UNIVERSITÄT SIEGEN

Vorlesung Datenbanksysteme II

Graph-Datenbanken

Inhalt

- Grundlagen und Motivation
- LPG und Cypher
- RDF und SPARQL

UNIVERSITÄT SIEGEN

Grundlagen und Motivation

Hintergrund

- Seit den 80er Jahren ist das relationale Datenbankmodell das dominierende Back-End professioneller Informationssysteme.
- Innovationen im Bereich der Informationssysteme (Stichwort: **Big Data**) fokussieren sich bisher vor allem auf neue Client-Anwendungen und Front-Ends sowie die Middleware und IR-Dienste.

Big Data

Trends der vergangenen 10 Jahre:

- Massive Verfügbarkeit und permanente (Neu-)Verknüpfung von immer mehr (Nutzer-/Anwendungs-)Daten.
- Sprunghaft wachsender Umfang der Daten.
- Immer schnellere Änderungen der Daten.
- Stark variierende, irreguläre (d.h. nicht mehr präzise festzulegende) Struktur der Daten.
- Benötigte Operationen zur Informationsgewinnung gehen weit über die bisher üblichen (relationalen) Abfragen hinaus.

Graphen sind heute allgegenwärtig

- Graphen: Menge von **Knoten** (Subjekte/Objekte) und **Kanten** (Beziehungen zwischen Knoten) als natürliche Repräsentation von **„Beziehungsnetzwerken“** in der Realwelt.
- Knoten: Sehr große Mengen **atomarer Einheiten** aus der Realwelt mit größtenteils primitivwertigen Attributen.
- Kanten: Sehr viele, ebenfalls attribuierte, **n:m-Beziehungen** zwischen Objekten (Knoten).
- Informationsgewinn: Suche/Analyse von „versteckten“ Zusammenhängen (über **Pfade** beliebiger Tiefe).

Graphen als allgegenwärtiges Datenmodell

Social Graph
Menschen und ihre Bekanntschaften

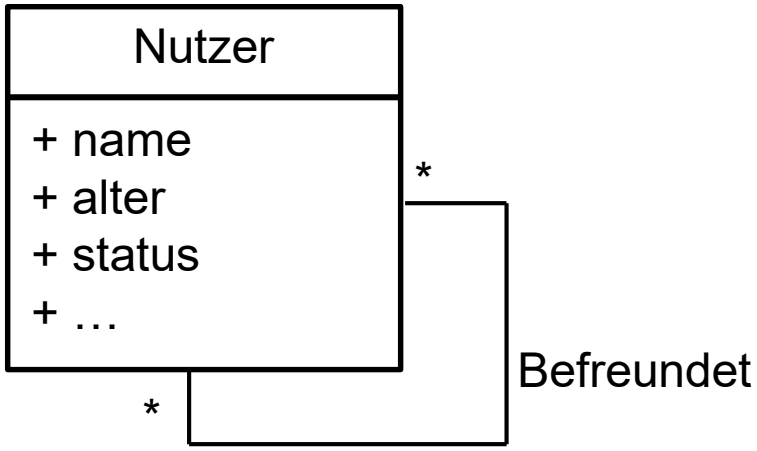
Consumtion Graph
Produkte und ihre Kunden

Web Graph
Webseiten und ihre Informationen

Repräsentation von Graphen in Relationalen DBS

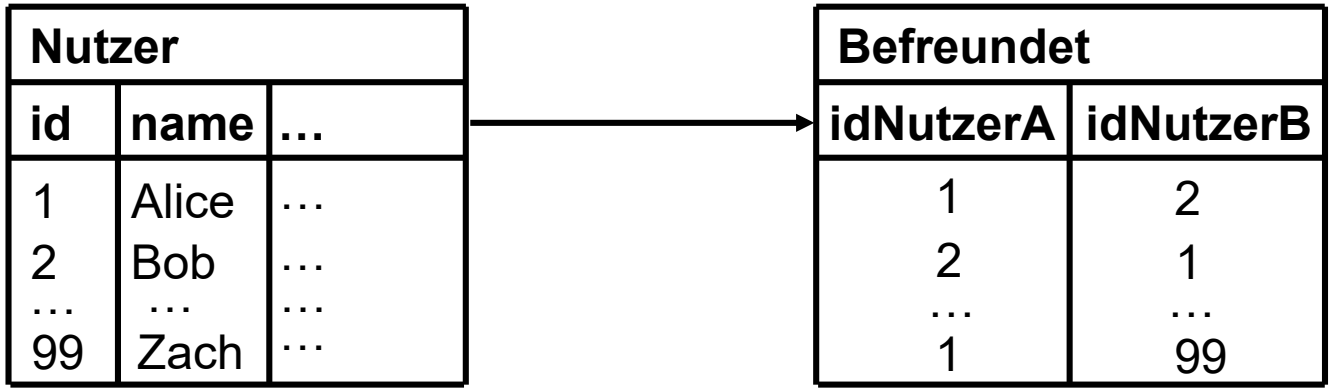
- Moderne RDBS eignen sich grundsätzlich auch zur effizienten **Repräsentation** sehr großer Graphen.
- Aber: **Operationen zur Informationsgewinnung** aus Graphen sind auf RDBS schwierig zu formulieren und zu implementieren bzw. ineffizient auszuwerten.
- Außerdem passt der Zwang zu einer festen Schema-Definition in RDBS nicht mehr zur **hohen Dynamik** Graph-basierter Datenmodelle und Graphdaten-Bestände.

Beispiel: Ein (sehr einfaches) soziales Netzwerk



- Für jeden **Nutzer** eines sozialen Netzwerkes sind zahlreiche **Eigenschaften (Attribute)** hinterlegt.
- Zwischen Nutzern können **Freundschaftsbeziehungen** bestehen.

