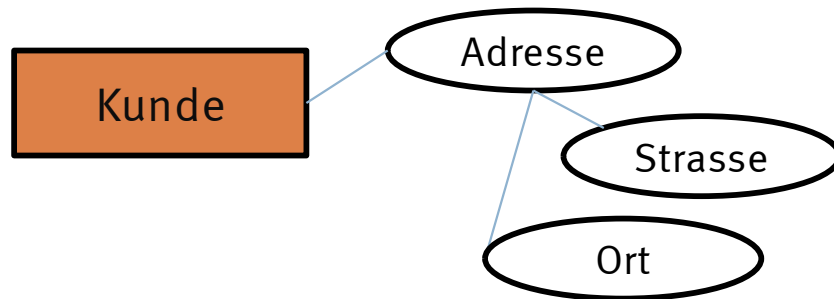


3. Unidirektionale, (implizit) umkehrbare Beziehung

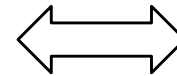
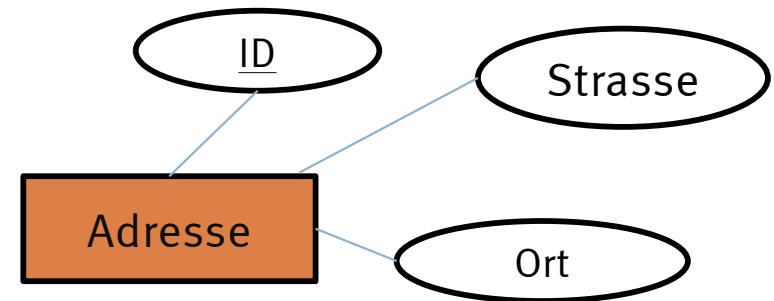


Ein Lokationsgraph (location graph) dient zur Modellierung von Objekten, die speziellen Orten (Lokationen) zugeordnet werden.

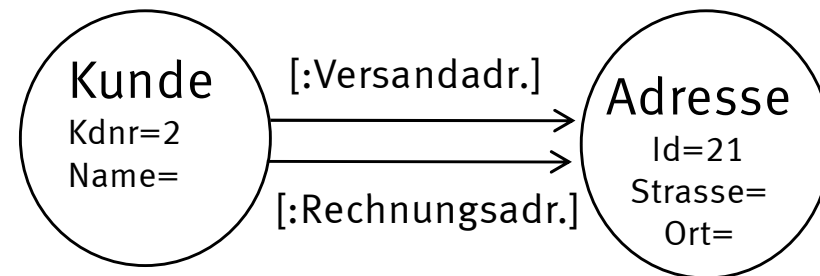
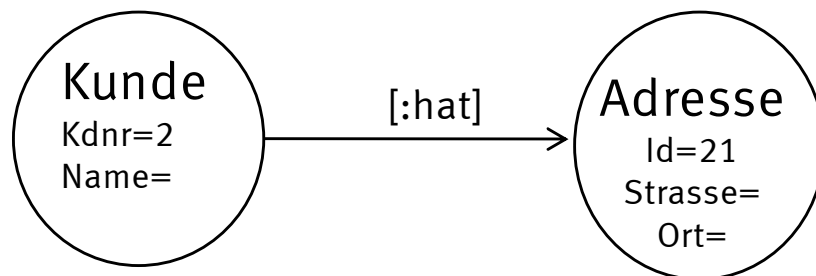
Strukturiertes Attribut



Adresse als Entität

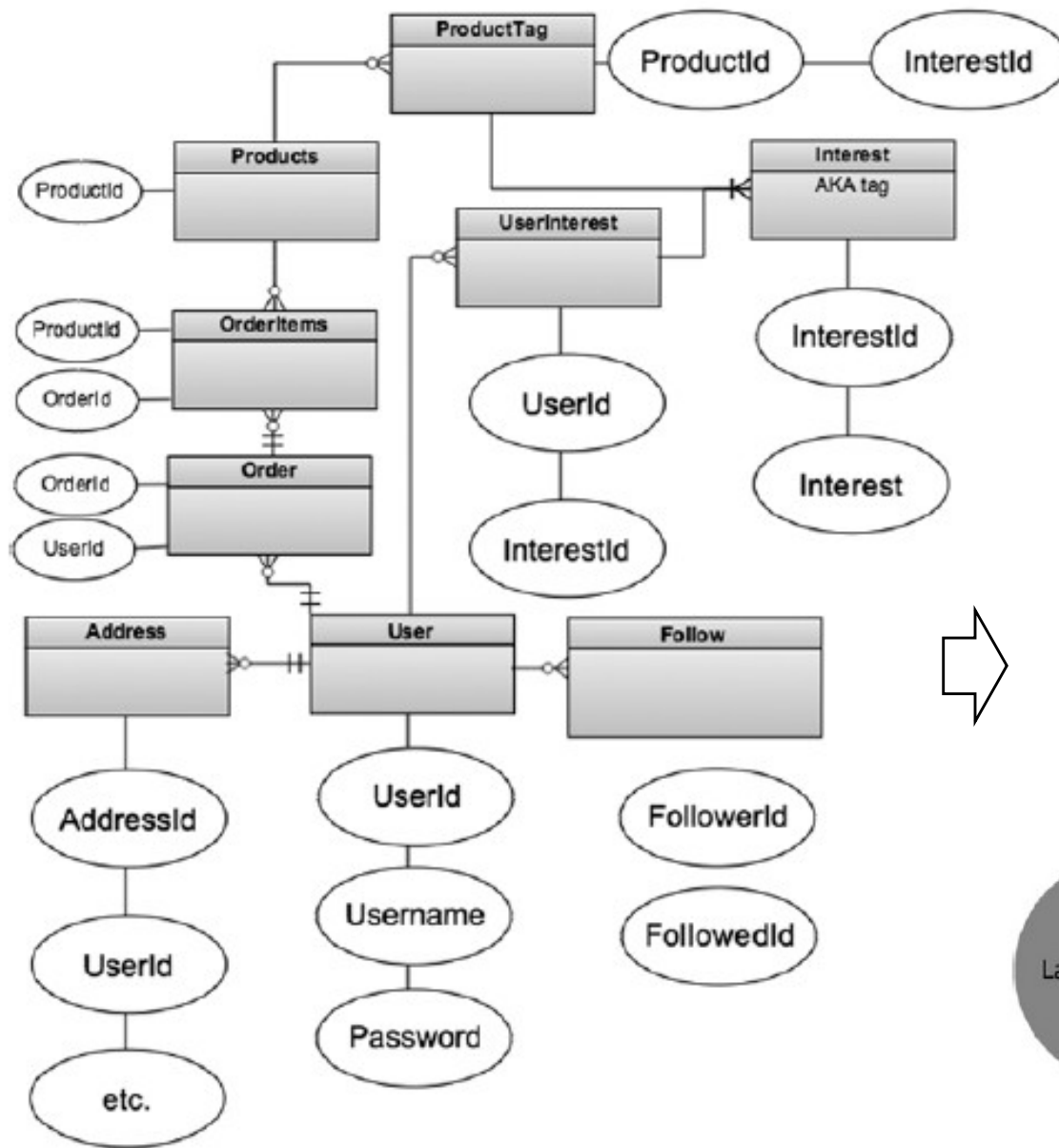


Im Graphmodell:



