

# Projektabgabe

- ▶ Abgabe: 07.01.2016
  - Git Repo Master Branch
- ▶ Vorträge:
  - 08.01.2016 Freitag
  - 15.01.2016 Freitag
  - Zufällige Verteilung der Gruppen auf beide Tage

# Bewertung

- ▶ Funktionalität
- ▶ Einsatz der NoSQL Datenbank
- ▶ Verarbeitung der Request
- ▶ Abfragen der NoSQL Datenbank

# Up to Come ...

- ▶ Graphendatenbanken
- ▶ ExpressJS
- ▶ ElasticSearch
- ▶ AngularJS/ React

# NoSQL Datenbanken

Vorlesung - Hochschule Mannheim

## Klassifizierungen

# Inhaltsverzeichnis

- ▶ Key-Value Stores
- ▶ Dokumentenbasierte Datenbanken
- ▶ **Graphendatenbanken**

# Graphen Datenbanken



























TITAN  
Distributed  
Graph  
Database



# Weitere

- ▶ Giraph
- ▶ AllegroGraph
- ▶ IniniteGraph
- ▶ Stardog



Rank			DBMS	Database Model	Score		
Nov 2015	Oct 2015	Nov 2014			Nov 2015	Oct 2015	Nov 2014
1.	1.	1.	Neo4j 	Graph DBMS	34.04	+0.63	+9.39
2.	2.	2.	Titan	Graph DBMS	6.06	+0.54	+3.45
3.	3.	3.	OrientDB	Multi-model 	5.50	+0.57	+3.48
4.	4.	 7.	ArangoDB 	Multi-model 	1.61	+0.13	+1.30
5.	5.	 6.	Giraph	Graph DBMS	0.92	-0.00	+0.45
6.	6.	 5.	AllegroGraph 	Multi-model 	0.91	+0.01	+0.30
7.	7.	 11.	Stardog	Multi-model 	0.54	+0.02	+0.41
8.	 9.	 9.	Sqrrl	Multi-model 	0.42	-0.01	+0.24
9.	 8.	 8.	InfiniteGraph	Graph DBMS	0.38	-0.05	+0.12
10.	10.	 4.	Sparksee	Graph DBMS	0.29	-0.02	-0.59
11.	11.	 15.	HyperGraphDB	Graph DBMS	0.25	-0.01	+0.22
12.	12.	12.	InfoGrid	Graph DBMS	0.21	+0.00	+0.09
13.	 14.		VelocityGraph	Graph DBMS	0.19	+0.02	
14.	 15.	 16.	GlobalsDB	Multi-model 	0.18	+0.02	+0.18
15.	 13.	 13.	FlockDB	Graph DBMS	0.16	-0.02	+0.04

Quelle: <http://db-engines.com/en/ranking/graph+dbms>

# Yet another NoSQL database?

- ▶ Semantic Web
- ▶ Soziale Netzwerke / Empfehlungssysteme
- ▶ Komplexe Strukturen
- ▶ Stark vernetzte umfangreiche Datenmengen

# Vergleich

	Komplexität	Größe
Dokumentenbasierte DBs	Mittel	Mittel
Key-Value Store	Gering	Groß
GraphenDB	Sehr hoch	Mittel

# Die Welt ist ein Graph



# Neue Anforderungen

- ▶ Wer kennt wen?
- ▶ Finde alle Freunde die X geliked haben
- ▶ Wer hat mein Profil wie oft gesehen
- ▶ Wer hat den YouTube Channel abonniert
- ▶ Welche Kategorien werden geliked?

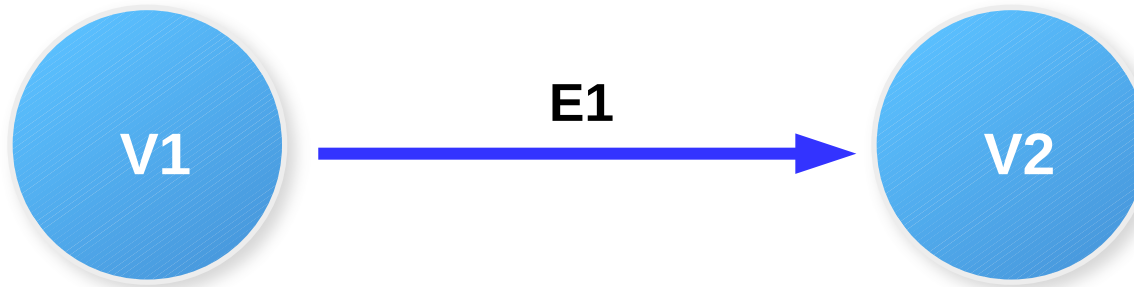
# Definition

- ▶ Basieren auf dem Graphenmodell
- ▶ Datensätze in Form von Knoten und Beziehungen
- ▶ Ermöglichen einfaches Errechnen über spezifische Eigenschaften

# Graphenmodell

Graph  $G(V, E)$

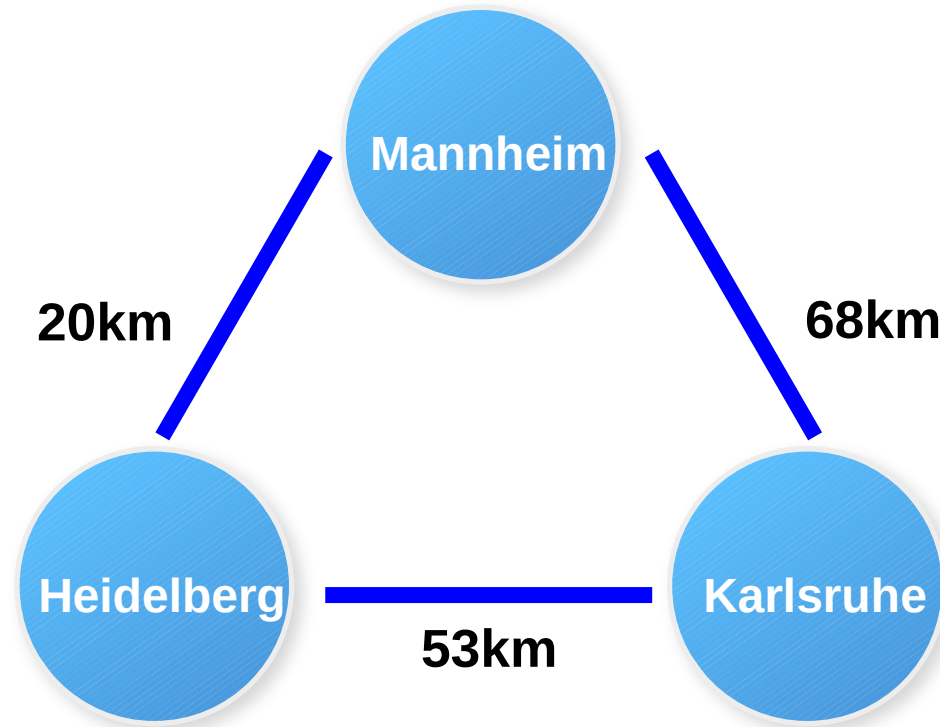
- Vertices: Menge von Knoten  $V$
- Edges: Menge von Kanten  $E$