Beispiel: Ein (sehr einfaches) soziales Netzwerk

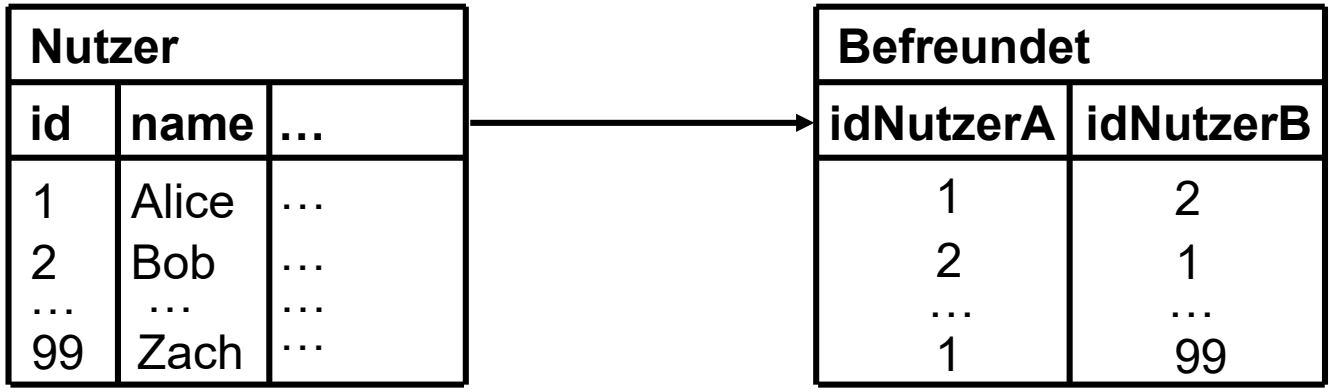


Nutzer		
id	**name**	**…**
1	Alice	…
2	Bob	…
…	…	…
99	Zach	…

Befreundet	
idNutzerA	**idNutzerB**
1	2
2	1
…	…
1	99

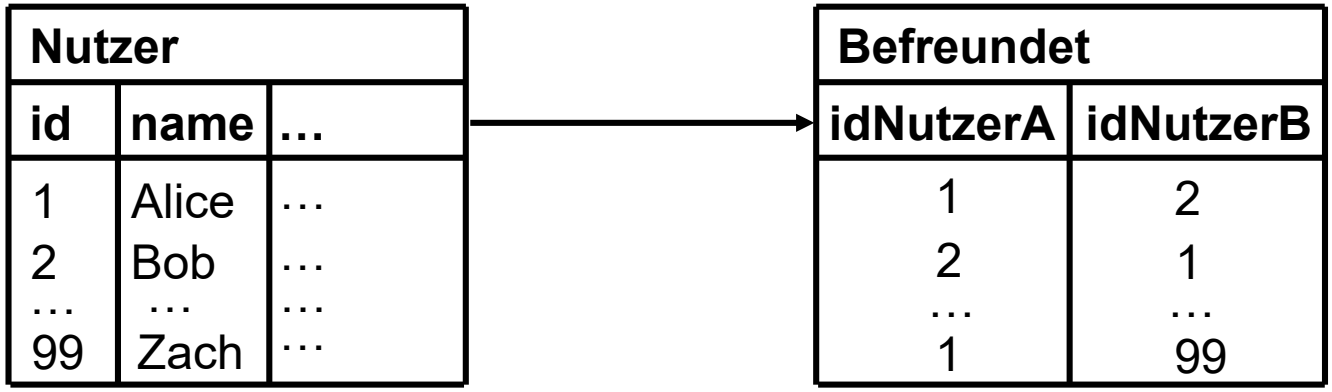
- Abbildung auf relationales Datenbankmodell durch eine Tabelle für Kundeneigenschaften und eine Tabelle für Freundschaftsbeziehungen.
- Anmerkung: Freundschaftsbeziehungen müssen nicht zwangsläufig symmetrisch sein.

Übungsaufgabe – Teil 1



Formulieren Sie eine SQL-Anfrage: *Mit wem ist Alice (direkt) befreundet?*

Lösung - Teil 1



SELECT n1.name

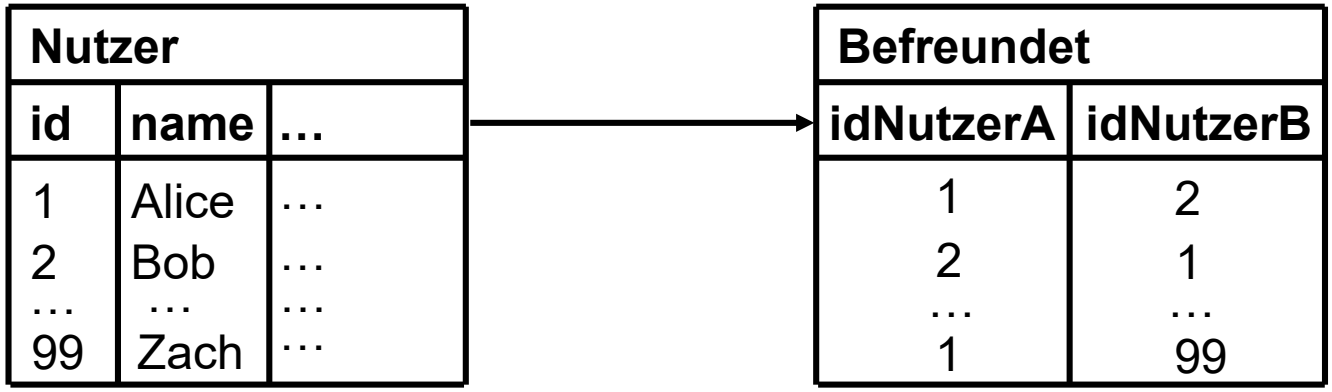
FROM Nutzer n1 **JOIN** Befreundet **ON**

Befreundet.idNutzerB = n1.id **JOIN** Nutzer n2 **ON**

Befreundet.idNutzerA = n2.id

WHERE n2.Person = ‘Alice‘

Übungsaufgabe – Teil 2

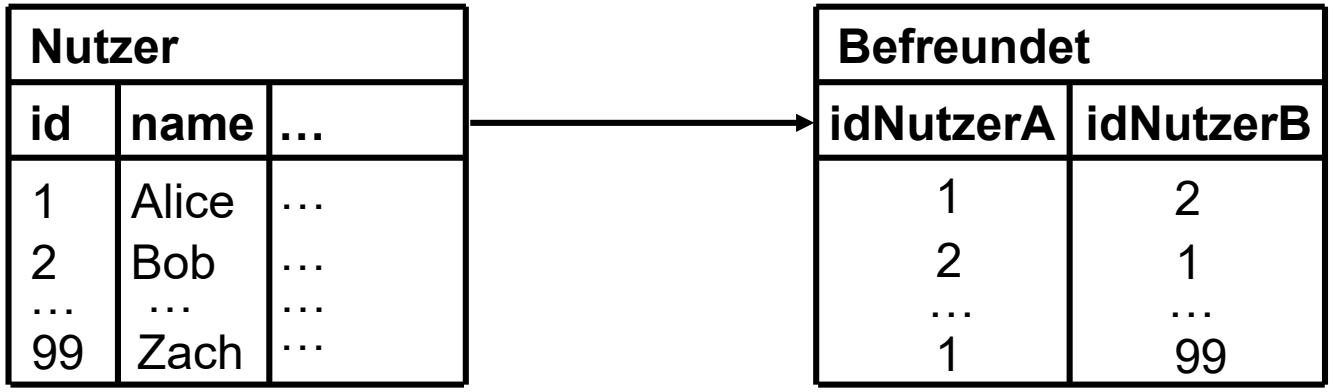


Nutzer		
id	**name**	**…**
1	Alice	…
2	Bob	…
…	…	…
99	Zach	…

Befreundet	
idNutzerA	**idNutzerB**
1	2
2	1
…	…
1	99

Formulieren Sie eine SQL-Anfrage: *Mit wem ist Bob über genau einen weiteren Freund (indirekt) befreundet?*

