



# Nicht-Standard Datenmodelle und Datenbanken

1. Einführung
2. RDF und Semantic Web
3. Graphdatenbanken

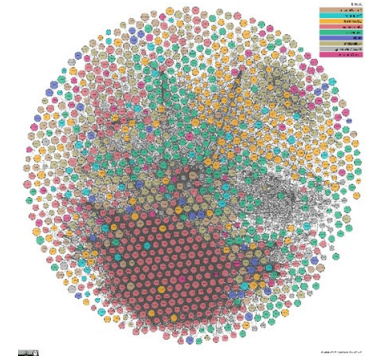
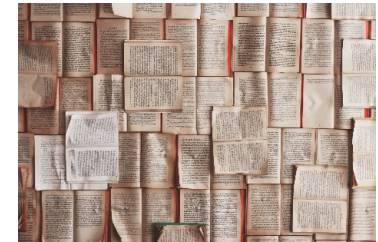


# Nicht-Standard Datenmodelle und Datenbanken

1. **Einführung**
2. RDF und Semantic Web
3. Graphdatenbanken

# Strukturierte und Semistrukturierte Datenmodelle

- Unstrukturierte Daten
  - Daten folgen keinem Format, Schema oder Grammatik
  - Text, Bitströme auf Speichern, rohe Video- & Bilddaten, ...
- Semi-Strukturierte Daten
  - Daten folgen einem flexiblen Format, optionale Felder, graphbasierte Daten
  - Webseiten, XML documente, JSON, RDF, ...
- Strukturierte Daten
  - Daten folgen einem striktem Schema
  - relationale Datenbanken, Tabellen, Sensordaten, ...



NoSQL  
(Not Only SQL)

Personalnummer	Name	Gehalt
1427	Meier	3217,33
8219	Schmidt	1425,87
2624	Müller	2438,21
⋮	⋮	⋮

Personalnummer	Abteilung
1427	3-1
8219	2-2
2624	3-1
⋮	⋮

### *Not only SQL*

Relationale Datenbanken sind für große Datenmengen **oder** viele gleichzeitige Zugriffe optimiert. Das Internet hat den Bedarf an Big Data Lösungen in dem die Datenmenge groß **und** die Anzahl der Zugriffe hoch ist

- **Nicht-Relational:** Keine Tabellenstruktur, kein Schema, ermöglicht agile Verarbeitung großer Datenmengen.
- **Verschiedene Datenmodelle**
- **Skalierbarkeit:** Hoch skalierbar, geeignet für Verteilung auf viele Server.
- **Flexibilität:** Schema-los, bietet Flexibilität bei Datenspeicherung und -manipulation.
- **Leistung:** Schnelle Schreib- und Lesevorgänge, ideal für Echtzeitanwendungen.
- **Big Data & Echtzeit-Web-Apps:** Ideal für Big Data und schnelle Webdienste.
- **Konsistenzmodell:** Nutzt "eventuelle Konsistenz" für hohe Verfügbarkeit.

# Skalierung des Datenzugriffs

---

Traditionelle DBMS skalieren:

## 1. Vertikal:

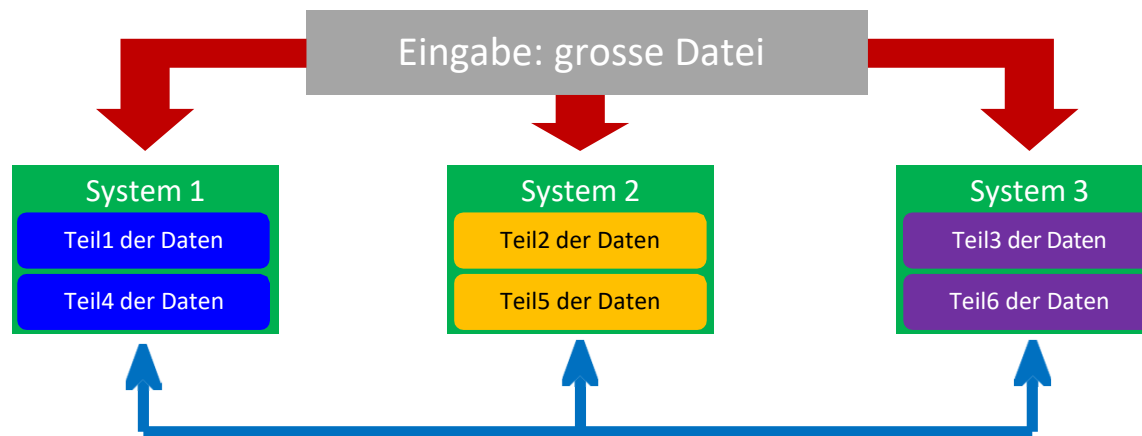
- Durch Aufrüsten der Hardware (z.B., schneller CPU, mehr RAM oder Sekundärspeicher)
- Limitiert durch CPUs, RAM, Sekundärspeicher, Netzwerkbandbreite, die mit einer einzelnen Maschine möglich sind

## 2. Horizontal:

- Durch hinzufügen von mehr Maschinen
- Benötigt verteilte Datenbanken und wahrscheinlich Replikationen
  - Verteilt Tabellen auf verschiedene Maschinen
  - Verteilt Tupel von Tabellen auf verschiedene Maschinen
  - Verteilt Spalten von Tabellen auf verschiedene Maschinen
- Limitiert durch Lese-Schreib Verhältnis (Synchronisation/Konflikte) sowie Kommunikationsoverhead

## Sharding: Verteilen von Zeilen

Grössere Performance durch Verteilung von Zeilen der Tabellen auf mehrere DBMS („sharding“). Sharding ist nötig für die Parallelisierung.



Z.B., Teil 1,5 und 3 können parallel verarbeitet werden

## Grenzen der Parallelisierung: das Amdahlsches Gesetz

---

- Das Amdahlsche Gesetz ist benannt nach Gene Amdahl, einem Pionier der Computerarchitektur.
- Es wird verwendet, um den maximalen Verbesserungsfaktor zu berechnen, der durch Parallelisierung einer Aufgabe erzielt werden kann.
- Annahmen:
  - Die sequentielle Ausführung eines Programmes braucht  $t$  Zeiteinheiten, und die parallele Ausführung auf  $p$  Prozessoren braucht  $T_p$  Zeiteinheiten.
  - $s$  steht für den nicht-parallelisierbaren Teil des Programmes, daher  $1 - s$  für den parallelisierbaren Teil.
- Dann ist der maximale Performanzgewinn gegeben durch Amdahls Formel: 
$$\frac{t_1}{t_p} = \frac{t_1}{t_1 \times s + t_1 \times \frac{1-s}{p}} = \frac{1}{s + \frac{1-s}{p}}$$
- Das Amdahlsche Gesetz stellt fest, dass die Verbesserung der Systemleistung durch Parallelisierung durch den Anteil der Aufgabe begrenzt ist, der nicht parallelisiert werden kann.