Gerichteter Graph

# Graphenmodell

- ▶ Ungerichteter Graph
- ▶ Gewichtete Kanten

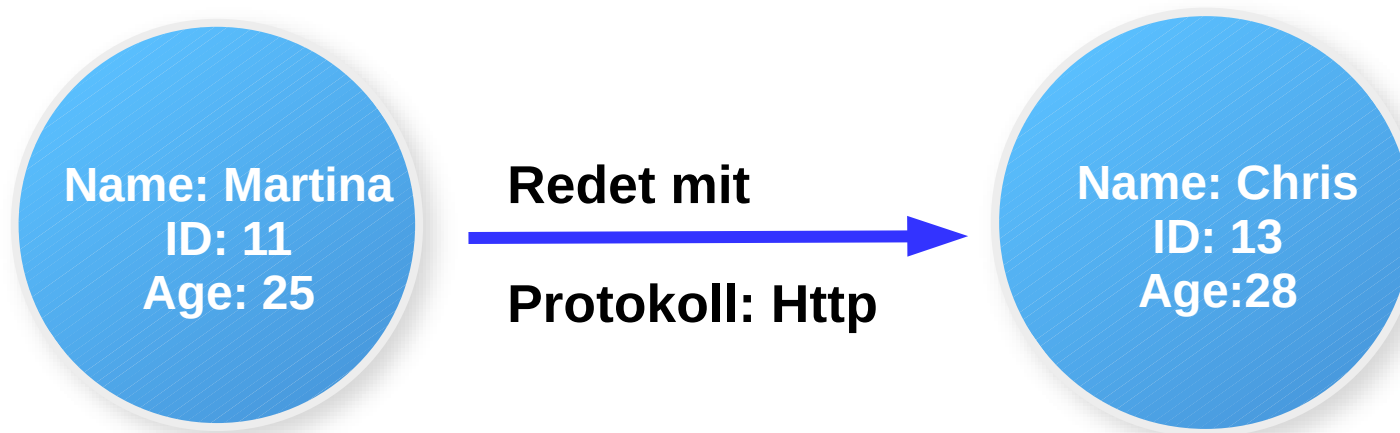




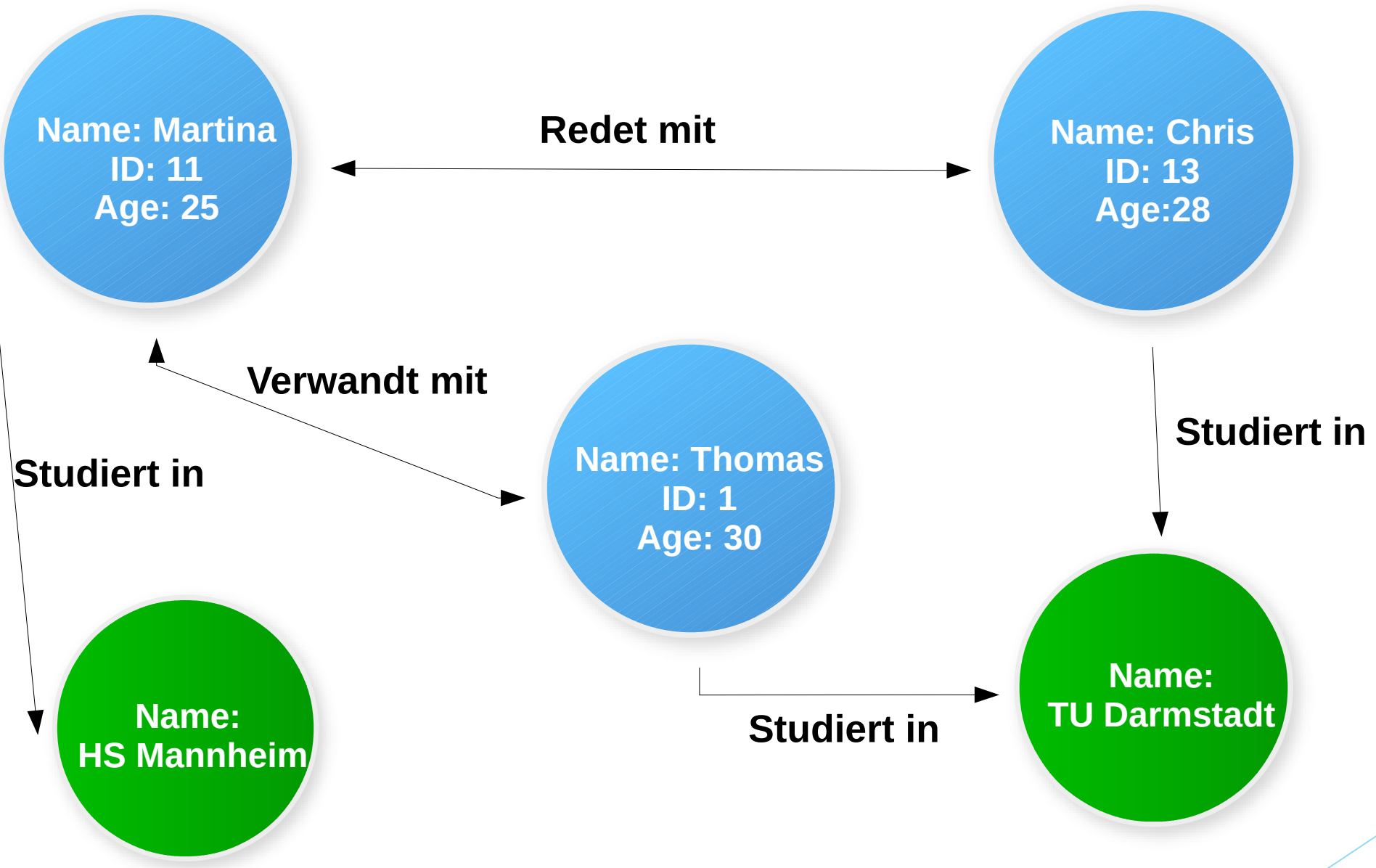
# Property Graphmodell

## Erweiterung

- ▶ Zusätzliche Eigenschaften an Knoten und Kanten
- ▶ Properties sind Key-Value Paare
- ▶ Keys werden vom Schema des Knotentyps vorgegeben



**Gerichteter/ Multirelationaler Graph**



# Terminologien

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	Node types
row	node
column	property
index	index
table joins	Traverse
.	

# Abgrenzung zu RDBMS - Property graph

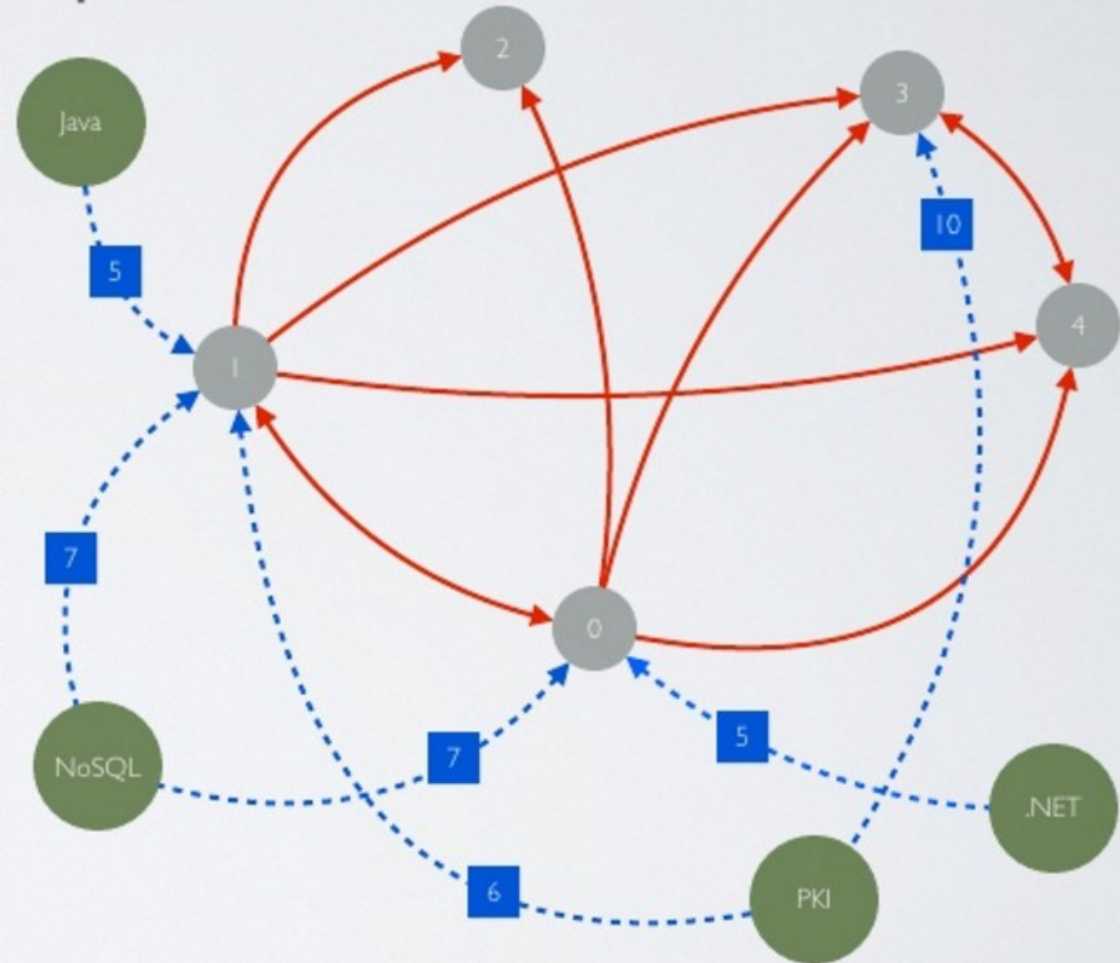
Person	
Id	Name
0	Henning Rauch
1	René Peinl
2	Foo Bar
3	Bruce Schneier
4	Linus Torwalds

Kennt_rel	
Id_1	Id_2
1	0
1	2
1	3
1	4
0	1
0	2
0	3
0	4
3	4
4	3

Tag	
Id	Name
0	.NET
1	Java
2	PKI
3	NoSQL

Tags_rel		
Tag_Id	Person_Id	Signifikanz
0	0	5
1	1	5
2	1	6
2	3	10
3	0	7
3	1	7

RDBMS | GraphDB



Knoten =  
Eigenschaften + Kanten

# Vorteile

- ▶ Direkte Abbildung realer Netzwerkstrukturen
- ▶ Keine Indizes für Relationen
  - Hohe Performance
- ▶ Traversierung
  - Breitensuche
  - Tiefensuche
  - Dijkstra
  - Suche nach Knoten mit bestimmten Eigenschaften

# Anwendung

- ▶ Wer kennt wen?
- ▶ Finde alle Freunde die X geliked haben
- ▶ Wer hat mein Profil wie oft gesehen
- ▶ Wer hat den YouTube Channel abonniert
- ▶ Welche Kategorien werden geliked?