Lösung – Teil 2

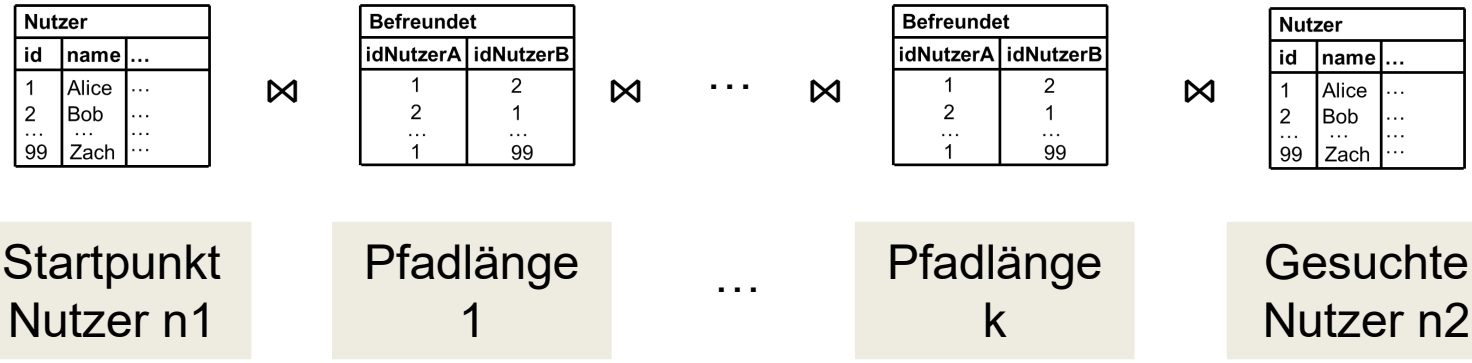


Nutzer		
id	**name**	**…**
1	Alice	…
2	Bob	…
…	…	…
99	Zach	…

Befreundet	
idNutzerA	**idNutzerB**
1	2
2	1
…	…
1	99

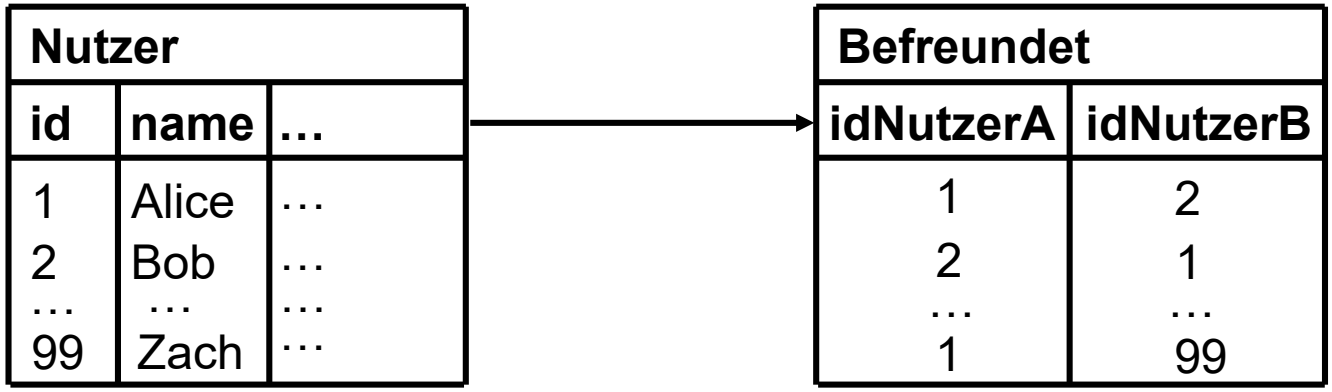
```
SELECT n1.name AS Nutzer, n2.name AS BefreundetÜberFreund
FROM   Befreundet bn1 JOIN Nutzer n1 ON
       bn1.idNutzerA = n1.id JOIN Befreundet bn2 ON
       bn2.idNutzerA = bn1.idNutzerB JOIN Nutzer n2 ON
       bn2.idNutzerB = n.id
WHERE n1.name = 'Bob' AND bn2.idNutzerB <> n1.id
```

Diskussion: Pfade fester Länge in RDB



- Suche nach Beziehungen über Pfade fester Länge *k* benötigen *k*-fachen JOIN der Beziehungstabelle.
- Für einen Datenbestand mit *n* Nutzern und *m* Freundschaftsbeziehung ergibt sich (ohne Verwendung von Indizes) ein Aufwand von $\boldsymbol{O}\left(n^k * m^k\right)$.
- Man spricht auch von sogenannten „Join-Bombs“.

Übungsaufgabe – Teil 3



Nutzer		
id	**name**	**…**
1	Alice	…
2	Bob	…
…	…	…
99	Zach	…

Befreundet	
idNutzerA	**idNutzerB**
1	2
2	1
…	…
1	99

Formulieren Sie eine SQL-Anfrage: *Mit wem ist Zach über beliebig viele weitere Freunde direkt oder indirekt befreundet?*

Lösung – Teil 3

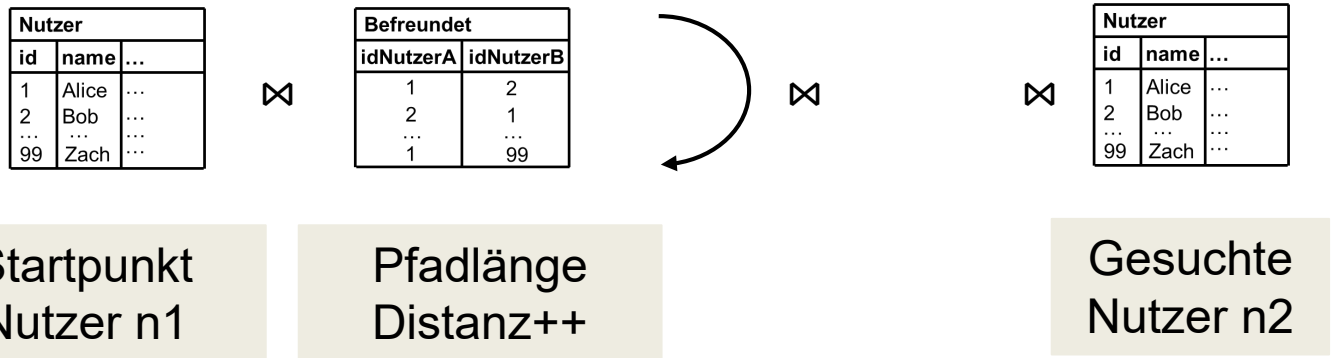
```
WITH RECURSIVE BefreundetClosure (Nutzer, Freund, Distanz) AS
( SELECT idNutzerA, idNutzerB, 1
  FROM    Befreundet
UNION
  SELECT bc.Nutzer, bn.idNutzerB, bc.Distanz+1
  FROM   BefreundetClosure bc, Befreundet bn
  WHERE bc.Freund = bn.idNutzerA )

SELECT n1.name
FROM    Nutzer n1 JOIN BefreundetClosure ON
          BefreundetClosure.idNutzerB = n1.id JOIN Nutzer n2 ON
          Befreundet.idNutzerA = n2.id
WHERE n2.Person = ‘Zach‘ AND n1.id ≠ n2.id
```

Diskussion: Pfade beliebiger Länge in RDB

- Seit SQL Version 1999 ist die Konstruktion **rekursiver Hilfsrelationen** möglich.
- Im Beispiel wird zunächst die vollständige **transitive Hülle** (Closure) der „Befreundet“-Relation berechnet und danach daraus die gesuchten Einträge für „Zach“ selektiert.
- Mögliche Optimierung: Festlegung des Start-Tupels bereits beim Rekursionsanfang (Gesamtaufwand bleibt dennoch hoch).

Diskussion: Pfade beliebiger Länge in RDB

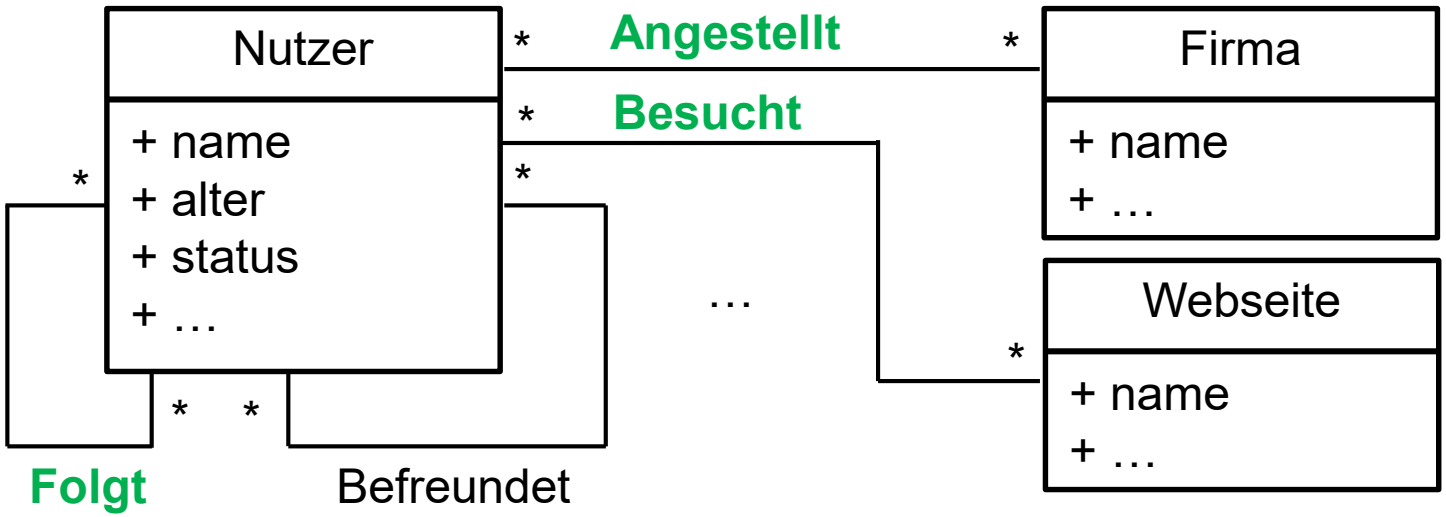


- Suche nach Beziehungen über Pfade beliebiger *Distanz*.
- Verbundbildung terminiert, wenn Ergebnistabelle keine neuen Beziehungen mehr enthält.