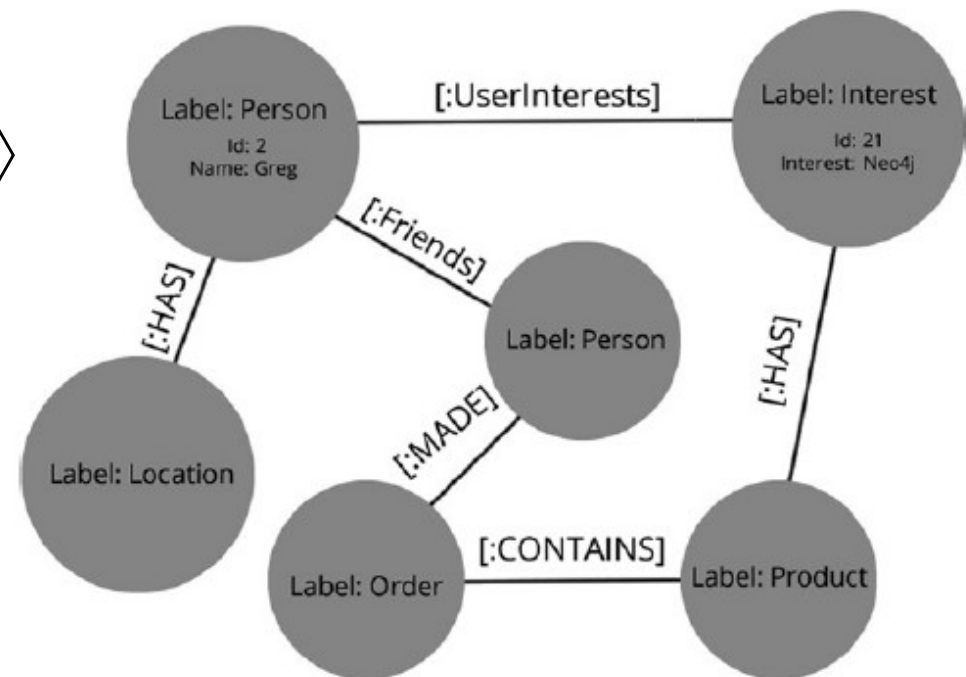
Vorteil: Das Einfügen weiterer Adressen und Lokationen ist im Graphmodell einfach umzusetzen.

Relationales Modell

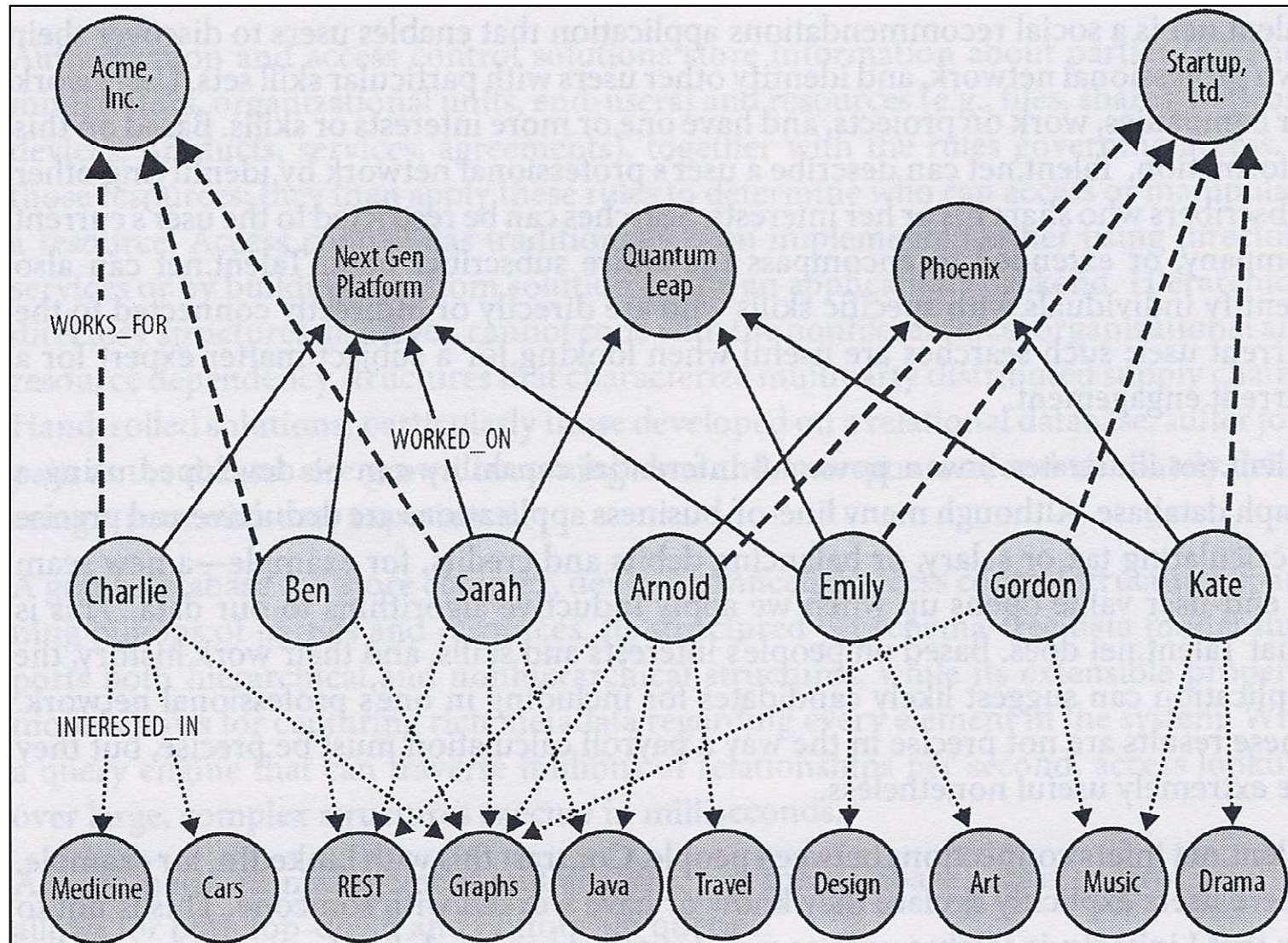


Ein Absichtsgraph (intent graph) dient zur Modellierung von Absichten, bspw. zur Ermittlung von Kaufempfehlungen. Hierfür werden die verschiedenen anderen Graphen miteinander kombiniert.

Graphmodell



Beispiel: Soziales Netzwerk & interest graph



1. A group of two or more people organize into a fraud ring
2. The ring shares a subset of legitimate contact information, for example phone numbers and addresses, combining them to create a number of synthetic identities
3. Ring members open accounts using these synthetic identities
4. New accounts are added to the original ones: unsecured credit lines, credit cards, overdraft protection, personal loans, etc.
5. The accounts are used normally, with regular purchases and timely payments
6. Banks increase the revolving credit lines over time, due to the observed responsible credit behavior
7. One day the ring "busts out", coordinating their activity, maxing out all of their credit lines, and disappearing
8. Sometimes fraudsters will go a step further and bring all of their balances to zero using fake checks immediately before the prior step, doubling the damage
9. Collections processes ensue, but agents are never able to reach the fraudster
10. The uncollectible debt is written off

Beispiel:

2 Personen verwenden 2 Adressen und 2 Telefonnummern zur Erstellung von 4 Fake-Identitäten.

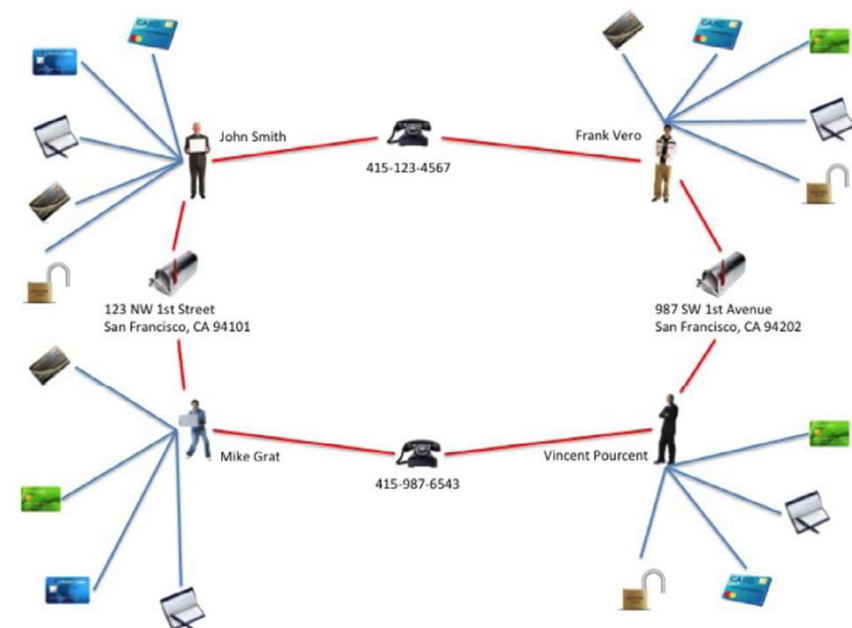
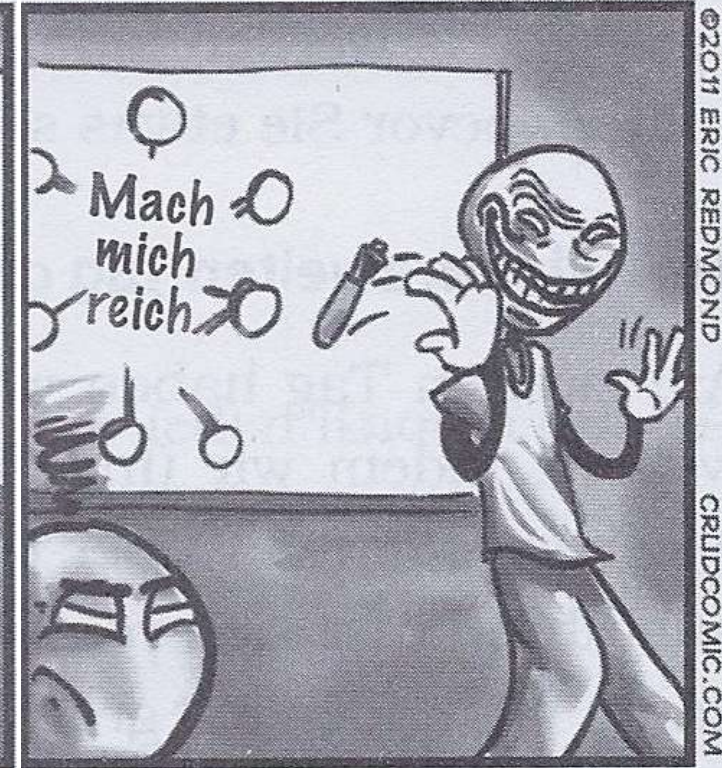


Diagram 1: 2 people sharing 2 pieces of data and creating 4 synthetic identities

1	Datenbankmodell	2
2	Datenmodellierung	6
3	Graphendatenbank neo4j	18
4	Traversierung von Graphen	35
5	Vergleich mit anderen Datenmodellen	43

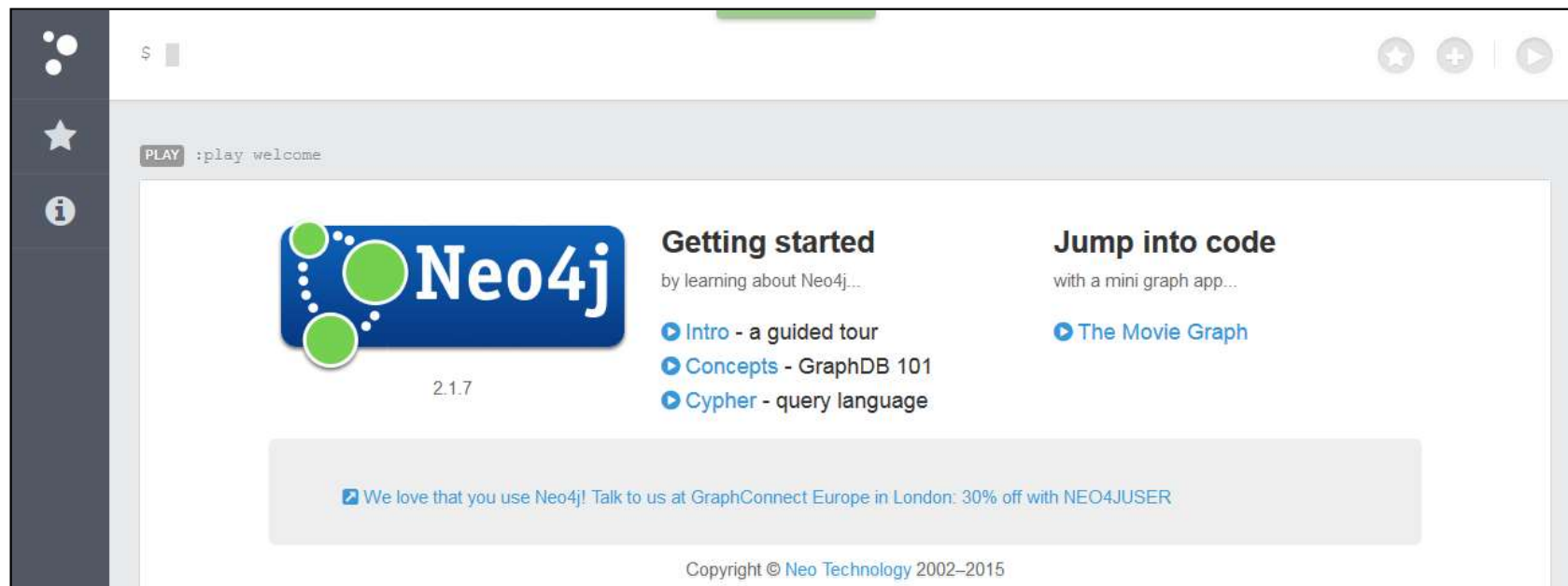


- Neo4j zählt zu den **Graphdatenbanken**
 - Entwickelt in Java durch Neo Technology
 - Open Source
 - eingebettete, dateibasierte, transaktionale Datenbank



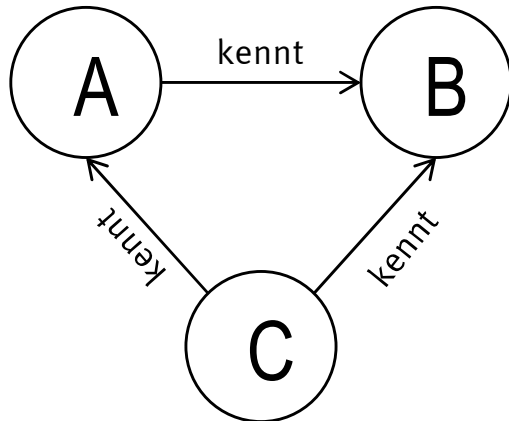
<http://neo4j.org>

Browseransicht: <http://localhost:7474/browser>



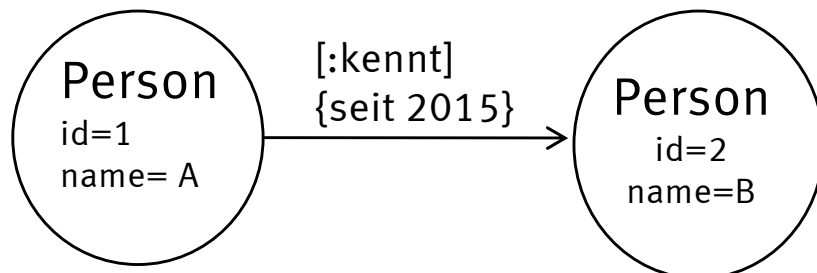
Cypher ist eine neo4j-spezifische deklarative Anfragesprache für Graphdatenbanken, die auch Modifikationen des Graphen ermöglicht.