**ArangoDB**

# ArangoDB

- ▶ 2011 gestartet unter dem Namen AvocadoDB
  - ▶ Initiales Release 2012
  - ▶ ArangoDB GmbH / triagens GmbH
  - ▶ Aktuelle Version 2.7.1, November 2015
  - ▶ Open Source
- 
- ▶ Geschrieben in C, C++ und Ruby
  - ▶ Treiber für C#, Java, JavaScript, PHP, Python, Ruby



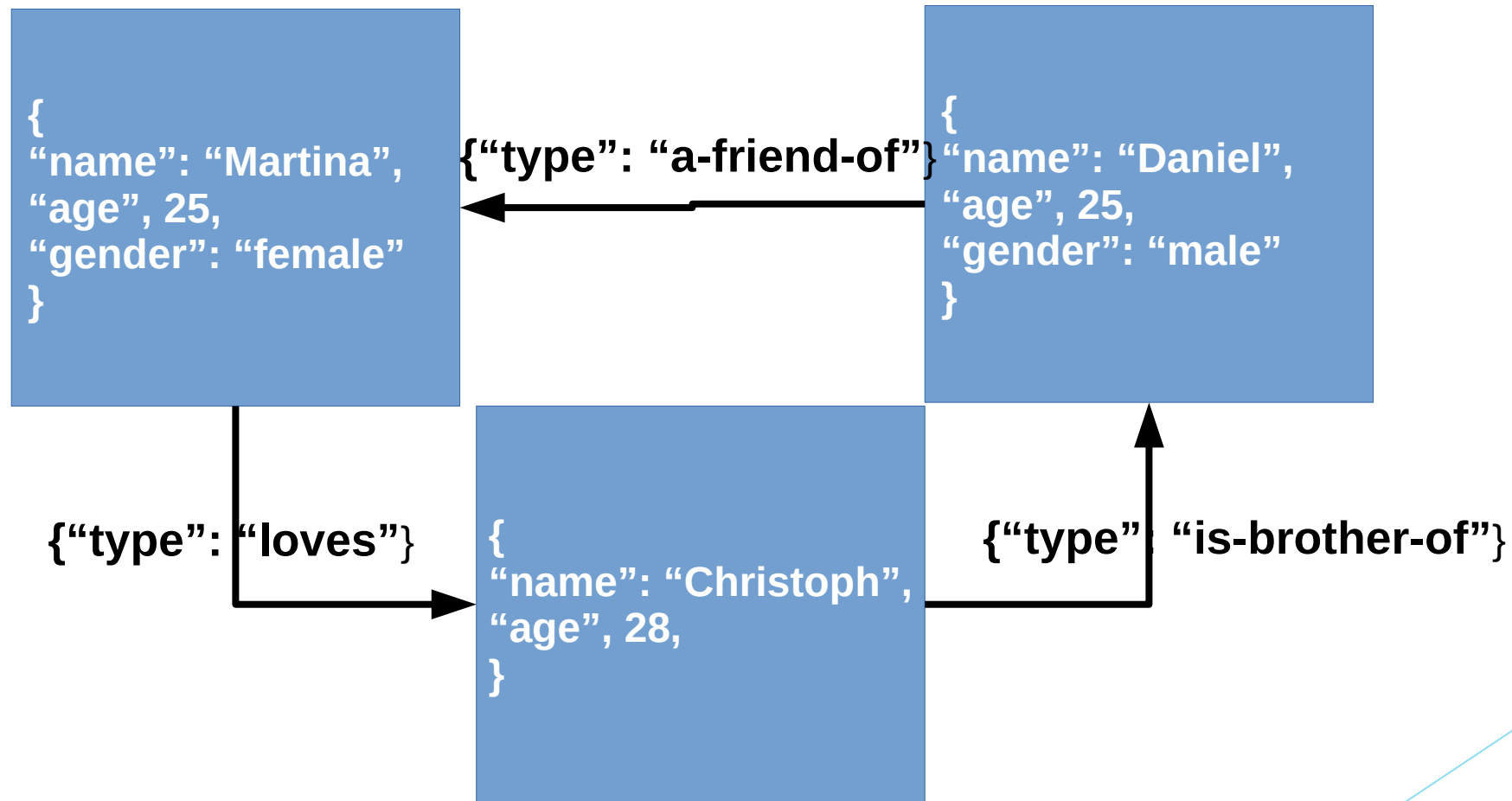
# ArangoDB

- ▶ Natives Multi-Model DBMS
  - Platz 4 für Graphen-Datenbanken
- ▶ Seit Version 2 Sharding Support
- ▶ Master-Slave / Master-Master Replication
- ▶ Eventual / Immediate Consistency (ACID)
  
- ▶ Abfragesprachen:
  - REST-Calls, API Calls
  - ArangoDB Query Language (AQL)

# Multi-Model DBMS

- ▶ “hybride” Datenbank
- ▶ Documents/ Collections everywhere
- ▶ Leichte Migration von unterschiedlichen Datenbanken
- ▶ Unterstützung von:
  - Dokumentenbasierte Datenbank
  - Key-Value Store
  - Graph-Datenbanken

# Speichermodell



# Wording

- ▶ Document - Node (Vertices)
- ▶ Collection - Node Type
- ▶ Edge Collection
  - Document Edge speichert Referenzen auf zwei Nodes

# Speichermodell

```
[{  
  "name" : "Charlie",  
  "_id" : "persons/charlie",  
  "_rev" : "2251893512",  
  "_key" : "charlie"  
},  
{  
  "name" : "Bob",  
  "_id" : "persons/bob",  
  "_rev" : "2251631368",  
  "_key" : "bob"  
}]
```

```
{  
  "_id" : "knows/2252942088",  
  "_rev" : "2252942088",  
  "_key" : "2252942088",  
  "_from" : "persons/bob",  
  "_to" : "persons/dave"  
}
```

# Graphen in ArangoDB

## ▶ General Graphs

- Default Graphen in ArangoDB
- Liefern das komplette Featureset von Graph-Datenbanken