Diskussion: Pfade beliebiger Länge in RDB

- In der Praxis ist bei sehr großen Datenbeständen (mit Zyklen) eine explizite Beschränkungen der Rekursionstiefe notwendig (im Beispiel weggelassen: z.B. durch einen Maximalwerttest für *Distanz* im Rekursionsschritt).
- Weitere häufige Pfad-basierte Analyse-Probleme: Finden kürzester Pfade (z.B. kürzeste Freundschaftsverbindung zwischen Nutzern, …)

Beispiel: Änderung in RDB



- Vielzahl (ungeplanter) Updates von Datenbeständen um zusätzliche, bisher nicht im Datenschema berücksichtigte Informationen und Beziehungen.
- Updates betreffen häufig nur eine kleine Menge von Objekten, dennoch muss in RDB vor jeder Änderung zunächst das Schema angepasst werden.

Fazit: Repräsentation von Graphen in RDB

- Denken in Relationen-Schemata und Tabellenoperationen passt nicht mehr zu Graph-artig strukturierten Daten.
- Pfad-basierte Standard-Operationen auf Graphen führen zu „Join-Bombs“ in RDB.
- Zwang zu festen Schemata in RDB zu unflexibel für hohe Dynamik Graph-artiger Datenbestände.
- Graph-Datenbanken (GDB) sind ein Versuch, diesen Schwächen von RDB entgegenzuwirken.

Grundidee von Graph-Datenbanken

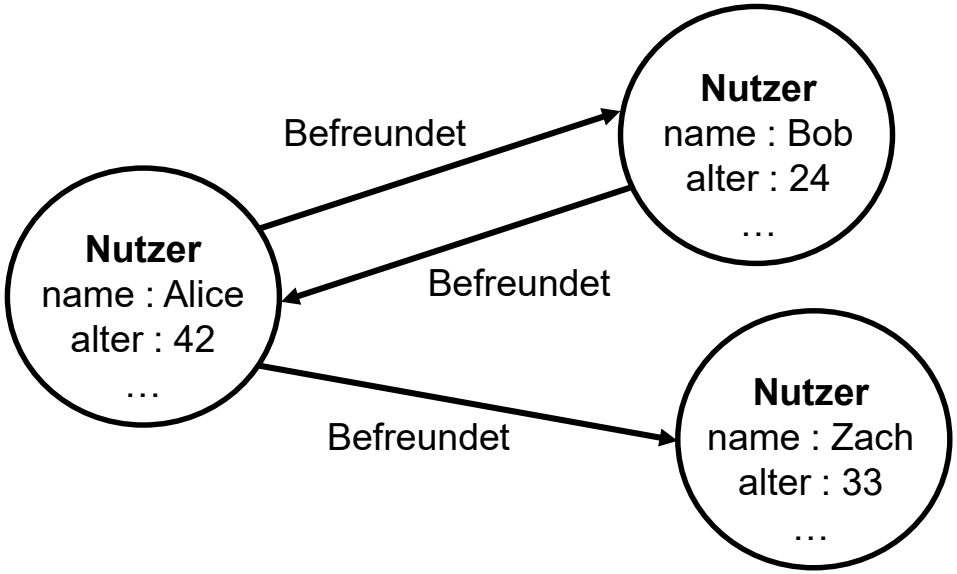
Graph-Datenbank (GDB) =

- Datenbank mit dezidiertem Datenmodell zur Speicherung von Graph-Datenstrukturen.
- Native Datenobjekte für **Knoten** (Nodes, Vertices), **Kanten** (Edges) zwischen Knoten, **Eigenschaften** (Properties) von Kanten und Knoten.
- CRUD-Operationen (Create, Read, Update, Delete) für einzelne Datenobjekte wie üblich.
- Anfragesprache mit **integrierten Graph-Operationen** (Graph-Muster-Suche, Pfad-basierte Suche, kürzeste Wege etc.)

Eigenschaften von Graph-Datenbanken

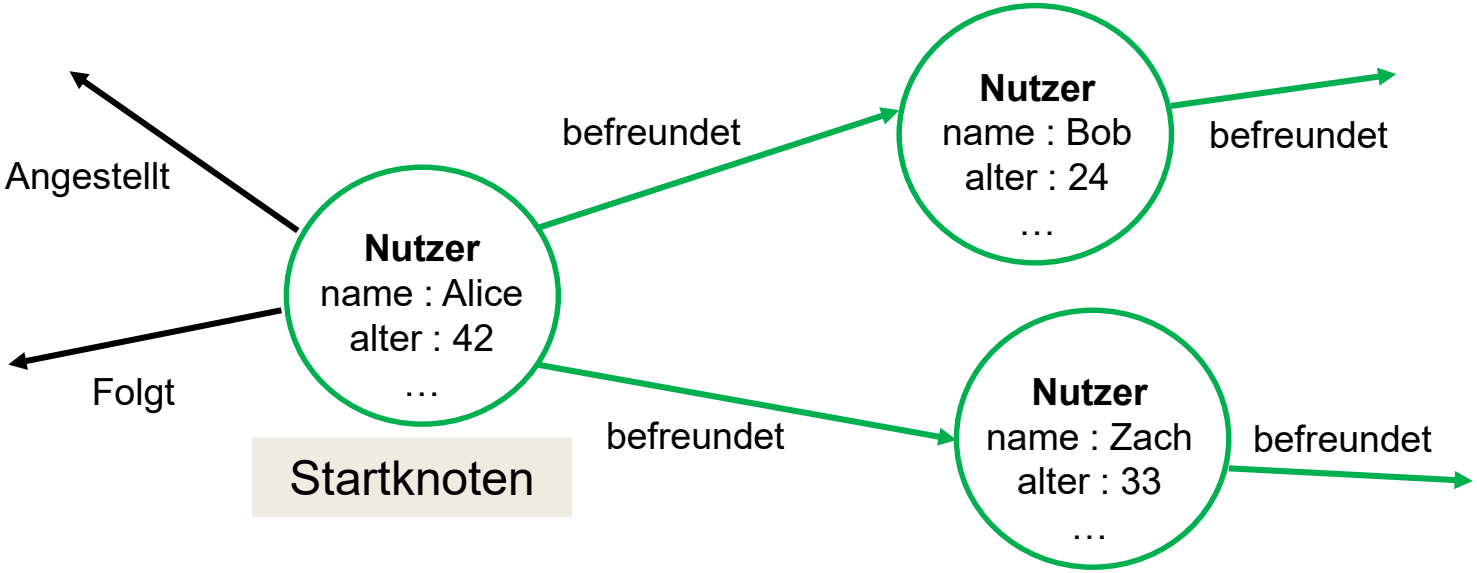
- Analog zu Relation/Tabellen in RDB, bilden in GDB Graphen die nativen Grundobjekte.
- Beziehungen nicht logisch/virtuell, sondern nativ/explizit als Kantenobjekte gespeichert.
- GDB beinhalten in der Regel keine Schema-Definitionen für „gültige“ Graph-Ausprägungen.