## Beispiel

---

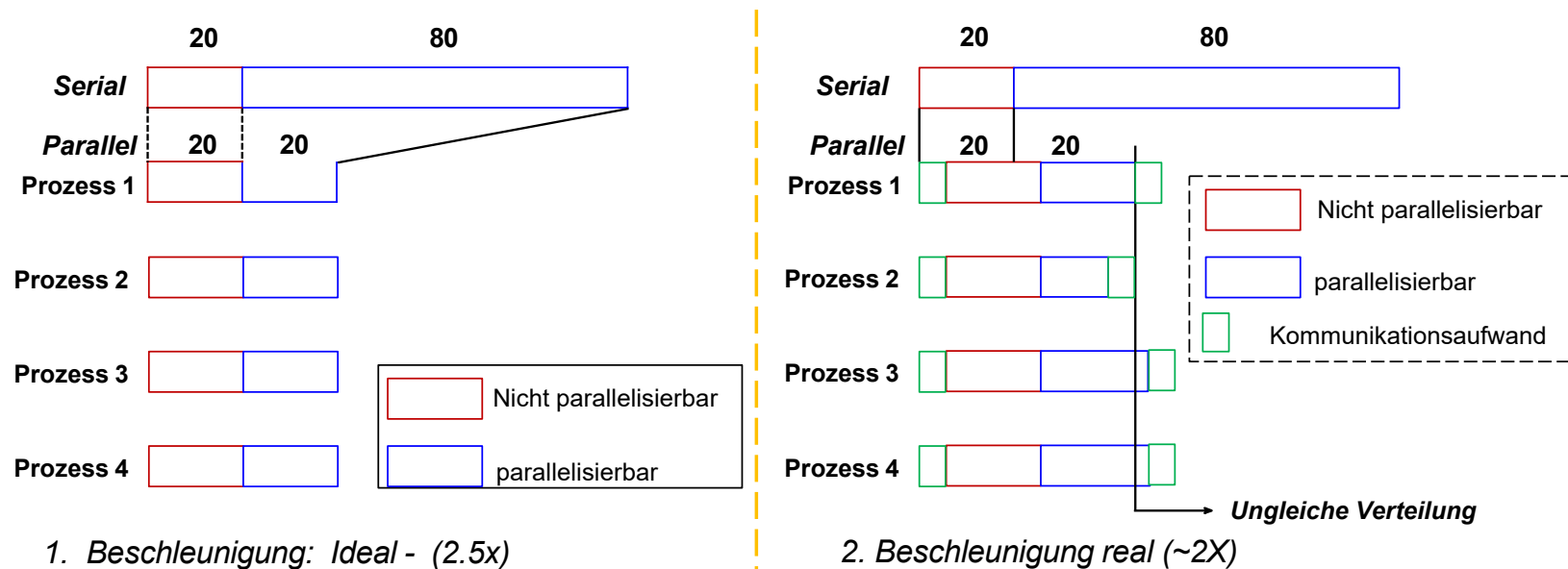
- Annahmen:
  - 60% einer Anfragebearbeitung kann parallelisiert werden
  - 6 Maschinen werden bei der Tupelselektion verwendet.
- Nach dem Amdahlschen Gesetz ist die maximale Beschleunigung:  $\frac{1}{s + \frac{1-s}{p}} = \frac{1}{0,4 + \frac{0,6}{6}} = 2.0$
- Das Amdahlsche Gesetz stellt fest, dass die Verbesserung der Systemleistung durch Parallelisierung durch den Anteil der Aufgabe begrenzt ist, der nicht parallelisiert werden kann.

Trotz 6 Prozessoren bekommt man nur die doppelte Performanz!



## Kommunikation und ungleiche Verteilungen

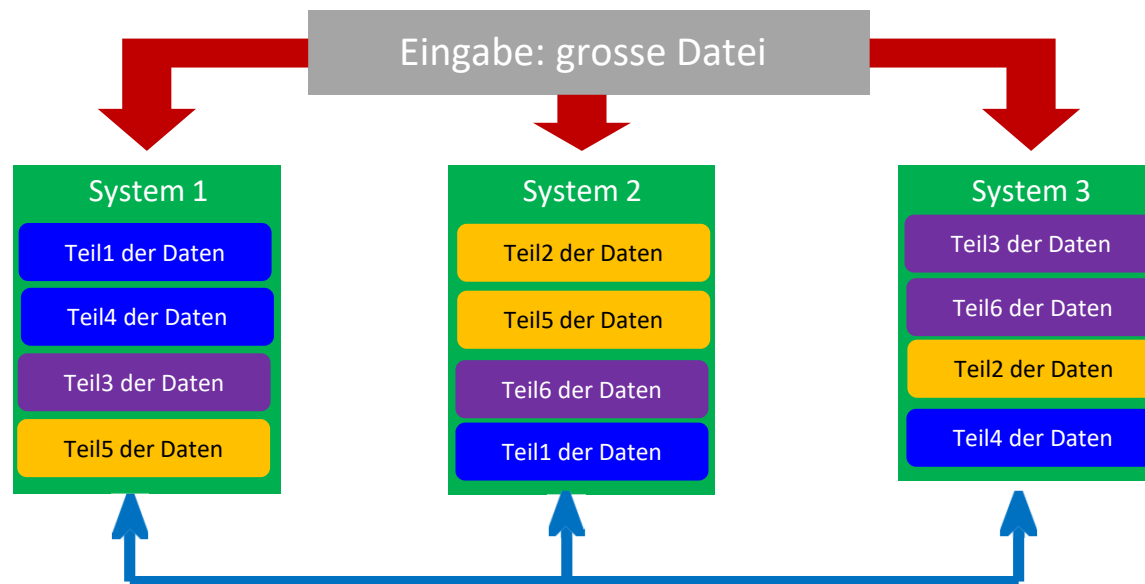
- Tatsächlich ist das Amdahlsche Gesetz zu sehr vereinfacht
- Kommunikationsaufwand und ungleiche Verteilung bei der Verarbeitung beeinflussen ebenfalls parallele Programme



# Sharding und Replizierung

## Warum Datenreplizierung?

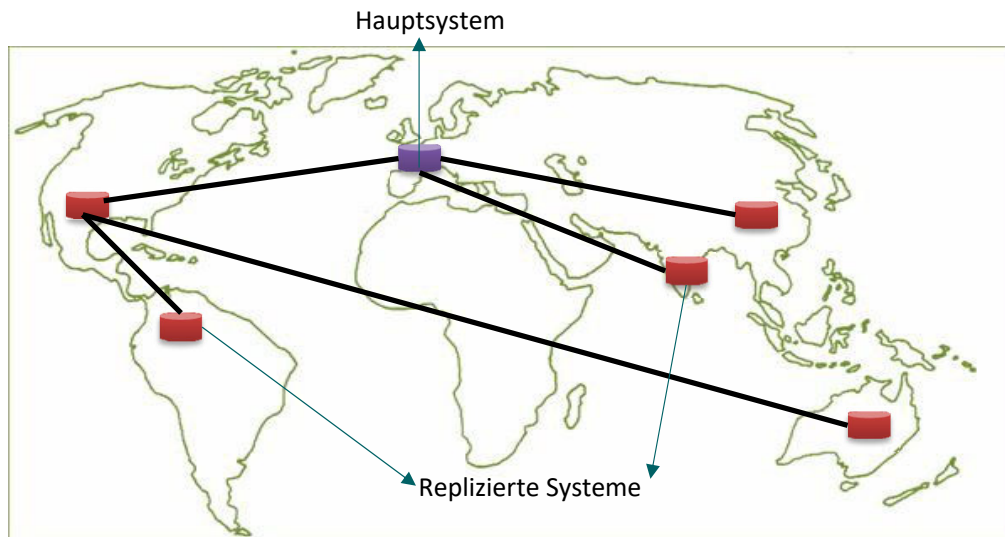
- Replizierung von Daten über Systeme hilft bei:
  - Vermeidung von Performanzflaschenhälsen
  - Erhöhung der Fehlertoleranz durch Vermeidung von Single Point of Failure



# Sharding und Replizierung

## Warum Datenreplizierung?

- Replizierung von Daten über Systeme hilft bei:
  - Vermeidung von Performanzflaschenhälsen
  - Erhöhung der Fehlertoleranz durch Vermeidung von Single Point of Failure
  - Verbessert auch die *Skalierbarkeit* und *Verfügbarkeit*