- Alle Änderungen erfolgen in einem ACID-Transaktionskontext.
- Einfügen von Knoten und Beziehungen:



```
CREATE (a {pid:1, name:"A", _label: "Person"}),  
         (b {pid:2, name:"B", _label: "Person"}),  
         (c {pid:3, name:"C", _label: "Person"}),  
         (a)-[:kennt] -> (b),  
         (c)-[:kennt]-> (a),  
         (c)-[:kennt]-> (b)
```

Alias (nur temporär)

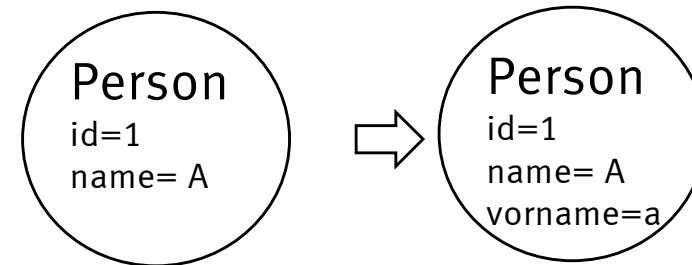


```
CREATE (a {id:1, name: 'A', _label: 'Person'}),  
         (b{id:2, name: 'B', _label: 'Person'}),  
         (a)-[:kennt {seit: 2015}] ->(b)
```

- Attribut hinzufügen

```
CREATE (a {id:1, name: 'A'})
```

```
MATCH (a {id:1})  
SET a+= {vorname: 'A'})
```



- Integritätsregeln hinzufügen

- Constraint erzeugen

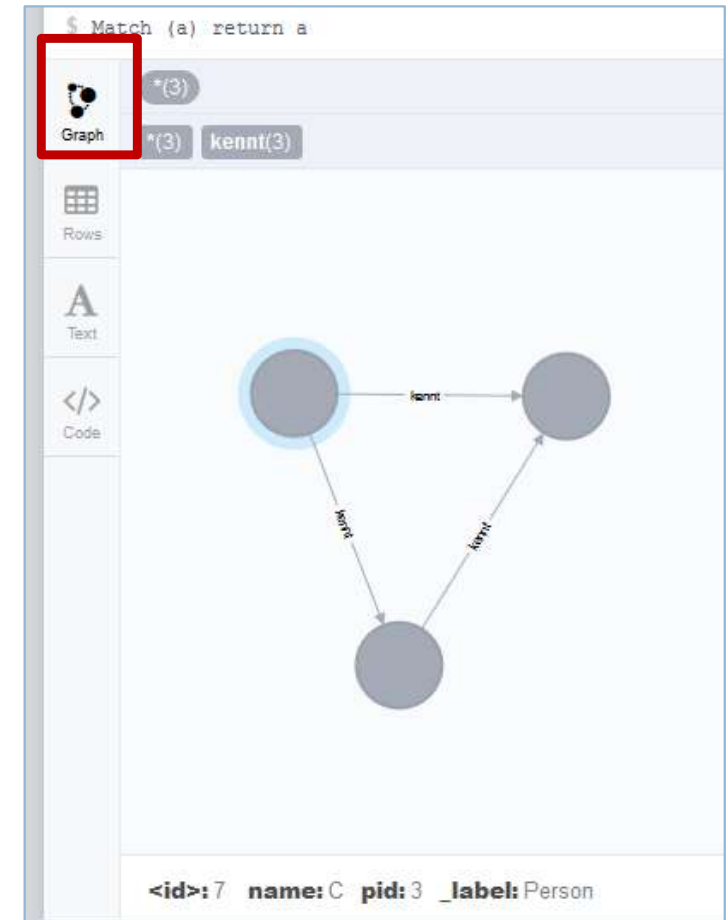
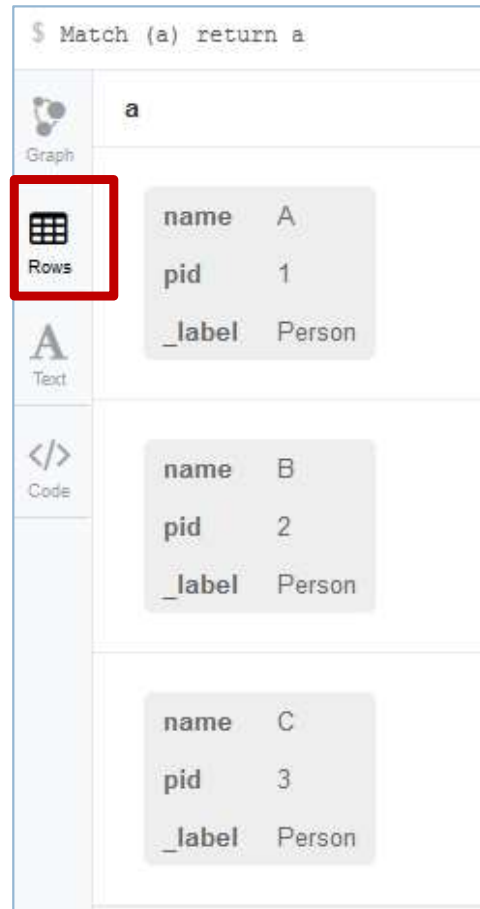
```
CREATE CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

- Constraint löschen

```
DROP CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

- Selektieren aller Knoten:

MATCH (a) **Return a**

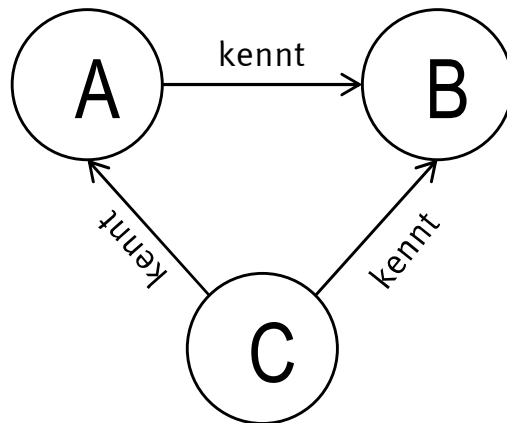


- Abfrage mit Projektion:

MATCH (a) **Return a {.name, .pid}**

Anfragen werden formuliert in der Form von Mustern im Graphen (specification by example), die sich auch auf Instanzen beziehen können.