Beispiel: Ein (sehr einfaches) soziales Netzwerk



Anmerkung: Schlüsselattribut „id“ verzichtbar, da „befreundet“-Beziehungen nun durch explizite Kanten repräsentiert werden.

Beispiel: Pfade beliebiger Länge in GDB



- Ausgehend von einen **Startknoten** („Alice“), suche alle ausgehenden Kanten des gesuchten Typs.
- Wiederhole Suche rekursiv von allen über diese Kanten erreichten Knoten.

Anfragen auf Graphen

Weitere Beispiele für Anfragen auf Graphen:

- Liste von Knoten/Kanten, deren Attribute bestimmte Eigenschaften erfüllen.
- Liste von Knotenpaaren, die gegenseitig über beliebig lange Folgen von Kanten erreichbar sind.
- Kürzester Weg zwischen Knoten.
- Liste von Teil- oder Untergraphen, die bestimmte Muster aufweisen.
- …

Drei wichtige Grundprinzipien von GDB

1. **Knoten-Index:** Sehr schneller Zugriff auf Knoten über Knoten-Indizes.
2. **Index-freie Adjazenz:** Jeder Knoten „kennt“ seine direkten Nachbarknoten.
3. **Bidirektionale Navigation:** Navigation entlang gerichteter Kanten ist in beide Richtungen stets gleich schnell.

Anmerkungen

1. Ermöglicht schnelles Auffinden von Ausgangsknoten für die Auswertung Pfad-basierter Anfragen.
2. Ermöglicht das schnelle Auffinden von direkten Folgeknoten während der (rekursiven) Auswertung Pfad-basierter Anfragen.
3. Ermöglicht das schnelle Auffinden lokaler Pfadmuster unabhängig von der Kantenrichtung.

Diskussion: RDB vs. GDB

GDB:

- Beziehungen zwischen Objekten (Knoten) werden explizit als Kanten modelliert.
- Beziehungen zwischen Objekten werden als physikalische Verweise gespeichert.

RDB:

- Beziehungen zwischen Objekten (Relationen) werden implizit über (Fremd-)Schlüsselattribute kodiert.
- Beziehungen zwischen Objekten werden als logische Verweise simuliert.

Diskussion: RDB vs. GDB

GDB:

- Intuitive Formulierung und Auswertung Pfad-basierter Anfragen durch Navigationsoperationen auf Graphen.
- Auswertungskomplexität Pfad-basierter Anfragen bleibt konstant für wachsende Größe des Datenbestandes.

RDB

- Komplizierte Formulierung und Auswertung Pfad-basierter Anfragen durch (rekursive) Join-Operationen.
- Auswertungskomplexität Pfad-basierter Anfragen wächst im schlimmsten Fall exponentiell mit der Größe des Datenbestandes.

Diskussion: RDB vs. GDB

GDB:

- Anfragesprachen beinhalten keine join-Operationen.
- Kanten in GDB können als (statisch vordefinierte) Joins angesehen werden.

RDB:

- Datenbeschreibungssprachen beinhalten keine expliziten Beziehungskonstrukte.
- Joins in RDB können als (dynamisch konstruierte) Beziehungsgraphen angesehen werden.

Diskussion: RDB vs. GDB

GDB:

- Keine Schema-Definition, somit keine Gültigkeits-garantien.
- Erlaubt beliebig flexible Änderungen von Graphen auf allen Granularitätsebenen.

RDB:

- Feste statische Schema-Definition, garantiert Gültigkeit des Datenbestandes zu jedem Zeitpunkt.
- Datenstruktur muss von Beginn an umfassend feststehen, spätere Anpassungen aufwendig.

Diskussion: RDB vs. GDB

GDB:

- Kein allgemein akzeptierter Standard, stattdessen zwei parallel existierende Hauptvertreter (LPG/Cypher vs. RDF/SPARQL)

RDB:

- SQL (bzw. Dialekte) ist Quasi-Standard.

Graph Data Base Management Systems (GDBMS)

LPG

RDF

(Auswahl)

GDB: Grundlagen

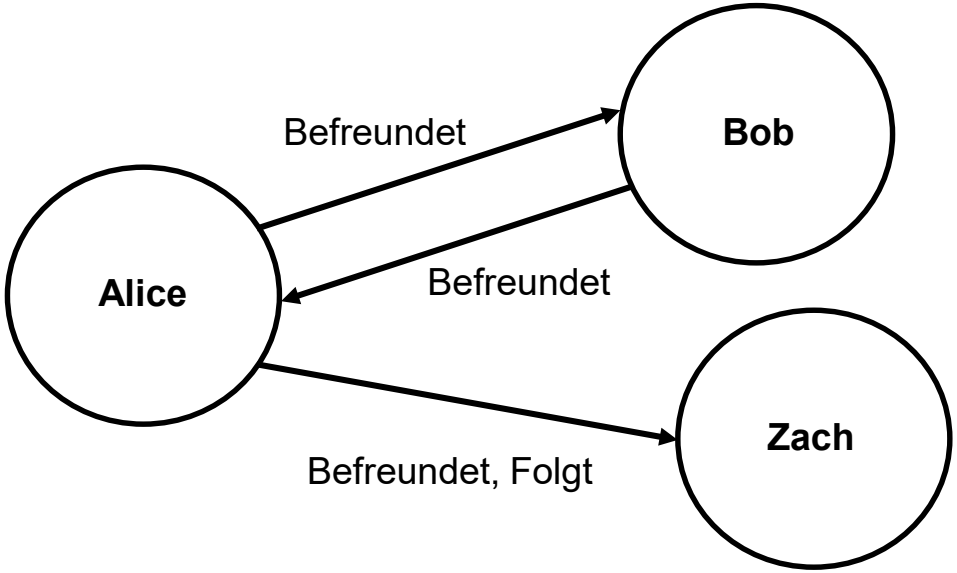
- Nachfolgend betrachten wir einige Grundlagen von GDB.
- Wir werden uns dabei aus Zeitgründen auf einige wesentliche Grundkonzepte fokussieren.

GDB: (Abstrakter) Graph-Datentyp

Ein Graph $G = (V, E, \Sigma, L)$ ist ein 4-Tupel, bestehend aus:

- V: endliche Menge (**Knoten**)
- $E \subseteq V \times V$: binäre Relation auf V (**Kanten**)
- Σ: endliche Menge (**Beschriftungen / Labels**)
- $L : V \times V \to 2^{\Sigma}$: Funktion (**Kantenbeschriftung**)

Beispiel: (Abstrakter) Graph-Datentyp



- $V = \{$ Alice, Bob, Zach $\}$
- $E = \{$ (Alice, Bob), (Bob, Alice), (Alice, Zach) $\}$
- $\Sigma = \{$ Befreundet, Folgt $\}$
- $L = \{$ (Alice, Bob) $\mapsto \{$ Befreundet $\}$, (Bob, Alice) $\mapsto \{$ Befreundet $\}$,
  (Alice, Zach) $\mapsto \{$ Befreundet, Folgt $\}\}$

Anmerkungen: (Abstrakter) Graph-Datentyp

- Diese einfache Definition entspricht einem **gerichteten, beschrifteten Multi-Graphen** (d.h. mehrere Beschriftungen pro Kante möglich).
- Beinhaltet (noch) keine Datenattribute für Knoten und Kanten (siehe später).