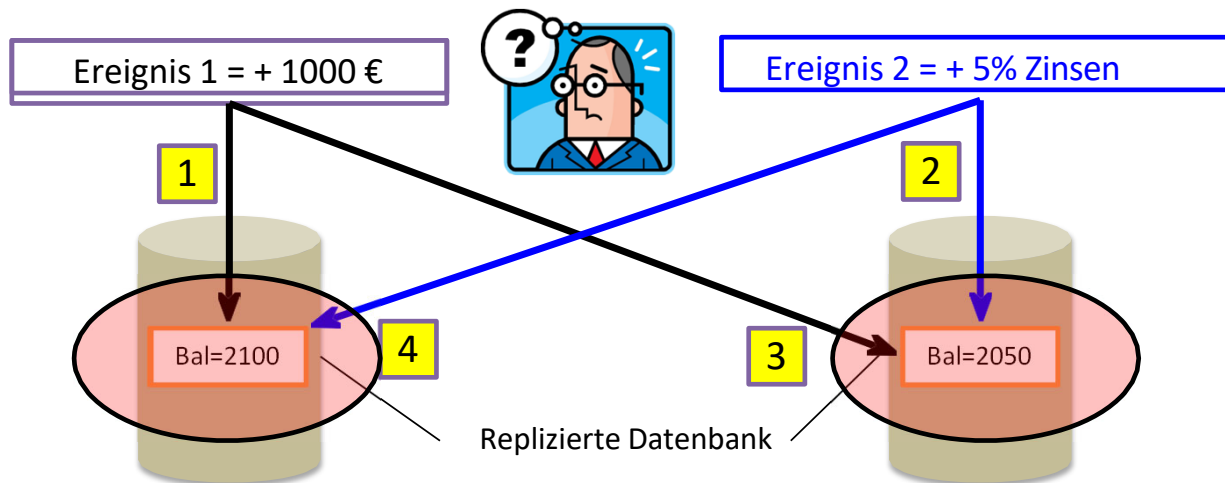


## Aber...

Konsistenz wird schwierig.

Beispiel:

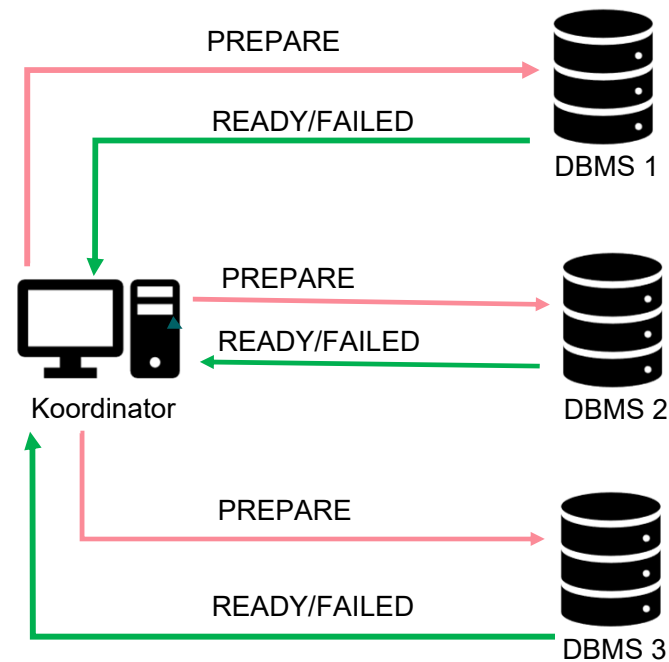
- Die Datenbank einer Bank sei über zwei Systeme repliziert
- Beibehaltung der Konsistenz der replizierten Daten wirft Fragen auf
- Scheduling nimmt serielle Ausführung an...



## Zwei Phasen Commit (2PC) Protokoll

Das Zwei-Phasen Comm Protokoll kann zur Sicherstellung der Atomarität und Konsistenz verwendet werden, erhöht aber den seriellen Anteil der Transaktionen, und braucht gewöhnlich eine zentrale „Lock-Autorität“ oder Koordinator.

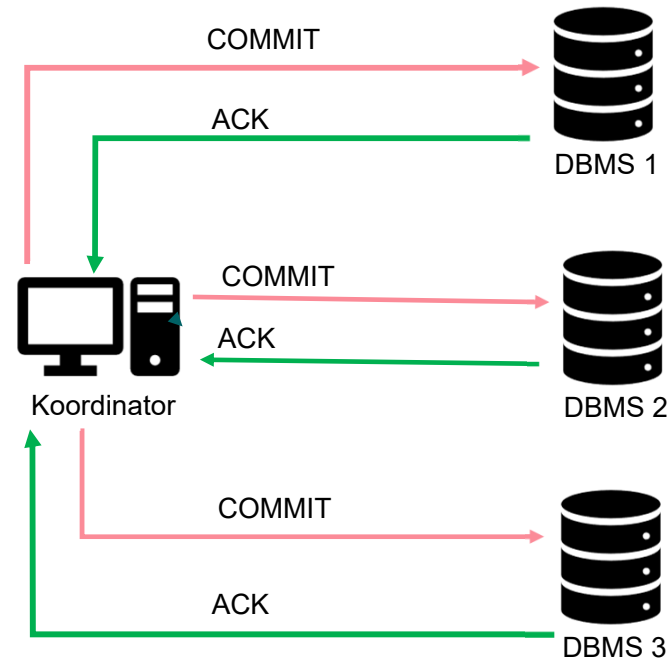
Phase 1:  
PREPARE



## Zwei Phasen Commit (2PC) Protokoll

Das Zwei-Phasen Commit Protokoll kann zur Sicherstellung der Atomarität und Konsistenz verwendet werden, erhöht aber den seriellen Anteil der Transaktionen, und braucht gewöhnlich eine zentrale „Lock-Autorität“ oder Koordinator.

Phase 2:  
COMMIT



Strikte Konsistenz limitiert Skalierbarkeit!

## Das CAP Theorem

---

Das CAP-Theorem, auch bekannt als Brewers Theorem, ist ein grundlegendes Prinzip, das die Einschränkungen von verteilten Datenbanksystemen beschreibt. CAP steht für Konsistenz (Consistency), Verfügbarkeit (Availability) und Partitionstoleranz (Partition Tolerance). Die drei Eigenschaften sind:

1. **Konsistenz (Consistency)**: Jeder Knoten sieht zu jedem gegebenen Zeitpunkt immer die gleichen Daten (d.h., strikte Konsistenz)
2. **Verfügbarkeit (Availability)**: Weiterbetrieb, auch wenn Knoten in einem Cluster abstürzen oder einige Hardware- oder Softwareteile aufgrund von Upgrades ausfallen
3. **Partitionstoleranz (Partition Tolerance)** : Weiterbetrieb bei Vorhandensein von Netzwerkpartitionen (Unterbrechungen in der Konnektivität)

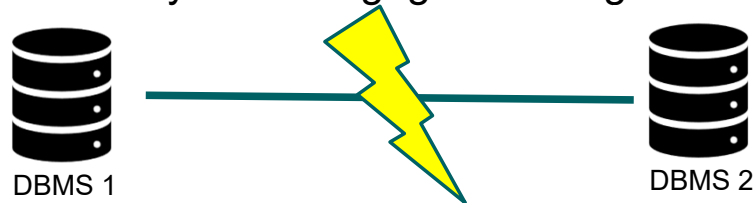
CAP-Theorem: Jede verteilte Datenbank mit geteilten Daten kann **höchstens zwei der drei** wünschenswerten Eigenschaften, C, A oder P, garantieren.

Gilbert, S., & Lynch, N. (2002, June). Brewers Conjunction and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News , p. 33(2).

## CAP Beispiele

---

Angenommen wir haben zwei System auf gegenüberliegenden Seiten einer Netzwerkpartition:



- Verfügbarkeit und Partitionstoleranz gibt die Konsistenz auf
- Konsistenz und Partitionstoleranz heißt die Systeme sind bei Netzwerkpartition nicht verfügbar, da Daten nicht synchronisiert werden können.
- Konsistenz und Verfügbarkeit ist nur möglich wenn es keine Netzwerkpartition gibt, da die Partitionstoleranz aufgegeben wird.