- Beispiel: „Liefere drei Freunde, die sich gegenseitig kennen“



Variable

```
MATCH
(a{name:"A"}) -[:kennt]-> (b)
RETURN b.name
```

Specification by example

Rückgabe (Projektion)

Richtung der Kante

- Liefere alle Personen:

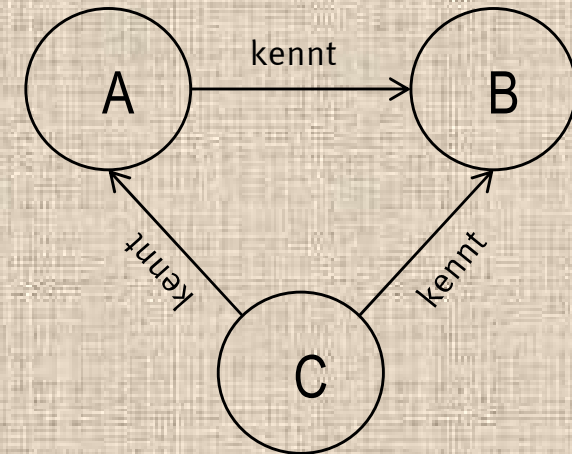
```
MATCH (a {_label:'person'})  
RETURN a
```

- Was liefern diese Abfragen?

```
MATCH (a{name:'A'}) -[:kennt]->(b)  
RETURN a,b
```

```
MATCH (a{name:'A'}) <-[:kennt]-(b)  
RETURN b
```

```
MATCH (a{name:'C'}) -[:kennt]->(b)  
OPTIONAL MATCH (b) -[:kennt]->(c)  
RETURN c
```

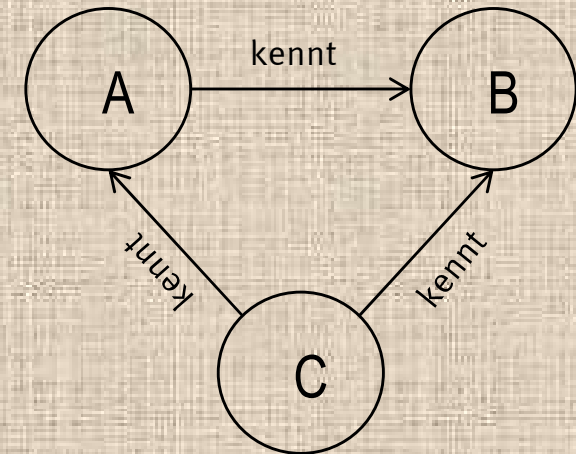


Schlüsselwort	Bedeutung
WHERE	Filterung der Ergebnismenge
CREATE CREATE UNIQUE	Einfügen von Knoten, Kanten und Eigenschaften
DELETE	Löschen von Knoten, Kanten und Eigenschaften
SET	Setzen von Eigenschaftswerten
FOREACH	Änderungsoperation wird auf alle Elemente einer Liste angewendet
UNION	Vereinigung von mehreren Ergebnismengen
WITH	Aneinanderhängen von Anfragen
ORDER BY	Sortierung
LIMIT	Begrenzung der Rückgabemenge
USING	Spezifizierung eines zu nutzenden Index
DISTINCT	Entfernung von Duplikaten
SKIP	Überspringen von Knoten (in der Ausgabe)
TYPE(<i>bezeichnung</i>)	Ausgabe der Kantenbezeichnung <i>bezeichnung</i>

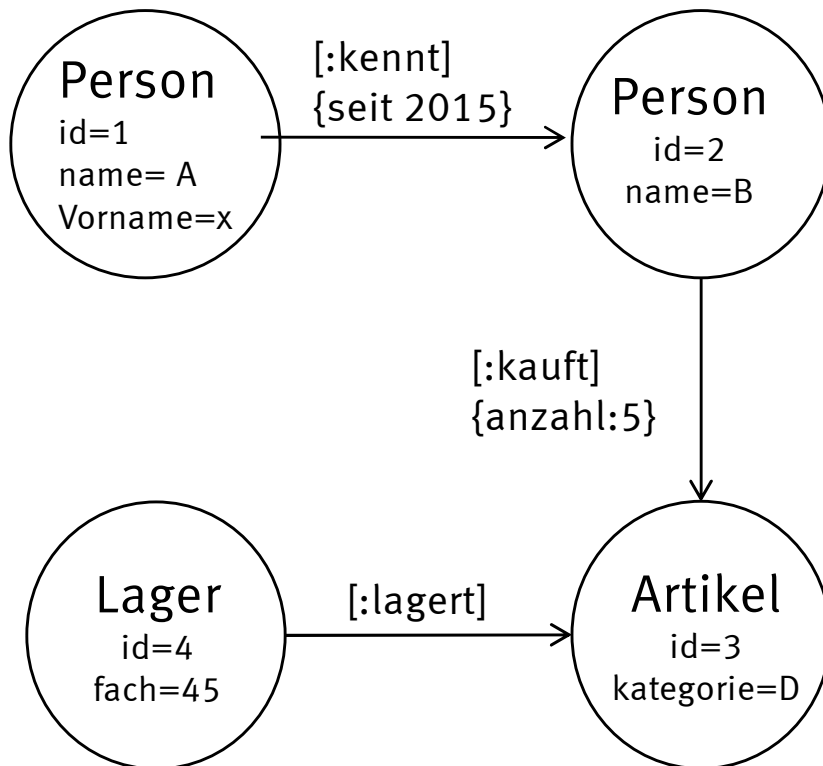
Was liefern die folgenden Abfragen?

```
MATCH (user)-[:kennt]->(follower)
WHERE user.name IN ['B','C']
RETURN user, follower.name
```

```
MATCH (user)-[:kennt]->(follower)
WHERE user.name IN ['B','C']
AND follower.name =~ 'A.*'
RETURN user, follower.name
```



- Strukturierung zusammengesetzter Anfragen



```
MATCH (a:person {id:1}) – [:kennt*0..1] -> b
```

```
WITH DISTINCT b, a
```

```
MATCH (b)-[:kauft]-(artikel)-[:lagert*0..1]-(lager)
```

```
RETURN lager, lager.fach as Fachnummer,
```

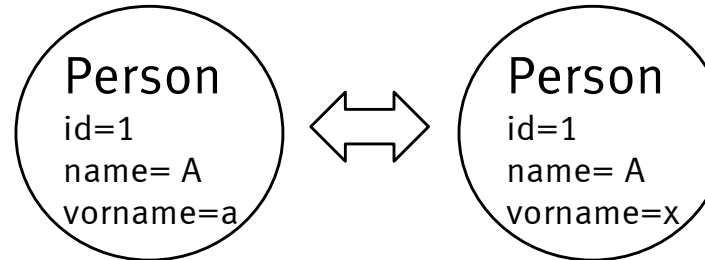
```
ORDER BY lager.fach desc
```

```
LIMIT 4
```



- Hinzufügen und Entfernen eines Attributs:

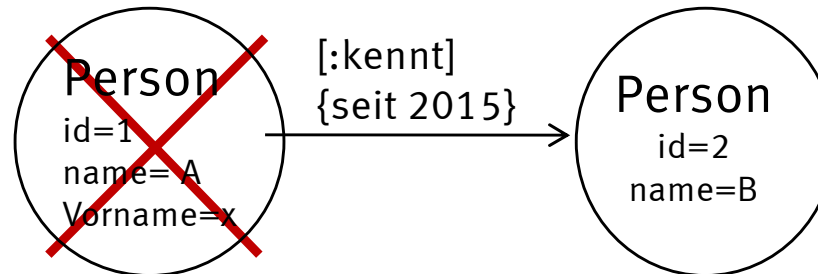
```
MATCH (a:Person {id:1})
SET a.vorname= 'x'
RETURN a
```



```
MATCH (a:Person {id:1})
REMOVE a.vorname
RETURN a
```