GDB: (Abstrakte) Basis-Operationen

- AddNode(G,x): Füge Knoten x in Graph G ein.
- DeleteNode(G,x): Lösche Knoten x aus Graph G.
- Adjacent(G,x,y): Gibt es in Graph G eine Kante von Knoten x nach Knoten y?
- Neighbors(G,x): Menge der Knoten in Graph G, zu denen es von Knoten x eine Kante gibt.
- AdjacentEdges(G,x,y): Menge der Beschriftugnen von Kanten, die in Graph G von Knoten x nach Knoten y führen.
- Add(G,x,y,l): Füge Kante von Knoten x nach Knoten y mit Beschriftung l in Graph G ein.
- Delete(G,x,y,l): Lösche Kante von Knoten x nach Knoten y mit Beschriftung l aus Graph G.
- Reach(G,x,y): Gibt es in Graph G einen Pfad von Knoten x nach Knoten y?
- Path(G,x,y): Einen (kürzesten) Pfad von Knoten x nach Knoten y.
- 2-hop(G,x): Menge der Knoten y, für die es in Graph G einen Pfad der Länge n von Knoten x nach y gibt oder umgekehrt.

LPG und Cypher

Neo4J

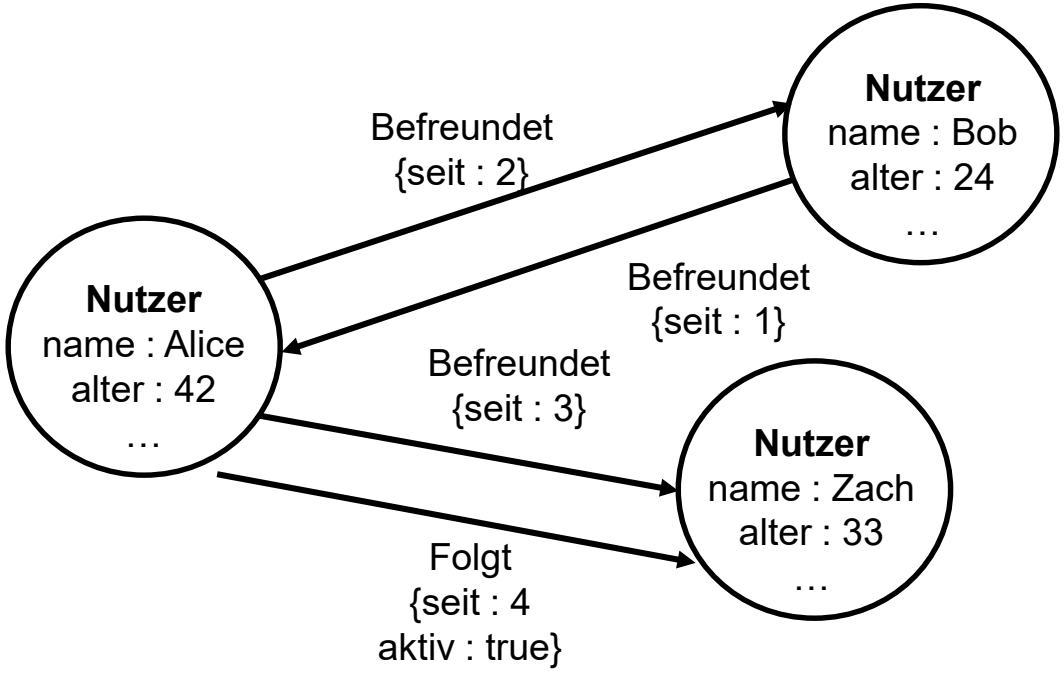
- Kommerziell / Open Source, sehr weit verbreitet (https://neo4j.com).
- Implementiert in Java.
- Erstes Release: 2007.
- Massiv skalierbar (auf Graphen mit Milliarden von Knoten).
- Transaktions-Engine, optimiert für Einzelmaschinen-Verarbeitung.
- Graph-Datenmodell: Labeled Property Graphs (LPG)
- Anfragesprache: Cypher

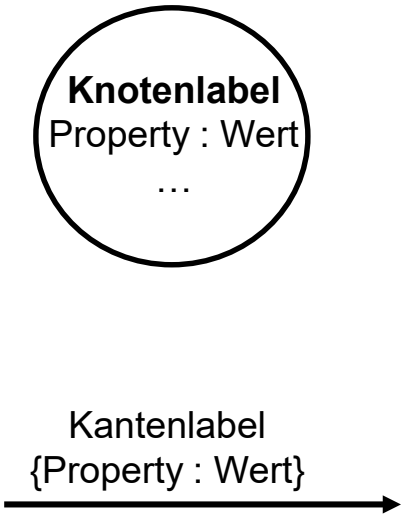
Labeled Property Graph (LPG)

Ein LPG $G = (V, E, \mathrm{s}, \mathrm{t}, \Sigma, L_V, L_E, P)$ ist ein Tupel, bestehend aus:

- V: endliche Menge (**Knoten**)
- E: endliche Menge (**Kanten**)
- $s: E \to V$: Funktion (**Startknoten** einer Kante)
- $t: E \to V$: Funktion (**Zielknoten** einer Kante)
- Σ: endliche Menge (**Beschriftungen / Labels**)
- $L_V: V \to 2^{\Sigma}$: Funktion (**Knotenbeschriftung**)
- $L_E: E \to \Sigma$: Funktion (**Knotenbeschriftung**)
- $P: V \cup E \to 2^{Prop}$: Funktion (**Eigenschaften / Properties**)

Beispiel: LPG





Beispiel

Notation

Anmerkungen. LPG

- Gerichteter, beschrifteter Multi-Graph.
- **Mehrere Kanten** zwischen dem gleichen Knotenpaar möglich.
- Kanten mit gleichem Start- und Endknoten möglich (Schlaufen, Lassos, …)
- Knoten haben feste und eindeutige (interne) **Identität** (u.a. für Indexbildung).
- Einem Knoten können **beliebig viele Labels** zugewiesen werden, einer Kante kann **genau ein Label** zugewiesen werden.
- Knoten und Kanten können **beliebig viele Properties** zugewiesen werden.

Properties

Eine Property ist ein **<key,value>-Paar**

- **key** ist ein String.
- **value** ist ein atomarer Wert eines primitiven Datentyps (vgl. z.B. Java) oder ein Array über einem solchen primitiven Datentyp.
- (Wir verzichten hier auf eine präzise Definition der Domäne $Prop$)