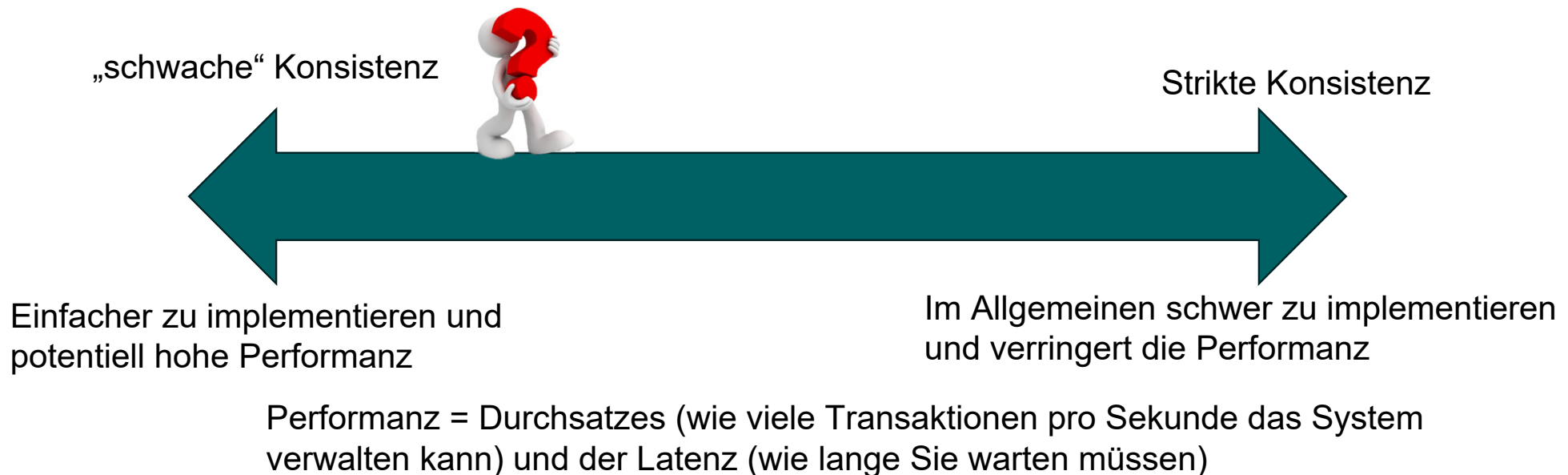


- 24/7 Verfügbarkeit ist bei Anwendungen, wie sie Unternehmen wie Google und Amazon haben, wichtig
  - Jede Nicht-Verfügbarkeit bedeutet Umsatzverluste
- Bei Skalierung einer Datenbank auf tausende von Systemen wächst die Wahrscheinlichkeit von Ausfällen oder Netzwerkproblemen stark an
- Um Verfügbarkeit und Partitionstoleranz zu garantieren, muss daher nach dem CAP-Theorem die (strikte) Konsistenz geopfert werden.

## Konsistenzkompromisse

---

- Balance bzgl. zwischen Konsistenz vs. Verfügbarkeit und Skalierbarkeit
- Was „ausreichende“ Konsistenz ist, hängt von der Anwendung ab



## BASE

---

- CAP-Theorem zeigt das strikte Konsistenz, Verfügbarkeit und Partitionstoleranz gleichzeitig nicht garantiert werden dann.
- Daher: Datenbanken mit Relaxierung der ACID-Eigenschaften
- Insbesondere Anwendung der **BASE** Eigenschaften:
  - **B**asically **A**vailable: das System garantiert Verfügbarkeit
  - **S**oft State: der Zustand des Systems verändert sich über die Zeit und kann für kurze Zeitintervalle inkonsistenz sein
  - **E**ventually Consistent: das System wird irgendwann konsistent

Dan Pritchett: BASE: An Acid Alternative. ACM Queue 6(3): 48-55 (2008)  
Werner Vogels: Eventually consistent. Commun. ACM 52(1): 40-44 (2009)

## Eventually Consistent

---

- Eine Datenbank hat die Eigenschaft *eventuell konsistent* zu sein, wenn:
  - Alle Kopien graduell auf einen einzigen konsistenten Zustand konvergieren, falls es in einem spezifizierten Zeitintervall keine Änderungen gibt.

## NoSQL (=Not Only SQL) Databases

---

Eigenschaften von NoSQL Datenbanken:

- Keine striktes Schema
- Keine strikte Einhaltung der ACID Prinzipien
- Verfügbarkeit wichtiger als (strikte) Konsistenz
- Konsistenz wird irgendwann erreicht, wenn es keine Updates gibt

Auch: nur einige dieser Eigenschaften

### Taxonomie von NoSQL Datenbanken:

- Key/Value-Datenbanken
- Dokumentdatenbanken
- Graph-Datenbanken
- spaltenorientierte Datenbank

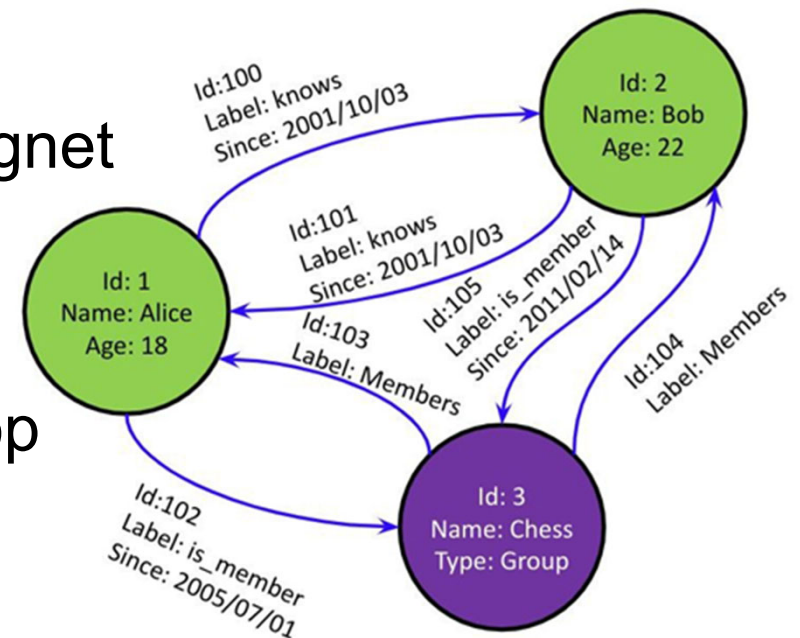
## Dokumentendankbanken

---

- Dokumente und Texte werden in einem etabliertem Format gespeichert (z.B., XML, JSON, PDF, Office)
- Dokumente können indexiert werden
  - Bessere Performanz als Dateisysteme
- Beispiele: MongoDB, CouchDB

## Graphdatenbanken

- Daten werden durch Knoten und Kanten dargestellt
  - Knoten sind wie ER „Entities“
  - Kanten sind ER „Relationen“
- Insbesondere für heterogene Daten geeignet
- Beispiele: Neo4J, RDF, Apache Tinkerpop





## Key-Value Datenbanken

---

- Schlüssel werden auf (möglicherweise komplexe) Werte abgebildet
- Schlüssel können gehashed und verteilt werden
- Solche Datenbanken unterstützen normalerweise CRUD (create, read, update, delete) Operationen
  - Keine Joins oder Aggregateoperationen
- Beispiele: Amazon DynamoDB oder Apache

- Hybrid zwischen Key-Value Datenbanken und relationalen Datenbanken
  - Werte werden in Gruppen von Null oder mehr Spalte für Spalte gespeichert.
  - Zeilenorientiert:  
1,Schmidt,Josef,40000; 2,Müller,Maria,50000; 3,Meier,Julia,44000;
  - Spaltenorientiert:  
1,2,3;Schmidt,Müller,Meier;Josef,Maria,Julia;40000,50000,44000;
- Beispiel: SAP Hana

- Datenbanken können nach strukturierte und nicht-strukturierte klassifiziert werden
- Datenbanken können horizontal oder vertikal skaliert werden
- Strikte Konsistenz beschränkt Skalierbarkeit
- CAP Theorem motiviert BASE Eigenschaften
- Verschiedene Typen von NoSQL Datenbanken
  - Im folgenden: Fokus auf Graphdatenbanken



# Nicht-Standard Datenmodelle und Datenbanken

1. Einführung
2. **RDF und Semantic Web**
3. Graphdatenbanken



# Das RDF Datenmodell

## Informationsdarstellung in Text

---



Aachen ist eine Großstadt in Nordrhein-Westfalen, Deutschland. Aachen ist außerdem eine Universitätsstadt. Aachen hat eine Bevölkerung von 247380 Menschen, [...]...