## ▶ Blueprint Graph

- Support für Tinkerpop
- Superset von General Graphs
- Apache2 License

# Graphen in ArangoDB

## ▶ Partial Infrastructures

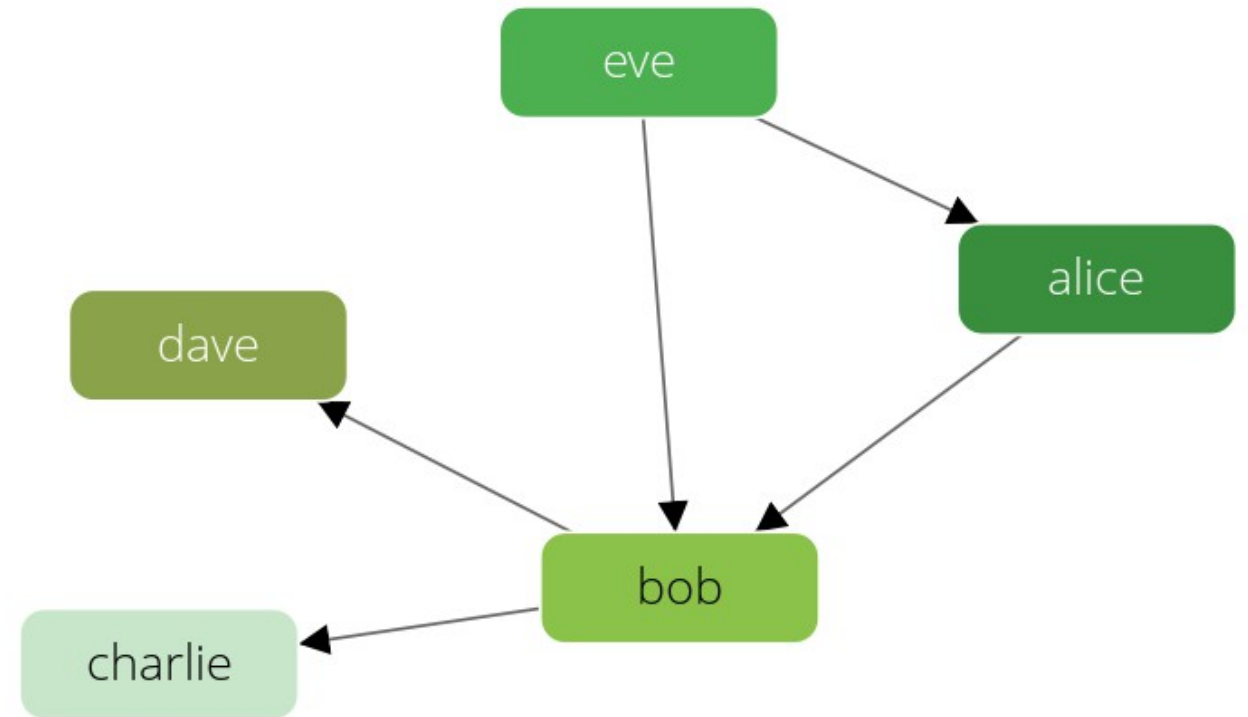
- Teilfunktionalitäten des General Graphs

## ▶ Example Graphs

- Beispielstrukturen von Graphen

# The Knows\_Graph

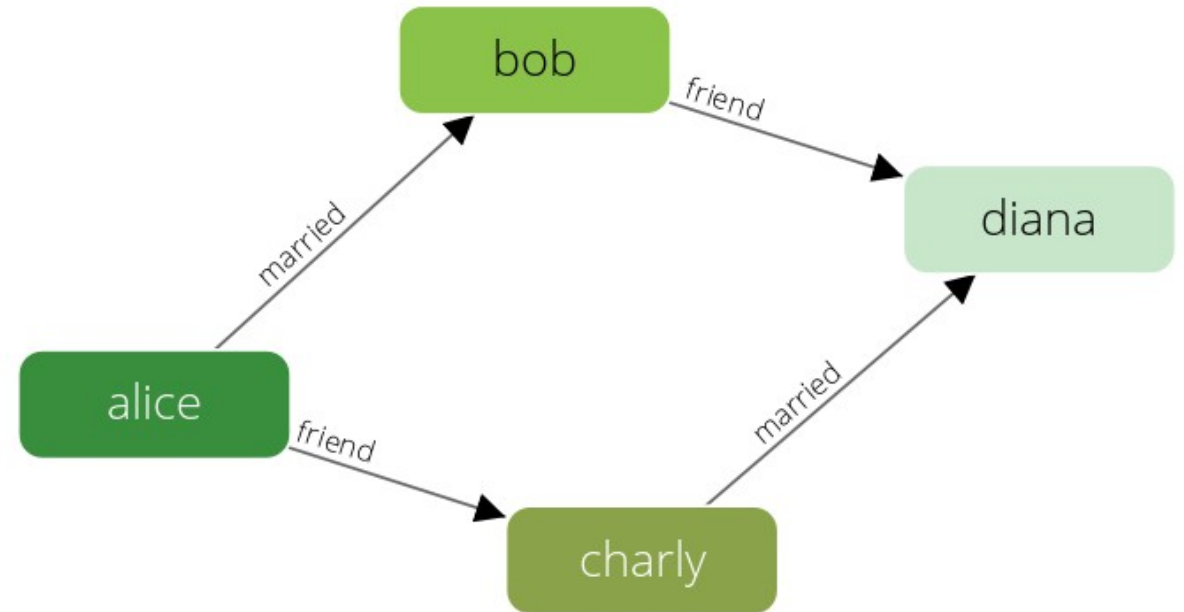
- ▶ Wer-kennt-wen
- ▶ Eine einzige Vertex-Collection
- ▶ Eine einzige Edge-Collection
- ▶ Alice kennt Bob
- ▶ Bob kennt Charlie
- ▶ Bob kennt Dave
- ▶ ...





# The Social Graph

- ▶ Zwei Vertex-Collection:
  - Male
  - Female
- ▶ Eine einzige Edge-Collection:
  - Relation
- ▶ Alice ist verheiratet mit Bob
- ▶ Bob ist befreundet mit Diana
- ▶ ...