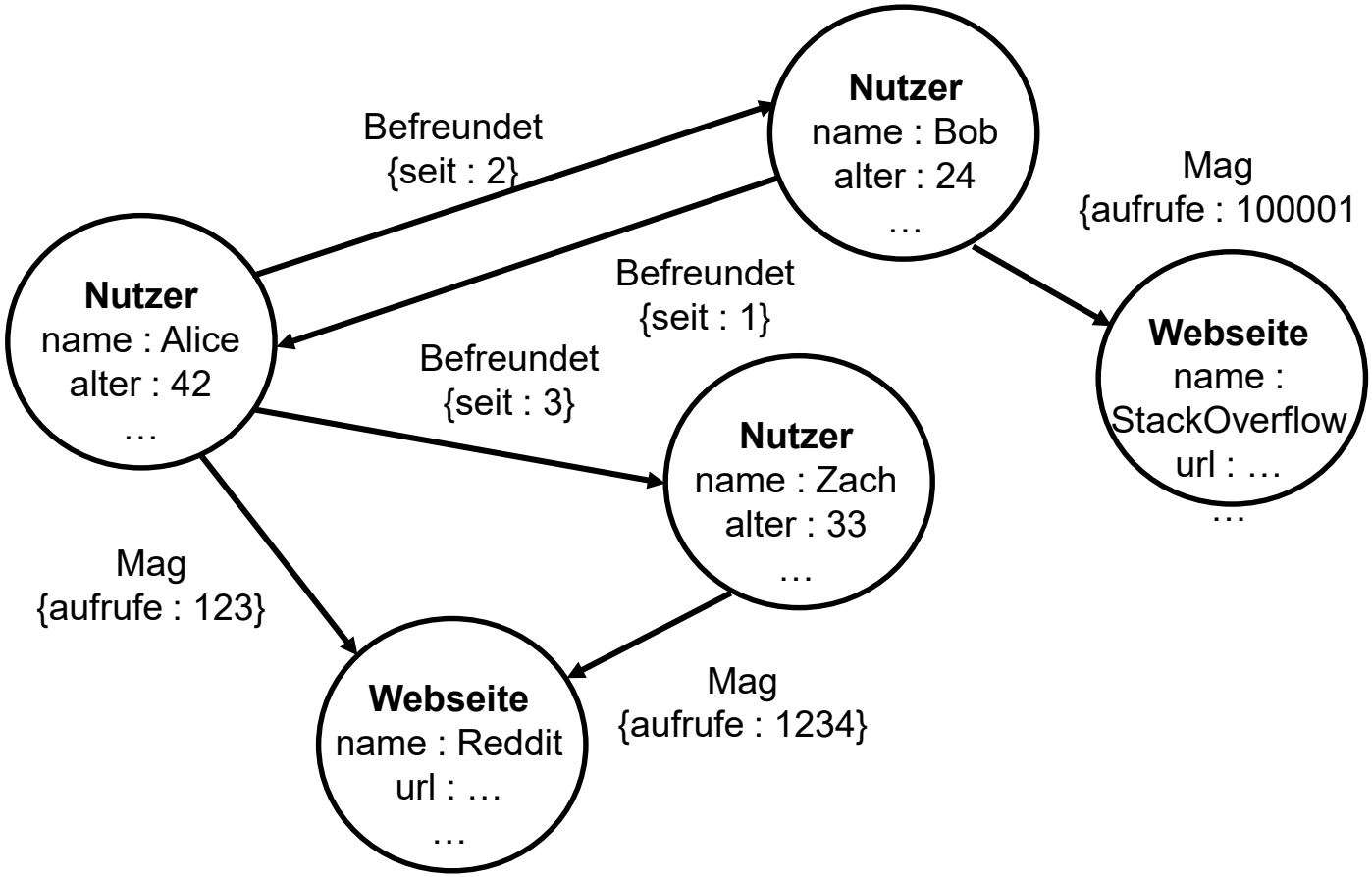
Diskussion: LPG

- Labels repräsentieren (eine Art von) **Typdeklaration** für Knoten und Katen.
- Properties repräsentieren (primitive) **Attribute** von Knoten und Kanten.
- Aber Vorsicht: Aufgrund der Abwesenheit einer Schema-Definition sagt das Label nichts über die Properties von Knoten und Kanten aus (z.B. können zwei Knoten mit dem gleichen Label völlig unterschiedliche Property-Mengen und ausgehende Kante aufweisen).

Cypher

- Deklarative Anfrage-Sprache für GDB, basierend auf SQL.
- Anfrageauswertung immer auf einem **einzelnen LPG** (keine Verbundbildung).
- Hauptoperationen sind Kombinationen aus Pfad-Navigationen (**Traversals**) und das Auffinden von Graph-Mustern (**Matches**) im Eingabe-Graphen.
- Ergebnis einer Anfrage ist im Allgemeinen wieder ein LPG (Teilmenge der Knoten/Kanten-Menge, Subgraph, neu konstruierter Ergebnis-Graph…).

Beispiel: Anfragen auf LPG



Beispiel: Anfragen in Cypher

1. Anfrage: Welche Nutzer mit Namen „Charlie“ sind Älter als 27?

```
MATCH (n1: Nutzer {name: “Charlie“})
WHERE n1.alter > 27
RETURN n1
```

Erläuterungen

- **MATCH**-Klausel: Deklaration einer Knoten-Variablen „n1“ für die Iteration über alle Knoten mit dem Label “Kunden“ und dem Wert “Charlie“ für die Property “name“.
- **WHERE**-Klausel zur weiteren Einschränkung der Knoten-Menge (hier: Property “alter“).
- **RETURN**-Klausel zur Definition des Ergebnis-LPG (hier: Knotenmenge aus dem Eingabe-LPG).

Beispiel: Anfragen in Cypher

2. Anfrage: Mit welchen Nutzern ist Charlie (direkt) befreundet?

```
MATCH (n1: Nutzer {name: “Charlie“}) -[:Befreundet]->(freund: Nutzer)
RETURN freund
```

Erläuterung

- Suche nach allen Knoten mit dem Label „Nutzer“, die vom Knoten „Charlie“ über eine Kante mit dem Label „Befreundet“ verbunden sind.
- Ausgabe aller gefundenen Knoten.

Beispiel: Anfragen in Cypher

3. Anfrage: Mit welchen Nutzern ist Charlie über genau einen anderen Freund befreundet?

```
MATCH (n1: Nutzer {name: “Charlie“}) -[:Befreundet * 2]-> (fvf: Nutzer)
RETURN fvf
```

Erläuterungen

- Explizite Angabe der Anzahl “Hops“ (hier: 2), über andere Nutzer über „Befreundet“-Kanten vom Knoten „Charlie“ aus erreicht werden sollen.

Beispiel: Anfragen in Cypher

4. Anfrage: Mit welchen Nutzern ist Charlie über beliebig viele andere Freund befreundet?

```
MATCH (n1: Nutzer {name: “Charlie“}) -[:Befreundet*]-> (fvf: Nutzer)
RETURN fvf
```

Beispiel: Anfragen in Cypher

5. Anfrage: Welche Freunde von Charlie mögen die Webseite „Reddit“?

```
MATCH (n1: Nutzer {name: “Charlie“}) -[:Befreundet]-> (freund: Nutzer)
        -[:Mag]-> (:Webseite {name : “Reddit“})
RETURN freund
```

Erläuterungen

- Einschränkende Graph-Muster können beliebig um den gesuchten Knoten „herum“ spezifiziert werden.

Beispiel: Anfragen in Cypher

5. Anfrage: Welches ist die kürzeste Verbindung zwischen Alice und Bob über beliebige Freundschaftsbeziehungen?

```
MATCH path = shortestPath
  ( (:Nutzer {name: “Alice“})-[:Befreundet*]->(:Nutzer {name: “Bob“})
RETURN path, length(path)
```

Erläuterungen

- Deklaration und Ausgabe von **Pfad-Variablen**.
- Zuweisung von Pfaden zur Pfad-Variablen durch **vordefinierte Graph-Operationen** (z.B. shortestPath, length) und weitere Einschränkungen durch Graph-Muster.

Beispiel: Graph-Updates mit Cypher

Erzeuge einen neuen Nutzer “Mike“.

```
CREATE (:Nutzer { name: “Mike“ })
```

Erläuterungen

- Da keine Schema-Definition existiert, werden neue Knoten ohne Gültigkeitsprüfungen stets „erfolgreich“ eingefügt.
- Zu beachten: Vor dem Löschen (**DELETE**) von Knoten müssen alle Kanten, die diesen Knoten als Start- oder Zielknoten haben, gelöscht werden.

Beispiel: Graph-Updates mit Cypher

Füge die Information hinzu, dass Bob neuerdings die Webseite „Reddit“ mag.

```
MATCH (n1: Nutzer {name: “Bob“}),(seite: Webseite {name: “Reddit“})
CREATE (n1) -[:Mag]-> (seite)
RETURN n1
```

Erläuterungen

- Anfragen und Updates können gemischt werden, die **RETURN**-Klausel ist in diesem Beispiel optional.
- **CREATE**-Klauseln können auch innerhalb der **RETURN**-Klausel verwendet werden, um „neue“ Ergebnis-Graphen zu konstruieren.