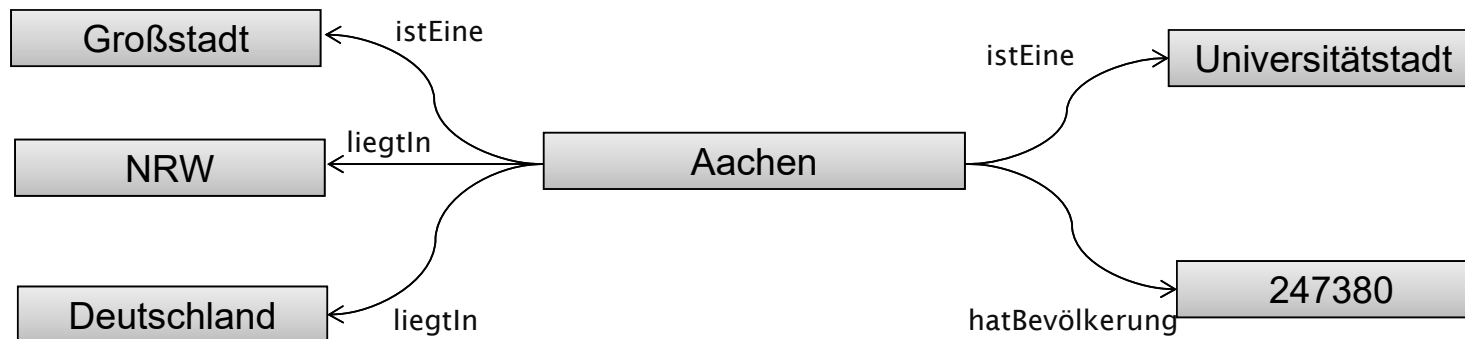
- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.
- ...

## Informationsdarstellung in Tripeln

- Vereinfachte Darstellung der Informationen folgt einer durchgängigen **Subjekt - Prädikat - Objekt** – Struktur, auch Tripel genannt
- Tripel: Beziehung einer Ressource zu anderen Ressourcen und Werten

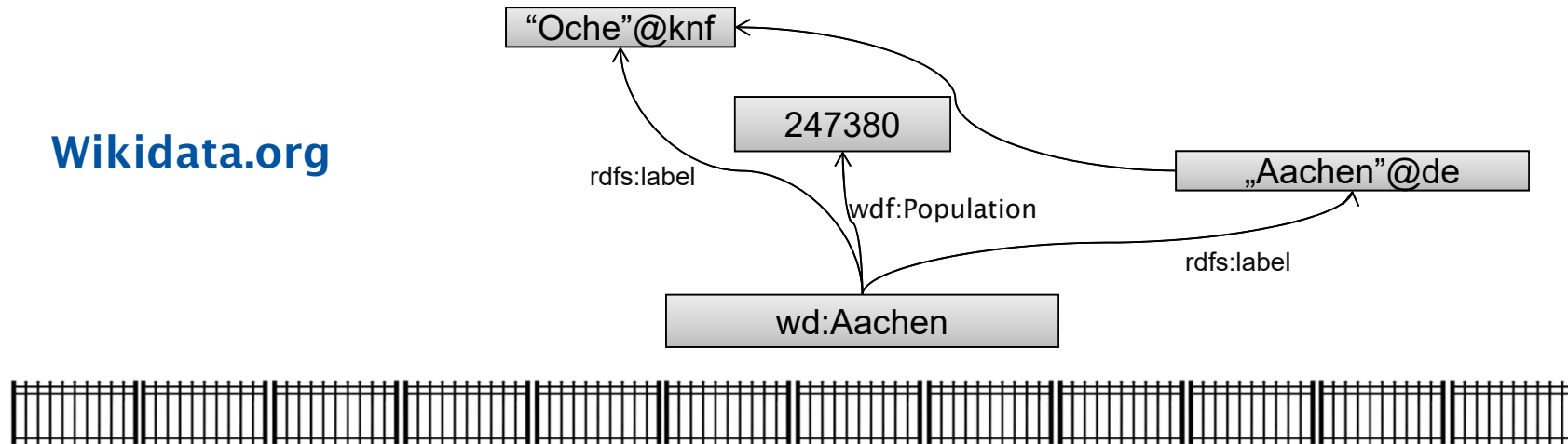


- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.