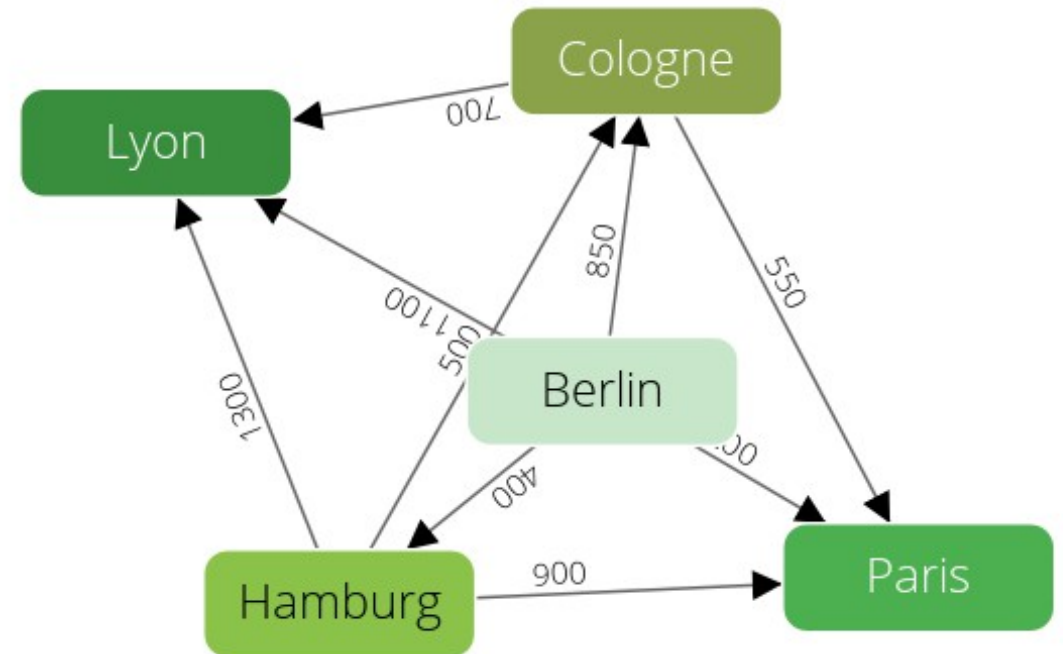


# Relation Dokument

```
{  
  "type" : "friend",  
  "_id" : "relation/bobAndDiana",  
  "_rev" : "2256939784",  
  "_key" : "bobAndDiana",  
  "_from" : "male/bob",  
  "_to" : "female/diana"  
}
```

# The City\_Graph

- ▶ Mehrere Vertex-Collections
  - Deutsche Städte
  - Französische Städte
- ▶ Mehrere Edge-Collections
  - Deutsche Straßen
  - Französische Straßen
  - Internationaler Highway



# Edge Collections

germanHighway

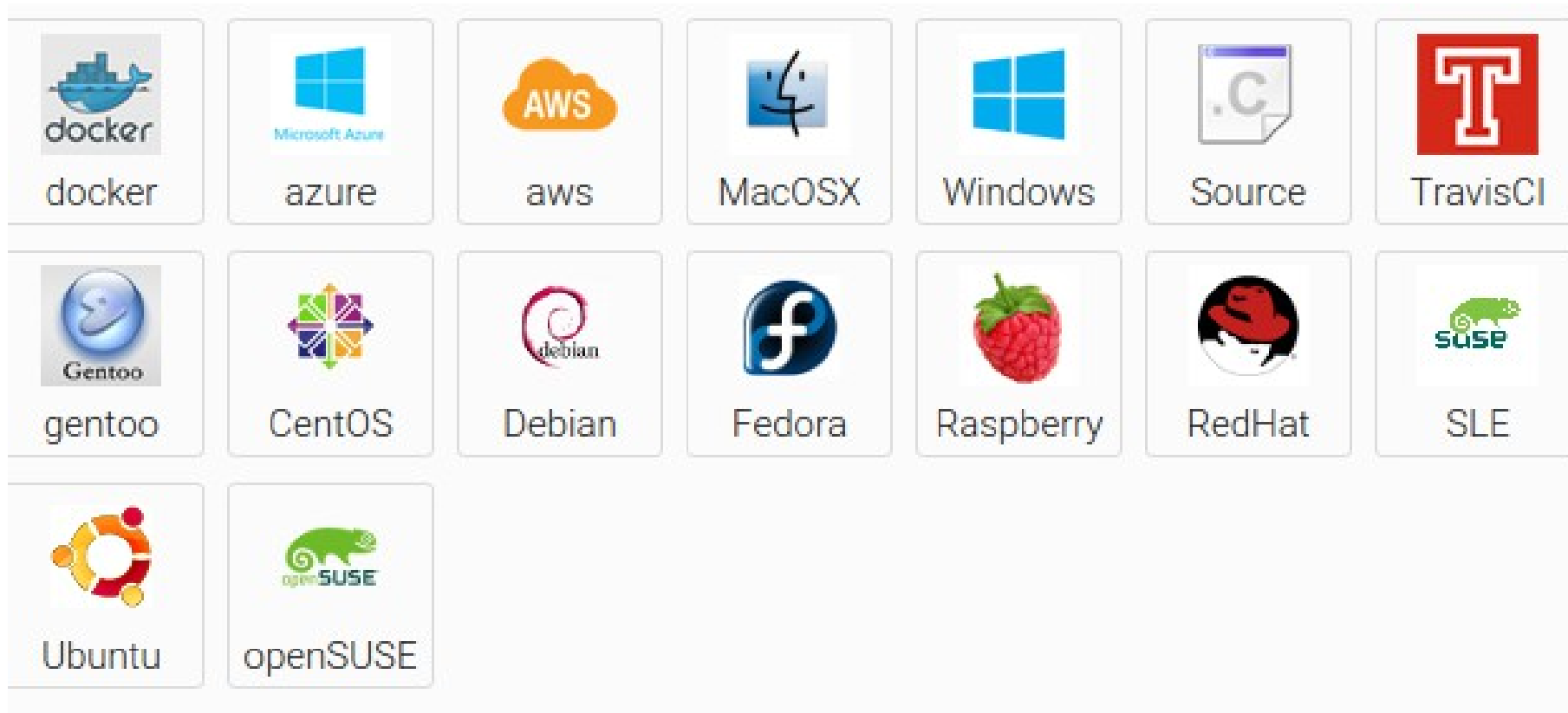
```
{  
  "distance" : 850,  
  "_id" :  
"germanHighway/2247109384",  
  "_rev" : "2247109384",  
  "_key" : "2247109384",  
  "_from" : "germanCity/Berlin",  
  "_to" : "germanCity/Cologne"  
}
```

frenchHighway

```
{  
  "distance" : 550,  
  "_id" : "  
"frenchHighway/2247830280",  
  "_rev" : "2247830280",  
  "_key" : "2247830280",  
  "_from" : "frenchCity/Paris",  
  "_to" : "frenchCity/Lyon"  
}
```