- Löschen eines Knotens:

```
MATCH (a:Person {id:1})
DELETE a
```

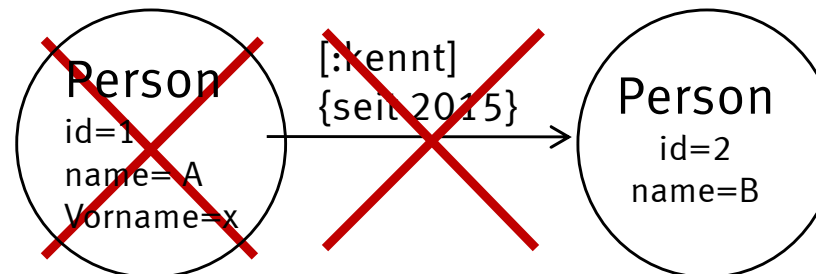


Dangling
References

Löschen eines Knotens mit allen seinen Kanten:

```
MATCH (a:Person {id:1})-[r]-()
DELETE a
```

```
MATCH (a{id:1})
OPTIONAL MATCH (a) - [r] - ()
DELETE a, r
```



- Räumliche Abstandberechnungen

```
WITH
  point({ x: 2.3, y: 4.5, crs: 'cartesian' }) AS p1,
  point({ x: 1.1, y: 5.4, crs: 'cartesian' }) AS p2
RETURN distance(p1,p2) AS dist
```

- Aggregationsfunktionen, z.B. SUM, MIN, MAX

```
MATCH (n:Person) RETURN MAX(n.age)
```

- Nutzerdefinierte Funktionen

```
MATCH (p: Person)
RETURN org.neo4j.examples.longestString(p.name)
```

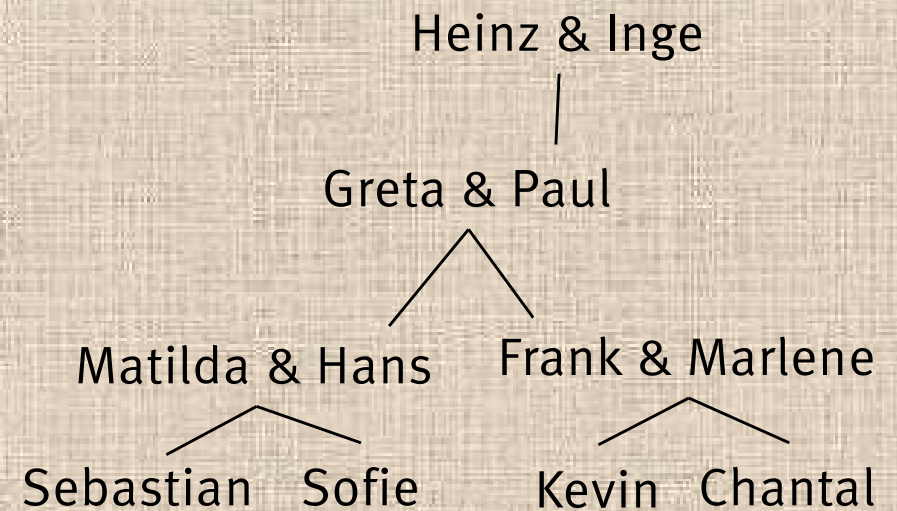
```
public class LongestString
{
    @UserAggregationFunction
    @Description( "org.neo4j.function.example.longestString(string)"
    public LongStringAggregator longestString()
    {
        return new LongStringAggregator();
    }
}
```

JAVA

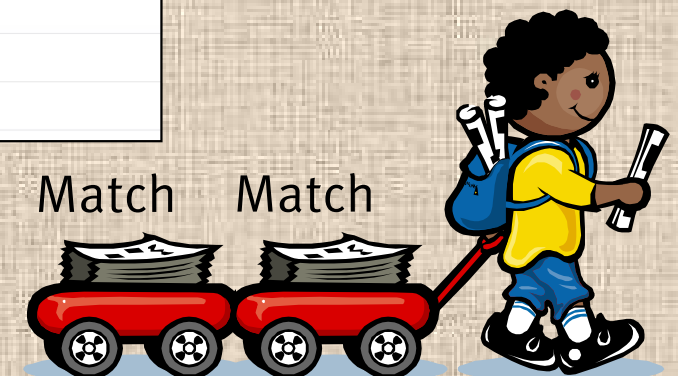
1	Datenbankmodell	2
2	Datenmodellierung	6
3	Graphendatenbank neo4j	18
4	Traversierung von Graphen	35
5	Vergleich mit anderen Datenmodellen	43

Finde alle Verwandten von Greta der
Kinder- und Enkelgeneration

```
MATCH
  (n:Person{vorname:'Greta'})-->(b)
WITH b
MATCH (b)-->(c)
RETURN b.vorname, c.vorname
```



b.vorname	c.vorname
"Frank"	"Chantal"
"Frank"	"Kevin"
"Marlene"	"Chantal"
"Marlene"	"Kevin"
"Matilda"	"Sofie"
"Matilda"	"Sebastian"
"Hans"	"Sofie"
"Hans"	"Sebastian"

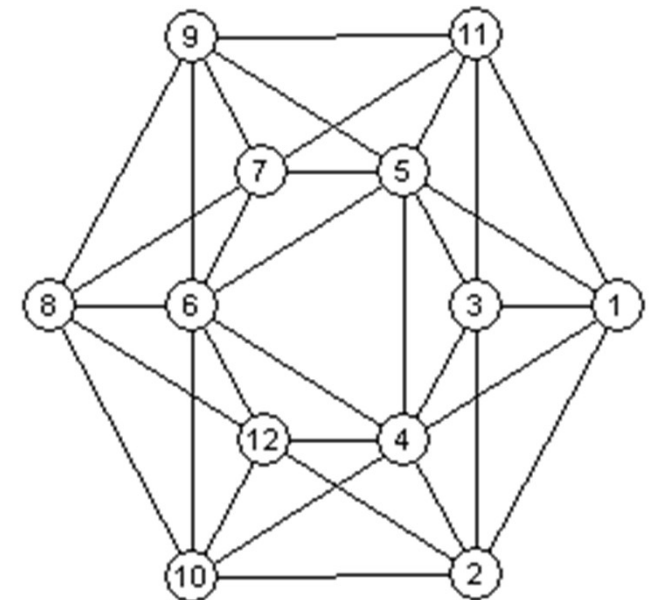


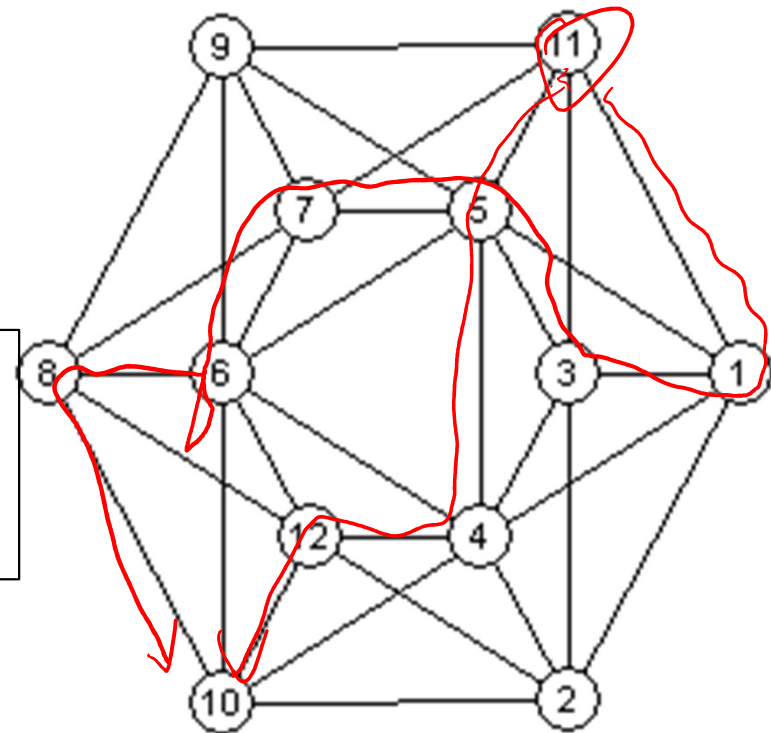
Bei der **Breitensuche** werden ausgehend vom Startknoten zunächst alle direkten Nachbarn bevorzugt besucht, bevor tiefer in den Graphen eingedrungen wird.