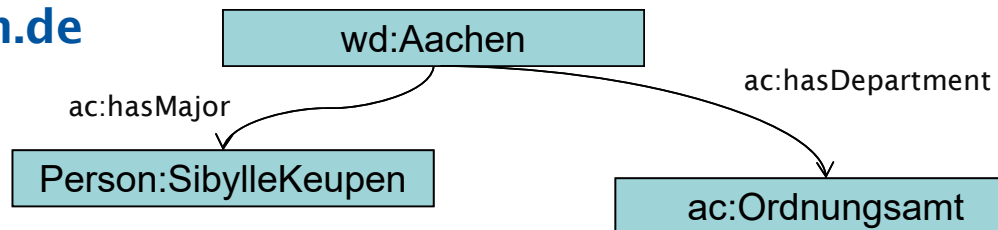


# Wieso Graphen?

## Wikidata.org

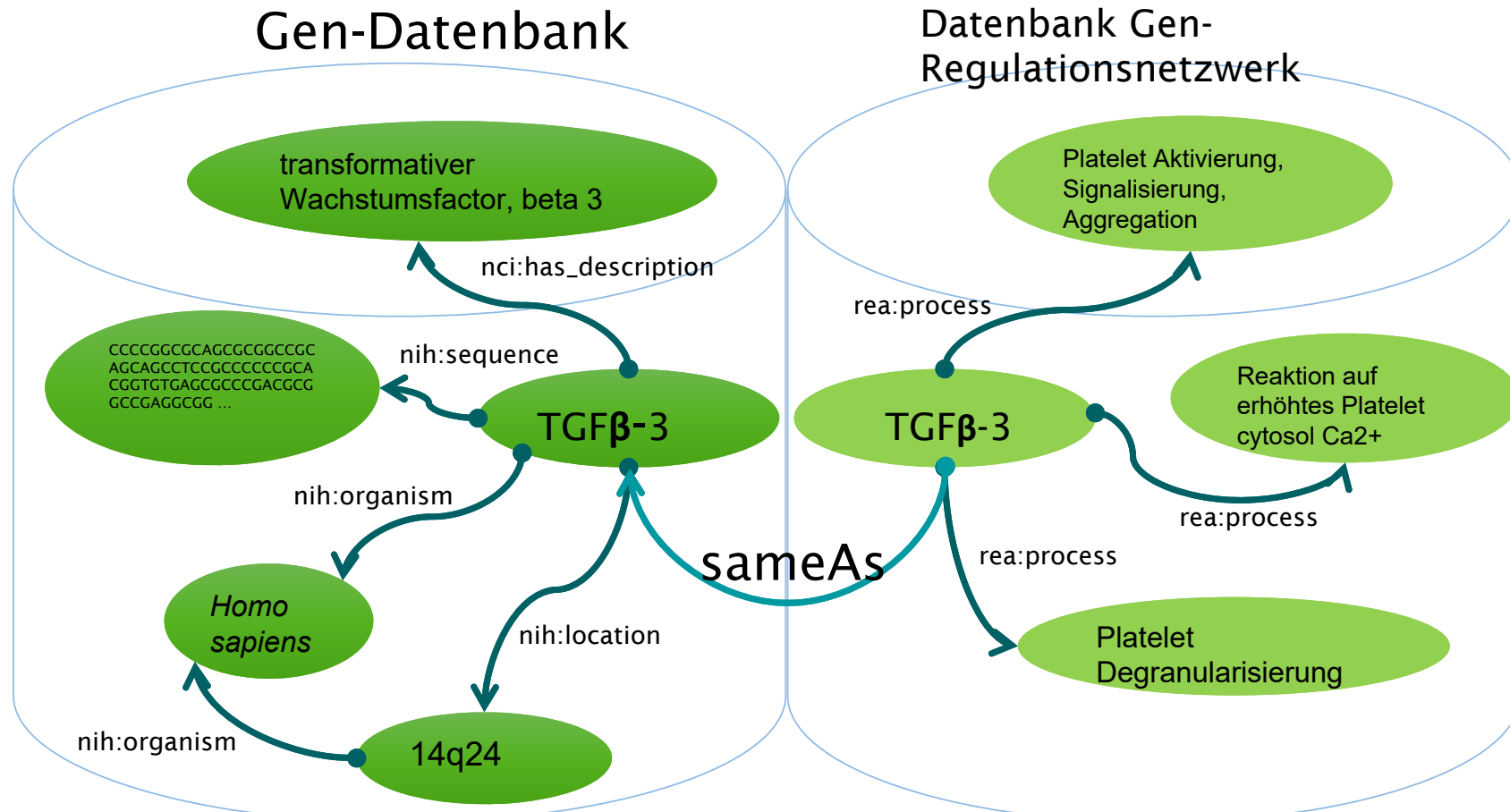


## OpenData.Aachen.de





# Ein Beispiel aus der Biologie



# Standards für das Web

---

## 1. RDF – Resource Description Framework

- Ein Graph-basiertes Datenformat: Knoten und Kanten
- Objekte eindeutig identifiziert durch URIs
- Information wird verknüpft

## 2. Vokabulare und Ontologien

- Ermöglicht das gemeinsame Verständnis für eine Domäne
- Erlauben das Organisieren von Wissen in einem maschinen-lesbaren Format
- Gibt Daten einen auswertbare Bedeutung



# RDF Überblick

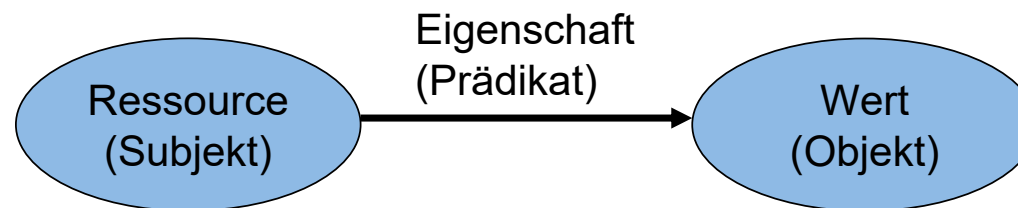
---

- RDF = Resource Description Framework
  - W3C Empfehlung seit 1998
    - <http://www.w3.org/RDF>
  - Version 1.1 seit 2014
    - <http://www.w3.org>
- RDF ist ein Datenmodell
  - Zuerst nur für Metadaten auf Web-Seiten verwendet, dann generalisiert
  - Drückt strukturierte Informationen aus
  - Universales Format zum automatisierten Datenaustausch
- Graph-basierte Datenstruktur
  - Mit Knoten und Kanten

# Der RDF Kern

---

- RDF baut auf den Beziehungen zwischen Ressourcen auf
- Tripel der Form (s, p, o) sind der fundamentale Baustein von RDF
  - **Subjekt**
  - **Prädikat**
  - **Objekt**
- RDF verwendet Identifikatoren vom Web (URIs) um Ressourcen zu identifizieren

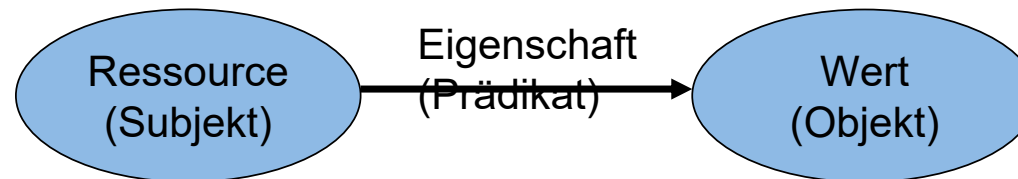


***Das Subjekt*** hat eine ***Eigenschaft*** mit einem ***Wert***

# RDF Tripel

---

- RDF Tripel:
  - **Subjekt:** Eine Ressource kann eine URI oder ein Blank Node sein.
  - **Prädikat:** Eine URI welche eine Eigenschaft der Ressource identifiziert.
  - **Objekt:** Der Wert einer Eigenschaft der Ressource. Dies kann eine URI, ein Literal oder ein Blank Node sein.



***Das Subjekt*** hat eine ***Eigenschaft*** mit einem ***Wert***

# Die Teile eines RDF Graphen

---

- URIs
  - Verwendet um Ressourcen eindeutig zu referenzieren
- Literale
  - Beschreiben Wertigkeiten
- „Blank Nodes“
  - Erlauben die existentielle Quantifizierung der Eigenschaften einer Entität ohne ihr einen Namen zu geben
- RDF Graphen müssen **nicht zusammenhängend** sein

# Was sind URIs ?

---

- URI = Uniform Resource Identifier  
<https://tools.ietf.org/html/rfc3986>
- Weltweiter, eindeutiger Identifikator für Ressourcen
- Jedes Objekt kann eine Ressource sein, wenn es eine eindeutige Identität hat
  - **Beispiele:** Bücher, Orte, Personen, Beziehungen zwischen diesen Dingen, oder abstrakte Konzepte
- Eindeutige Kennzeichner werden schon für bestimmte Domänen verwendet, z.B. ISBN für Bücher oder Steuer-ID für Personen in Deutschland
- URIs sind eine Erweiterung der URL
  - Nicht jede URI gehört zu einer Webseite, aber meistens werden URLs als URIs für Webseiten verwendet.

# Syntax von URIs

---

- Protokoll “:“ Hierarchie [ “?” Anfrage ] [ “#“ Fragment ]
- Beispiele:
  - `http://en.wikipedia.org/w/index.php?search=rdf`
  - `http://en.wikipedia.org/wiki/Resource_Description_Framework#Examples`
  - `urn:example:animal:ferret:nose`



# Literale

---

- Verwendet um Datenwerte auszudrücken
- Durch Strings repräsentiert
- Interpretation des Literals anhand des **Datentyps**
- Literale können niemals der Ursprung einer Kante in einem RDF Graph sein (Literale können nicht Subjekt eines Tripels sein)
- Literale können nicht einer Kante zugeordnet sein
  
- **Language Tags:** Optionale Auskunft über die Sprache eines Strings, kann **nicht** zusammen mit einem Datentyp verwendet werden. Beispiel: "Aachen"@DE

# Datentypen für Literale

---

- Literale ohne Datentypen werden wie Strings behandelt
  - Beispiel mit „kleiner als“: "02" < "100" < "11" < " 2"
- Datentypen erlauben eine semantische Interpretation
- Datentypen werden durch frei wählbare URIs identifiziert
- Am gebräuchlichsten sind XML Schema Datentypen (XSD)
- Syntax: "*Datenwert*" ^^<*Datentyp-URI*>
- Es gibt nur zwei vordefinierte Datentypen in RDF:
  - *rdf:HTML* und *rdf:XMLLiteral*
- Beispiel:
  - "123" ^^http://www.w3.org/2001/XMLSchema#int