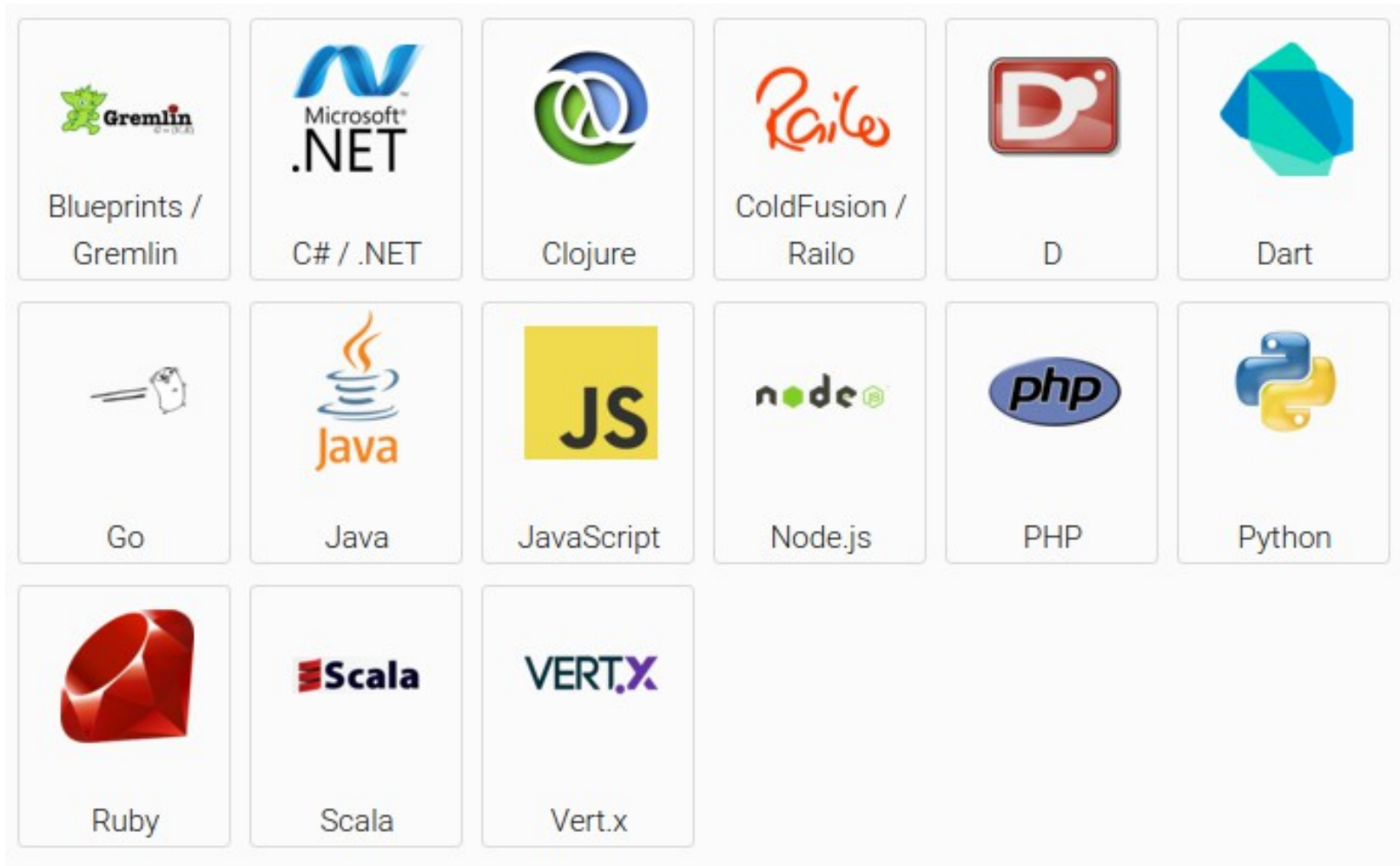
# First Steps

# Installation



<https://www.arangodb.com/download/>

# Installation (Treiber)



# Server starten

## ▶ Linux

```
martina@Deathwing:/usr/local$ sudo /etc/init.d/arangodb start  
[ ok ] Starting arangodb (via systemctl): arangodb.service.  
martina@Deathwing:/usr/local$ sudo /etc/init.d/arangodb stop  
[ ok ] Stopping arangodb (via systemctl): arangodb.service.  
martina@Deathwing:/usr/local$
```

Server erreichbar unter <http://127.0.0.1:8529/>

## ▶ Windows

arangod.exe ausführen in <ROOTDIR>\bin.



# Arangosh - ArangoDB Shell

- ▶ Mitgelieferte CLI für Administrative Aufgaben
- ▶ JavaScript Shell (wie Node)

> *(sudo) arangosh*

- ▶ Default Verbindung zu dem Server localhost auf Port 8529
- ▶ Cheat Sheet:  
[https://www.arangodb.com/wp-content/uploads/2012/08/arangodb\\_1.2\\_shell\\_reference\\_card\\_1.jpg](https://www.arangodb.com/wp-content/uploads/2012/08/arangodb_1.2_shell_reference_card_1.jpg)

# Arangosh Datenbanken

- ▶ `db_createDatabase("dbName");`  
Legt eine Datenbank unter angegebenem Namen an
- ▶ `db_useDatabase("dbName");`  
Datenbank wechseln (muss vorher angelegt sein)
- ▶ `db_dropDatabase("dbName");`  
Löscht angegebene Datenbank

# Beispielgraphen laden

```
var examples=require("org/arangodb/graph-examples/example-graph.js");  
var graph = examples.loadGraph("social");  
// "knows_graph", "routeplanner",  
db.female.toArray()
```

- ▶ Graph anzeigen lassen.

# Hands On

- 1) ArangoDB installieren
- 2) Sever starten
- 3) Arangosh ausführen
- 4) Beispiel Graphen “routeplanner” laden und deren VertexCollection und EdgeCollections anzeigen lassen.