Beispiel: Indizes mit Cypher

Beobachtung: Auswahl von Nutzern erfolgt häufig über "name"-Property.

```
CREATE INDEX ON :Nutzer(name)
```

Übersicht: Cypher-Klauseln

- Read-Klauseln:
  - **MATCH-WHERE** (siehe Beispiele).
- Write-Klauseln:
  - **CREATE**: Erzeugen neuer Knoten und Kanten.
  - **DELETE**: Löschen bestehender Knoten und Kanten.
  - **SET**: Hinzufügen/Ändern von Labels und Properties (Ausnahme: Kanten-Label nicht änderbar).
  - **REMOVE**: Entfernen von Labels und Properties (Ausnahme: Kanten-Label nicht löschbar).

Übersicht: Cypher-Klauseln

- Allgemeine Klauseln:
  - **RETURN**: Spezifikation von Anfrage-Ergebnissen.
  - **ORDER BY**: Sortierungsvorschrift für das Anfrage-Ergebnis (z.B. anhand von Property-Werten).
  - **LIMIT**: Beschränkung der Anzahl von Ergebnissen.
  - **WITH**: Verknüpfung von Teil-Anfragen (z.B. mit logischen Konnektoren)
  - **DISTINCT**: Duplikateneliminierung.
  - *uvm*.

Pfad-Muster

Syntax inspiriert durch „ASCII-Art“:

- (…) für Knotenmuster
- -->, <-- für gerichtete Kantenmuster
- -- für ungerichtete Kantenmuster

Semantik:

- Ermittelt Menge aller passenden Pfade (keine Teilgraphen!)

Knotenmuster

Syntax:

- Knotenvariablen zum späteren Iterieren über Ergebnis-Knotenmenge.
- Label-Menge: Gefundene Knoten müssen **alle** Labels aufweisen.
- Property-Map: Gefundene Knoten müssen **alle** Property-Eigenschaften erfüllen.

Semantik:

- Ergebnis-Knotenmenge enthält alle Knoten, die die obigen Eigenschaften erfüllen.

Kantenmuster

Syntax:

- Kantenvariablen zum späteren Iterieren über Ergebnis-Kantenmenge.
- Label-Menge: Gefundene Kanten müssen **eines** der geforderten Label aufweisen.
- Property-Map: Gefundene Kanten müssen **alle** Property-Eigenschaften erfüllen.
- Muster für Pfadlänge: Kombination gefundener Kanten zu Pfaden mit minimaler/maximaler/beliebiger Anzahl Kanten (die alle obige Eigenschaften erfüllen).

Semantik:

- Ergebnis-Kantenmenge enthält alle Kanten, die die obigen Eigenschaften erfüllen.

Auswertung von Match-Klauseln

- Anfrage-Ergebnis ist eine Menge von **Teilgraphen**.
- Genauer: Jedes Ergebnis ist eine **Bindung aller Knoten- und Kantenvariablen** an konkrete Knoten und Kanten im Eingabe-Graphen, sodass sämtliche Bedingungen erfüllt sind.
- Durch Kombination mit CREATE-Klauseln könne aus Ergebnissen neue Ausgabe-Graphen konstruiert werden.

Eindeutigkeitsforderung

Für jedes Anfrage-Ergebnis gilt:

- Ein Knoten des Eingabegraphen kann potenziell an **beliebig viele Knotenvariablen** der Anfrage gebunden werden.
- Eine Kante des Eingabegraphen darf nur an **maximal eine Kantenvariable** der Anfrage gebunden werden.

Optionale Matches

Finde alle Nutzer, die jünger als 18 sind und gebe deren Freunde aus (falls es welche gibt).

```
MATCH (n1: Nutzer)
  WHERE (n1.alter < 18)
MATCH OPTIONAL (n1)-[:BEFREUNDET]->(n2:Nutzer)
RETURN n1.name, n2.name
```

Erläuterungen

- Für Nutzer, die jünger als 18 sind, aber keine Freunde haben, wird **ein** Ergebnis-Element ausgegeben, in dem die zweite Komponente mit einem **null**-Eintrag aufgefüllt wird.
- Konstrukt ist umstritten, da es in Kombination mit negierten/geschachtelten Teilausdrücken evtl. sehr ineffizient auszuwerten.

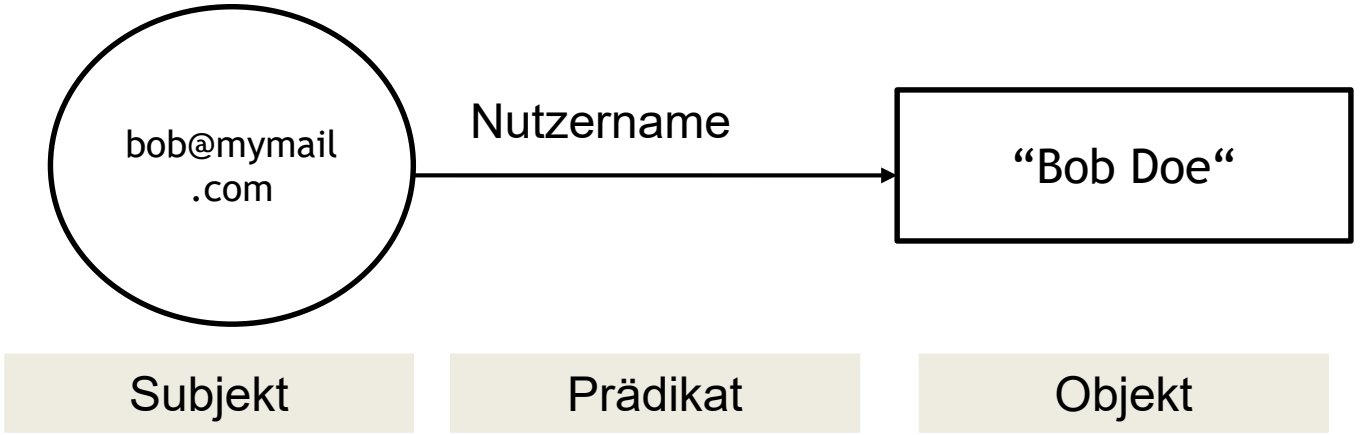
UNIVERSITÄT SIEGEN

RDF und SPARQL

Ressource Description Framework (RDF)

- RDF ist eine allgemeine Syntax zur Darstellung von Daten im Web.
- Jede in RDF ausgedrückte Information wird als ein **Triple** dargestellt:
  - **Subjekt:** Eine Quelle, die mit einer URI identifiziert werden kann.
  - **Prädikat:** eine URI-identifizierte wiederverwendete Spezifikation der Beziehung.
  - **Objekt:** eine Quelle oder Literale, mit dem das Thema verwandt ist.

Beispiel: RDF



- Graph = Sammlung von Tripeln, deren Subjekte/Objekte über Beziehungen miteinander verknüpft sind.
- Standardisierte RDF-Syntax im XML-Format.
- Siehe: http://www.w3.org/2009/12/rdf-ws/papers/ws11

SPARQL Protocol and RDF Query Language

- Standardisierte Query Language für RDF.
- Syntaktisch noch näher an SQL als Cypher.
- Umfang vergleichbar zu Cypher: (Optionale) Graph-Muster, Geschachtelte Anfragen, Negierte Muster, Aggregation, Navigation, Graph-Konstruktoren, …
- Siehe: https://www.w3.org/TR/sparql11-query/

Prüfungsstoff

- Motivation und Grundlagen von GDB erläutern können.
- Vergleich RDB und GDB anstellen können.
- Grundidee von LPG/Cypher verstanden haben.

Literatur

- Ian Robinson et al.: Graph Databases 2e (Englisch) Taschenbuch – 31. Juli 2015, ISBN-13: 978-1491930892