Bei der **Tiefensuche** wird ein Pfad bevorzugt bis zum Ende des Graphen verfolgt, bevor weitere Nachbarn des Startknotens besucht werden.

Anm.:

- Ein Knoten wird nur dann besucht, wenn er nicht zuvor besucht wurde und auch erreichbar ist.
- Die **Breitensuche** eignet sich für Anfragen, bei denen die Antworten im lokalen Umfeld des Startknotens zu suchen sind.
- Implementierungen der **Tiefensuche** sind i.d.R. speichereffizienter und schneller, liefern oftmals aber nicht den kürzesten Pfad zurück.



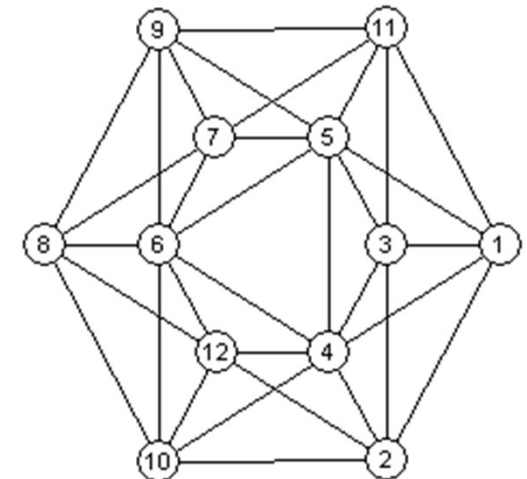


Welches ist der kürzeste Weg von Knoten A nach Knoten B?
Oder: Wer kennt wen?

■ Algorithmus von Dijkstra

1. Füge den Startknoten zur Menge der Knoten M hinzu, deren kürzester Pfad bekannt ist.
2. Ausgehend vom Startknoten finde die nächsten Nachbarn und bestimme die zugehörige Pfadlänge.
3. Ermittle die kürzesten Pfade zu den nächsten Nachbarn und übernehme diese Knoten in die Menge M.
4. Ausgehend von den Knoten in der Menge M ermittle die Pfadlängen zu den zugehörigen nächsten Nachbarn. Nachbarknoten, die bereits in der Menge M sind, brauchen nicht berücksichtigt werden.
5. Wiederhole die Schritte 3+4 bis der Zielknoten in der Menge M ist.

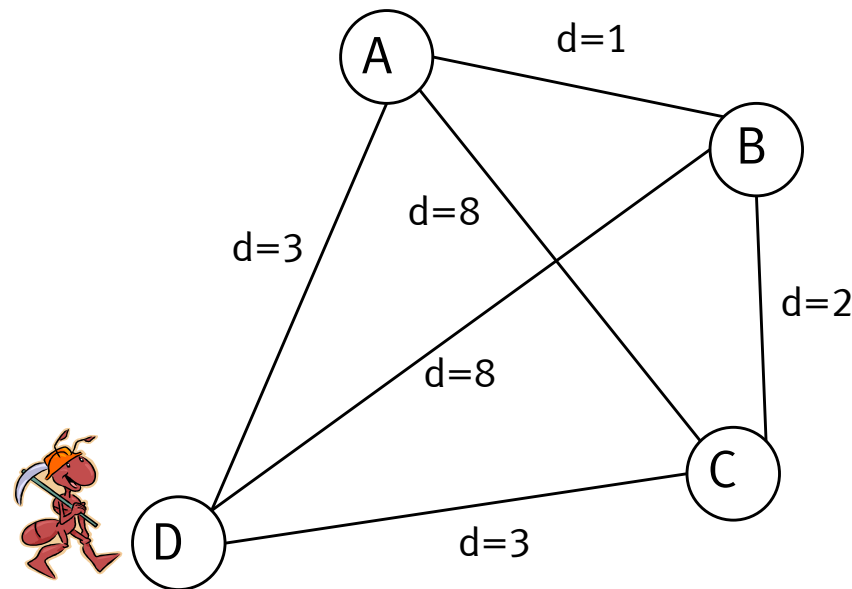
```
MATCH first = node:user(name='A'),  
      second = node:user(name='B')  
WITH first, second  
MATCH p=shortestPath(first-[0..*]-second)  
RETURN length(p) AS depth
```



Hamiltonweg:

Besuche jeden Knoten des Graphen exakt einmal und versuche dabei, die Kosten zu minimieren.

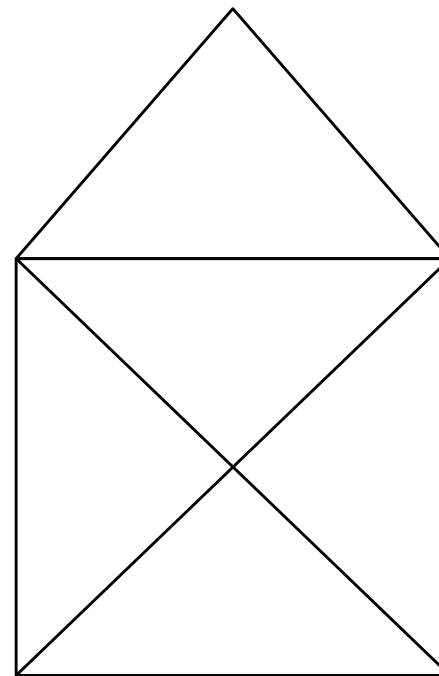
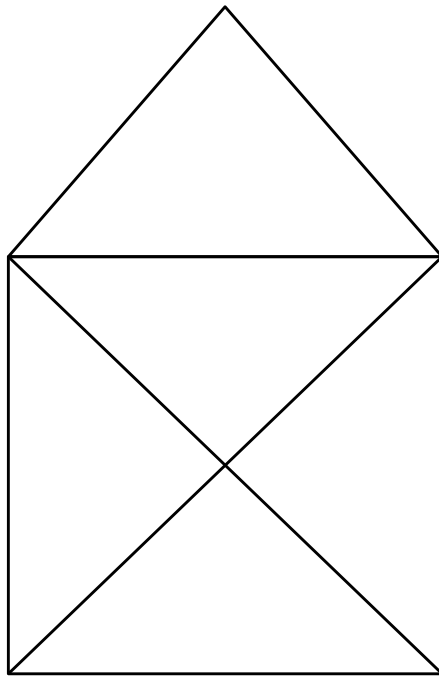
Ein **Hamiltonkreis** ist ein Hamiltonweg, bei dem der Start- und Zielknoten identisch sind. (Problem des Handlungsreisenden)



Lösungen können bspw. durch Schwarmalgorithmen gefunden werden.
Gremlin ist eine Skriptsprache zum Traversieren von Graphen.
Neo4j bietet ein entsprechendes Gremlin-Plugin an.
Beispiele zur Verwendung von Gremlin finden sich bspw. in [5].