# Arangosh Datenbanken

- ▶ `db_createDatabase("dbName");`  
Legt eine Datenbank unter angegebenem Namen an
- ▶ `db_useDatabase("dbName");`  
Datenbank wechseln (muss vorher angelegt sein)
- ▶ `db_dropDatabase("dbName");`  
Löscht angegebene Datenbank

# Graphen anlegen

- ▶ Modul für den General Graph laden

```
var graph_module = require("org/arangodb/general-graph");
```

- ▶ Eine eigenen Graphen daraus initialisieren

```
var graph = graph_module._create("myGraph");
```

- ▶ Graph anzeigen lassen

```
graph;
```

```
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [ ] ]
```

# Graphen löschen

```
graph_module._drop(graphName, dropCollections)
```

- ▶ *graphName*: Eindeutige ID des Graphen
- ▶ *DropCollections*: Bestimmt ob Collections mit gelöscht werden sollen (Default false)
  - Collections können immernoch mit `db._collection("collectionName");` aufgerufen werden

# VertexCollection anlegen

```
graph._addVertexCollection("nodeName");
```

- ▶ Legt eine Collection / NodeType nach entsprechendem Namen an

```
graph;
```

```
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [  
  "nodeName1",  
  "nodeName2"  
] ]
```



# EdgeCollection anlegen

- ▶ Relation initialisieren und deklarieren

```
var rel = graph_module._relation("loves", ["female"], ["male"]);
```

- ▶ Dem Graphen die EdgeCollection hinzufügen

```
graph._extendEdgeDefinitions(rel);
```

```
graph;
```

```
[ Graph myGraph EdgeDefinitions: [
```

```
  "loves: [female] -> [male]"
```

```
] VertexCollections: [ ] ]
```

# Kanten manipulieren

- ▶ `graph_module._editEdgeDefinition(edgeDefinition)`

```
var old = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
```

```
var new= graph_module._relation("myEC1", ["myVC2"], ["myVC3"]);
```

- ▶ Alte Edge Definition laden

```
var graph = graph_module._create("myGraph", [old]);
```

- ▶ Neue Edge Definition auf geladenes Objekt anwenden

```
graph._editEdgeDefinitions(modified);
```

# Knoten anlegen

- ▶ Eine Katze in entsprechende “cat” VertexCollection anlegen:
- ▶ `graph.cat.save({name: "Morle", _key: "morle"});`

```
arangosh [myCats] > graph.cat.save({name: "Morle", _key: "morle"});
{
  "_id" : "cat/morle",
  "_rev" : "1590634787",
  "_key" : "morle"
}
arangosh [myCats] > graph
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [
  "cat"
] ]
```

# Knoten ersetzen

- Den Name der Katze mit der \_id "cat/morle" ändern  
`graph.cat.replace("cat/morle", {name: "John"});`

```
arangosh [myCats] > graph.cat.replace("cat/morle", {name: "John"});
{
  "_id" : "cat/morle",
  "_rev" : "1628055843",
  "_key" : "morle"
}
arangosh [myCats] > db.cat.toArray();
[
  {
    "name" : "John",
    "_id" : "cat/morle",
    "_rev" : "1628055843",
    "_key" : "morle"
  }
]
```

# Knoten updaten

- Felder des Dokuments updaten / manipulieren  
`graph.cat.update("cat/morle", {name: "Chris"});`

```
arangosh [myCats] > graph.cat.update("cat/morle", {name: "Chris"});
{
  "_id" : "cat/morle",
  "_rev" : "1637034275",
  "_key" : "morle"
}
arangosh [myCats] > db.cat.toArray();
[
  {
    "name" : "Chris",
    "_id" : "cat/morle",
    "_rev" : "1637034275",
    "_key" : "morle"
  }
]
```

# Knoten löschen

- ▶ Vertex aus dem Graphen herauslöschen

```
db._exists("cat/morle") // if true  
graph.cat.remove("cat/morle")
```

# Edge anlegen

- ▶ Eine Relation/ Edge zwischen zwei Documents anlegen  
`graph.edgeCollectionName.save(from, to, data, options)`
- ▶ From: ID des Documents zu der Beziehung abgeht
- ▶ To: ID des Documents zu der Beziehung hinführt
- ▶ data: JSON.Objekt der relation
- ▶ Options: `waitForSync`, `waitForSave` ...

# Edge anlegen

```
graph.relation.save("cat/morle", "cat/chris", {type: "married",  
_key: "morleAndChris"});
```

- ▶ VertexCollection und EdgeCollection müssen zuvor angelegt sein



# Edge manipulieren

## ▶ Replace

```
graph.relation.replace("relation/martinaAndChris",{type: "knows"});
```

## ▶ Update

```
graph.relation.update("relation/martinaAndChris",{type: "married",  
_key: "martinaAndChris"});
```

## ▶ Delete

```
db._exists("relation/martinaAndChris")  
graph.relation.remove("relation/martinaAndChris")
```

# Funktionen des Graphen

- ▶ Knoten auslesen via EdgId
  - **From**  
`graph._fromVertex(edgId)`
  - **To**  
`graph._toVertex(edgId)`
- ▶ RückgabeTyp ist hierbei ein JSON

# Funktionen des Graphen

- ▶ Alle Nachbarn eines Knoten bekommen  
`graph._neighbors(vertexExample, options)`
- ▶ VertexExample:
  - Null: Alle Knoten werden angezeigt
  - String: VertexID
  - JSON Objekt mit Key-Value Kriterium Attributen

# Funktionen des Graphen

- ▶ Options:
  - MinDepth / maxDepth
  - Vertex/edgeExamples

```
graph._neighbors('germanCity/Hamburg',{direction : 'outbound',  
maxDepth : 2});
```

# Funktionen des Graphen

## ▶ **\_shortestPath**

`graph._shortestPath(startVertexExample, endVertexExample, options)`

- Ermittelt den kürzesten Pfad angegebener Knoten
- Gibt Traversierte Edges und Vertices zurück

## ▶ **\_distanceTo**

`graph._distanceTo(startVertexExample, endVertexExample, options)`

Ermittelt die Distanz zweier Knoten

# Funktionen des Graphen

## ▶ **\_absoluteCloseness**

*graph.\_absoluteCloseness(vertexExample, options)*

- Ermittelt die Distanz zu angegebenen Knoten

Fragen?