Eulerweg:

Besuche jede Kante des Graphen exakt einmal und versuche dabei, die Kosten zu minimieren.



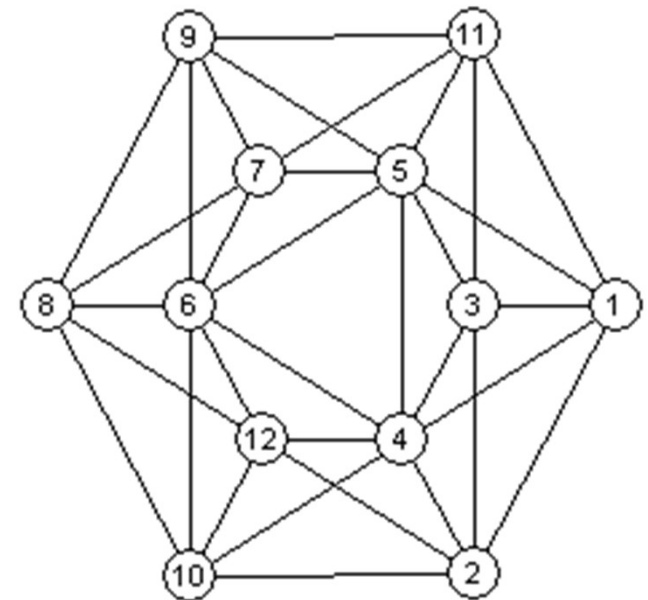
Randomisierte Traversals:

Für sehr große Graphdatensätze ist eine vollständige Traversierung des Graphen nicht mehr sinnvoll. Hier helfen randomisierte Ansätze, die schnelle Antworten liefern, die jedoch eine Fehlerwahrscheinlichkeit aufweisen.

Anwendung bspw. bei Empfehlungssysteme.

Beispiel: Liefere etwa 20% repräsentative Freundesfreunde einer Person zurück.

Lösung: „Random Walk“
Wähle per Zufallsgenerator
20% der Freunde aus und liefere
deren Freundesliste zurück.



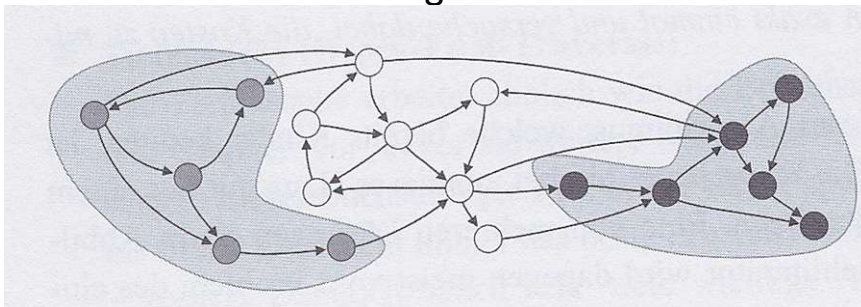
1	Datenbankmodell	2
2	Datenmodellierung	6
3	Graphendatenbank neo4j	18
4	Traversierung von Graphen	35
5	Vergleich mit anderen Datenmodellen	43

- Stärken von neo4j
 - Typ- und Schemafrei und kann daher unstrukturierte Daten verwalten.
 - Verbindungen sind persistiert, daher ist die Suche nach nächsten Nachbarn („Freunde“) performanter als in einer alternativen relationalen Lösung
- Performancevergleich:

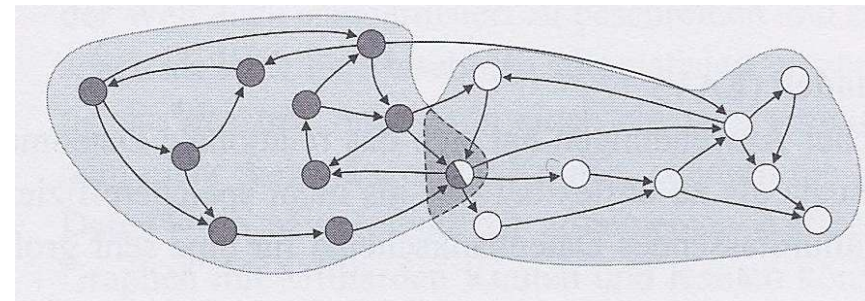
Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

■ Schwächen von neo4j

- Selbstverweise (Kante von einem Knoten auf sich selbst) sind nicht möglich.
- Es gibt keine Möglichkeit in Cypher die Durchführungsart der Graphtraversierungen zu beeinflussen. Daher sind bei großen Datenmengen Cypher-Anfragen langsam, insbesondere bei mehrfachen Filterbedingungen oder Zyklen. Abhilfe schafft hier eine Implementierung der Anfragen gegenüber der Java-API von neo4j (→Faktor 60-100 schneller!).
- Graph-Datenbanken sind nicht geeignet, um alle oder eine Untermenge der Knoten und Kanten zu ändern.
- Die Partitionierung von Graphen problematisch:
 - Es gibt keine exakte mathematische Methode, welche einen Graphen effizient in zwei oder mehrere gleichgroße Teilgraphen zerlegen kann. Angewendet werden können lediglich Heuristiken, wie das mehrfache Anwenden von Clustering-Algorithmen.
 - Praktische Anwendungen zeigen, dass 20% der Knoten zu etwa 80% angefragt werden. Daher ist zusätzliches Domänenwissen über typische Anfragen erforderlich, um eine geeignete Partitionierung zu finden.



Graph besteht aus 3 Teilgraphen



Zusammenhängender Graph über einzelne Knoten
(häufig in sozialen Netzwerken) Abb. Entnommen aus [1]

	Relational	Objektorientiert	Graph
Festlegung der Datenstruktur	Vor dem Einfügen der Daten	Vor dem Einfügen der Daten	Mit dem Einfügen der Daten
Vernetzung	Fremdschlüssel	Assoziation und Referenzen, ohne Kantentypen	Kanten mit unterschiedlichen Kantentypen
Dynamische Änderung des Schemas	Aufwändig	Aufwändig; abhängig von der Programmiersprache	Während des laufenden Betriebs
Graph-traversierung	Komplex (rekursive Joins !) z.T. in die Anwendungsebene verlagert	Anwendungsebene	Cypher und Gremlin

Reflexion Graph-Datenbanken

- 1) Was wird unter einer NoSQL-Datenbank verstanden?
- 2) Welche typischen Anwendungsfälle gibt es für Graph-Datenbanken?
- 3) Wie werden in neo4j Daten eingefügt, geändert und abgefragt?
- 4) Welche Möglichkeiten gibt es in neo4j, um Graphen zu traversieren?
- 5) Welche Vor- und Nachteile bieten der Einsatz der Graph-Datenbank neo4j?

Vielen Dank für Ihre Aufmerksamkeit

- [1] S. Edlich, A. Friedland, J. Hampe, B. Brauer
NoSQL – Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken
Hanser Verlag 2010
- [2] Practical Neo4j, Gregory Jordan, Apress, 2014
- [3] Graph Databases, I. robinson, J. Webber & E. Eifrem, O'Reilly, 2013
- [4] NoSQL Distilled, P. Sadalage, M. Fowler, Addison Wesley, 2013
- [5] Sieben Wochen, sieben Datenbanken, E. Redmond & J. Wilson, O'Reilly, 2012