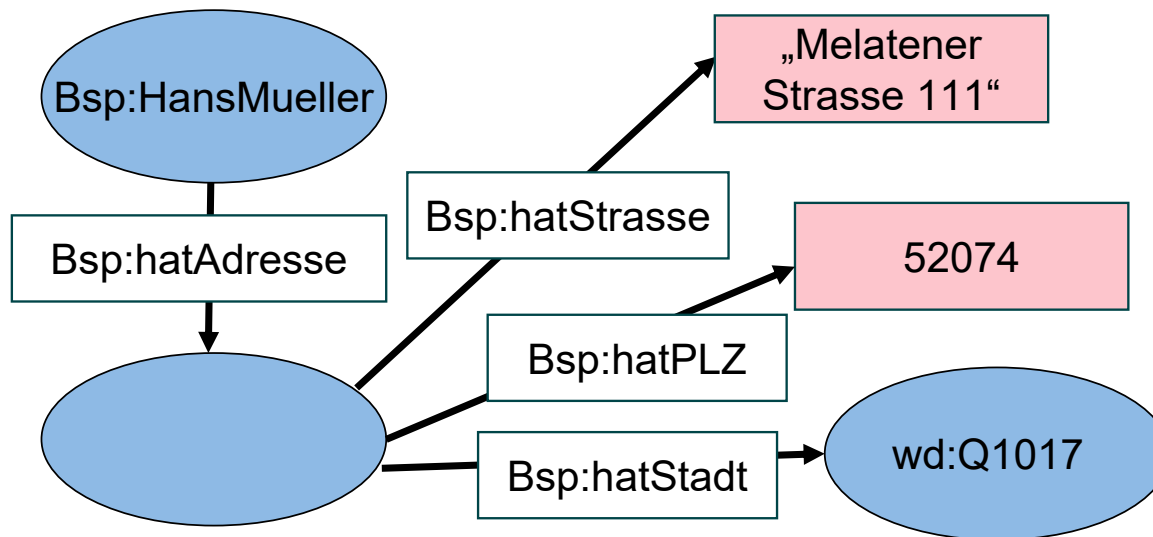
# Blank Nodes

---

- Ein Blank Node hat keinen globalen Identifikator
  - Idee: „Ein Blank Node ist ein Platzhalter“
  - Jeder Blank Node hat eine eindeutige Identität innerhalb eines Graphen
  - Es kann geprüft werden ob zwei Blank Nodes gleich sind
  - Blank Nodes werden als “Existentielle Quantifikatoren” verwendet

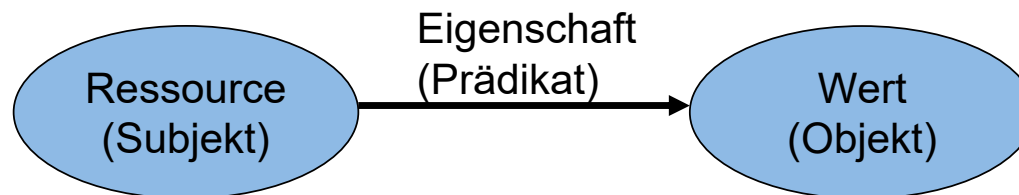
## Beispiel für die Verwendung von Blank Nodes

- “Hans Mueller hat die Adresse „Melatener Strasse 111, 52074 Aachen”



# Übersicht: Regeln für RDF Tripel

	URI	Literal	Blank Node
Subjekt	X		X
Prädikat	X		
Objekt	X	X	X





# Serialisierung von RDF Graphen

# Formate um RDF Graphen zu Serialisieren

---

**Frage:** Wie kann man einen Graphen in einer Datei speichern?

- **Turtle:** Text-Format mit dem Ziel der besseren Lesbarkeit durch Menschen (wird in dieser Vorlesung verwendet)
- **JSON-LD:** W3C Proposed Recommendation. Format um RDF als JSON zu serialisieren, bzw existierendes JSON als RDF zu interpretieren. Empfohlen von Google.
- **N-Triples:** Text-Format welches sich sehr einfach parsen lässt
- **Notation 3 (N3):** alternatives Text-Format mit Nicht-Standard Features die RDF erweitern
- **RDF/XML:** das erste offizielle Format um RDF als XML zu serialisieren
- **RDFa:** Mechanismus um RDF in (X)HTML einzubetten

[1] <https://www.youtube.com/watch?v=cSF48tbsjJw&feature=youtu.be&t=1353>

# Turtle Syntax 1

---

- Turtle steht für “Terse RDF Triple Language”
- Format um RDF Triples als Strings darzustellen
- URIs immer in <>-Klammern:  
`<http://www.wikidata.org/entity/Q1017>`
- Literale in doppelten Anführungszeichen:
  - `"Aachen"@DE`
  - `"51.33332"^^xsd:float`
  - Integer werden als Literale mit dem Integer Datentyp interpretiert: `32`  
als `"32"^^xsd:int`
- Alle Tripel werden als Sätze von Subjekt, Prädikat und Objekt dargestellt, gefolgt von einem Punkt:
  - `<http://www.wikidata.org/entity/Q1017> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .`
- Leerzeichen und Zeilenumbrüche werden außerhalb von Identifikatoren ignoriert



## Turtle Syntax 2

---

- Abkürzungen können für Namespaces definiert werden:
  - @prefix abbr ':' <URI> .
  - **Beispiel:** @prefix wd: <http://www.wikidata.org/entity/> .
- Ohne Namespaces:  
<http://www.wikidata.org/entity/Q1017> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .
- Mit Namespaces:

```
@prefix wd: < http://www.wikidata.org/entity/> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .  
    wd:Q1017 rdfs:label "Aachen"@de .
```

## Turtle Syntax 3

---

- Tripel mit dem selben Subjekt können zusammengefasst werden:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: http://www.w3.org/2000/01/rdf-schema#
@prefix wd: < http://www.wikidata.org/entity/>
@prefix wdt <http://www.wikidata.org/prop/direct/>
```

```
wd:Q1017      rdfs:label    "Aachen"@de ;
              wdt:P6        wd:Q1893149 .
```

- Tripel mit dem gleichen Subjekt und Prädikat können auch zusammengefasst werden:

```
wd:Q1017 rdfs:label „Aachen“@de, „Aix-la-Chapelle“@fr;
          wdt:P6      wd:Q99688800 .
```

# Turtle: Vorteile und Nachteile

---

- Vorteile:
  - Kurzgefasst, daher effiziente Speicherung möglich
  - Einfach für Menschen zu lesen
  - Sehr nah am RDF Datenmodell
- Nachteile:
  - Geringe Unterstützung in Software-Werkzeugen
  - Geringe Verbreitung außerhalb des Semantic Webs

Siehe: <https://www.w3.org/TR/turtle/>



## Beispiele zur Verwendung von RDF Daten

# RDF in freier Wildbahn: Schema.org

---

- 2010 von Google, Yahoo!, Microsoft & Yandex gestartet
- Ziele:
  - Vokabulare welches von allen Suchmaschinen unterstützt wird
  - Vereinfacht den Job des Webmasters und der Suchmaschinen-Optimierung
- Vokabular um Web-Seiten zu annotieren
- Kann Entitäten von den folgenden Typen annotieren:
  - „Creative“
  - „Works“
  - „Events“
  - „Organisations“
  - „Persons“
  - „Places“
  - „Products“
  - ...

# Screenshot von Linked Data Service DNB

DNB - Linked Data Service

dnb.de/EN/Professionell/Metadatendienste/Datenbezug/LDS/lids\_node.html

Deutsch Sign language Simple language

DEUTSCHE NATIONAL BIBLIOTHEK

MENU

DNB FOR USERS DNB PROFESSIONAL

Home > DNB Professional > Metadata Services > Linked Data Service

## LINKED DATA SERVICE

- Overview
- Integrated Authority File (GND)
- Bibliographic data
- Test data
- Subscription Terms and Terms of Use
- Further development and service information
- Frequently asked questions (FAQ)
- Documentation
- Download
- Contact

# Wikidata

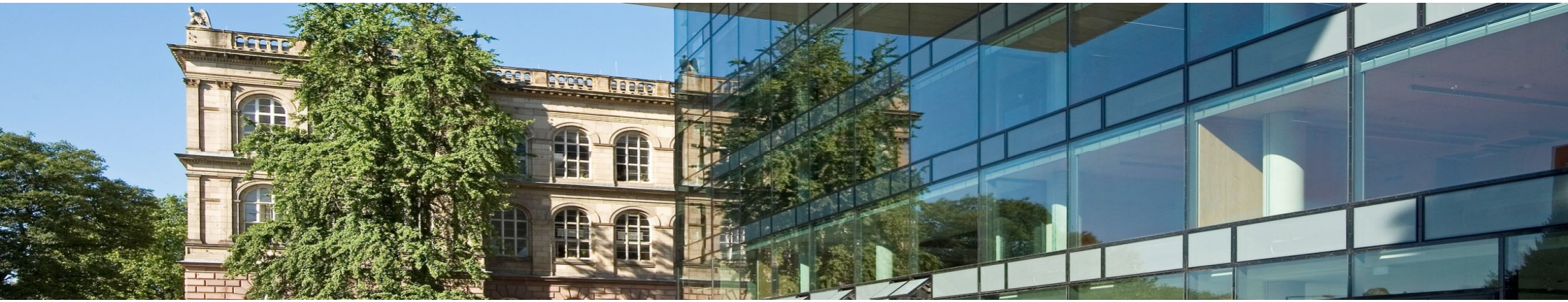
- frei bearbeitbaren Wissensdatenbank mit dem Wikipedia zu unterstützen.
- von Wikimedia Deutschland gestartet als gemeinsame Quelle für Daten
- Globale Datenbank für Identifier
- 82,4 Millionen Datenobjekte vorhanden (Stand März 2020), wird u.a. von Siri und IBM Watson verwendet [1][2]

The image shows a Wikidata entity page for Douglas Adams (Q42). The page is annotated with labels and lines pointing to specific features:

- Bezeichnung:** Points to the name "Douglas Adams (Q42)".
- eindeutiger Bezeichner:** Points to the identifier "(Q42)".
- Beschreibung:** Points to the description "Britischer Schriftsteller" and the alternative name "Douglas Noël Adams | Douglas Noel Adams".
- Alternativbezeichnung:** Points to the alternative name "Douglas Noël Adams | Douglas Noel Adams".
- Eigenschaft:** Points to the "Alma Mater" property.
- Wert:** Points to the value "St John's College".
- Qualifikatoren:** Points to the table of qualifiers for the alma mater, including "Endzeitpunkt" (1974), "Hauptfach im Studium" (Englische Literatur), "akademischer Grad" (Bachelor of Arts), and "Startzeitpunkt" (1971).
- Rang:** Points to the "2 Fundstellen" section.
- Aussagen-gruppe:** Points to the entire "Aussagen" section.
- Fundstelle (offen):** Points to the first citation from Encyclopædia Britannica Online.
- Fundstelle (eingeklappt):** Points to the second citation from Brentwood School.

[1] <https://www.wired.com/story/inside-the-alexa-friendly-world-of-wikidata/>

[2] <https://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext>



# SPARQL: Anfrage-Sprache für RDF Graphen



# SPARQL: Anfrage-Sprache für RDF Graphen

---

- SPARQL steht für “SPARQL Protocol and RDF Query Language” (gesprochen: “Sparkl”)
- <https://www.w3.org/TR/sparql11-query/>

## Struktur einer SPARQL Anfrage

---

```
# prefix declarations
PREFIX ex: <http://example.com/resources/>
....
# query type# projection # dataset definition
SELECT    ?x ?y          FROM ...

# graph pattern
WHERE {
    ?x a ?y
}

# query modifiers
ORDER BY ?y
```

## Bestandteile einer SPARQL Anfrage:

Definition von Namespaces

Anfrage Klausel: Projektion

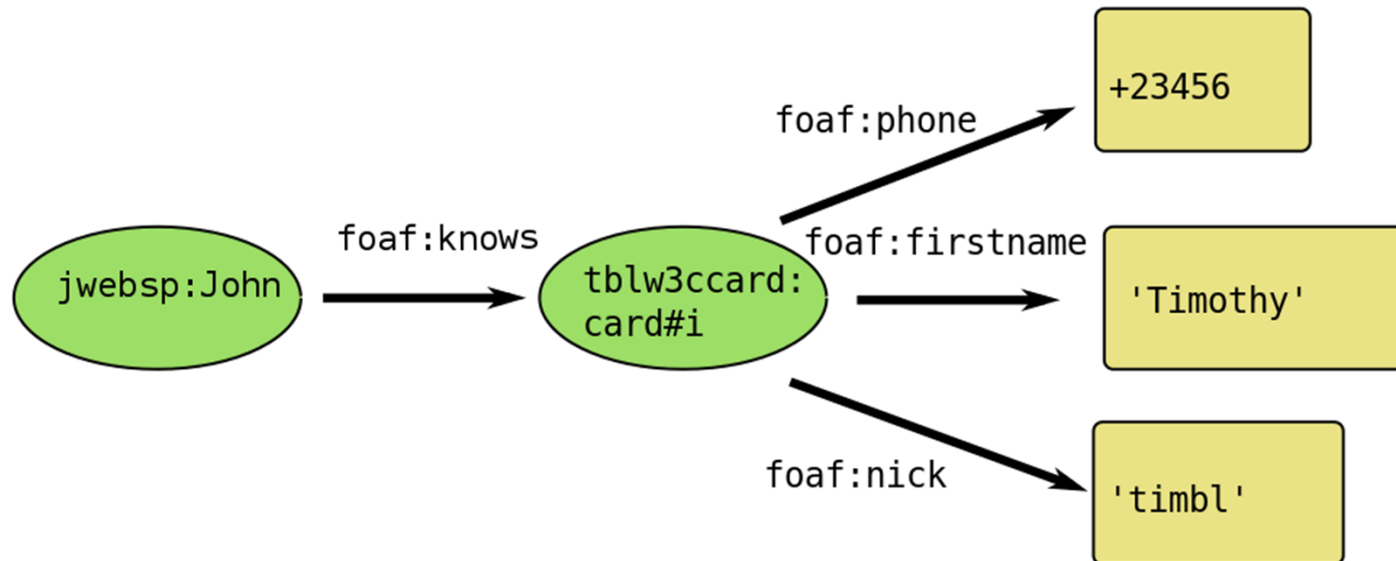
Vier Möglichkeiten: SELECT, ASK,  
CONSTRUCT, DESCRIBE

WHERE Klausel: Selektion durch ein  
Graph-Muster

Anfrage Modifikatoren

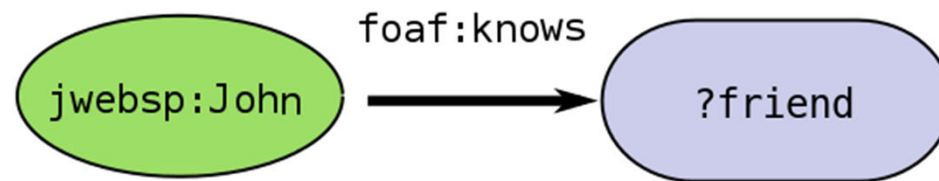
# Ein kleiner Beispiel RDF Graph

```
@prefix wbsp: <http://mywebpace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```

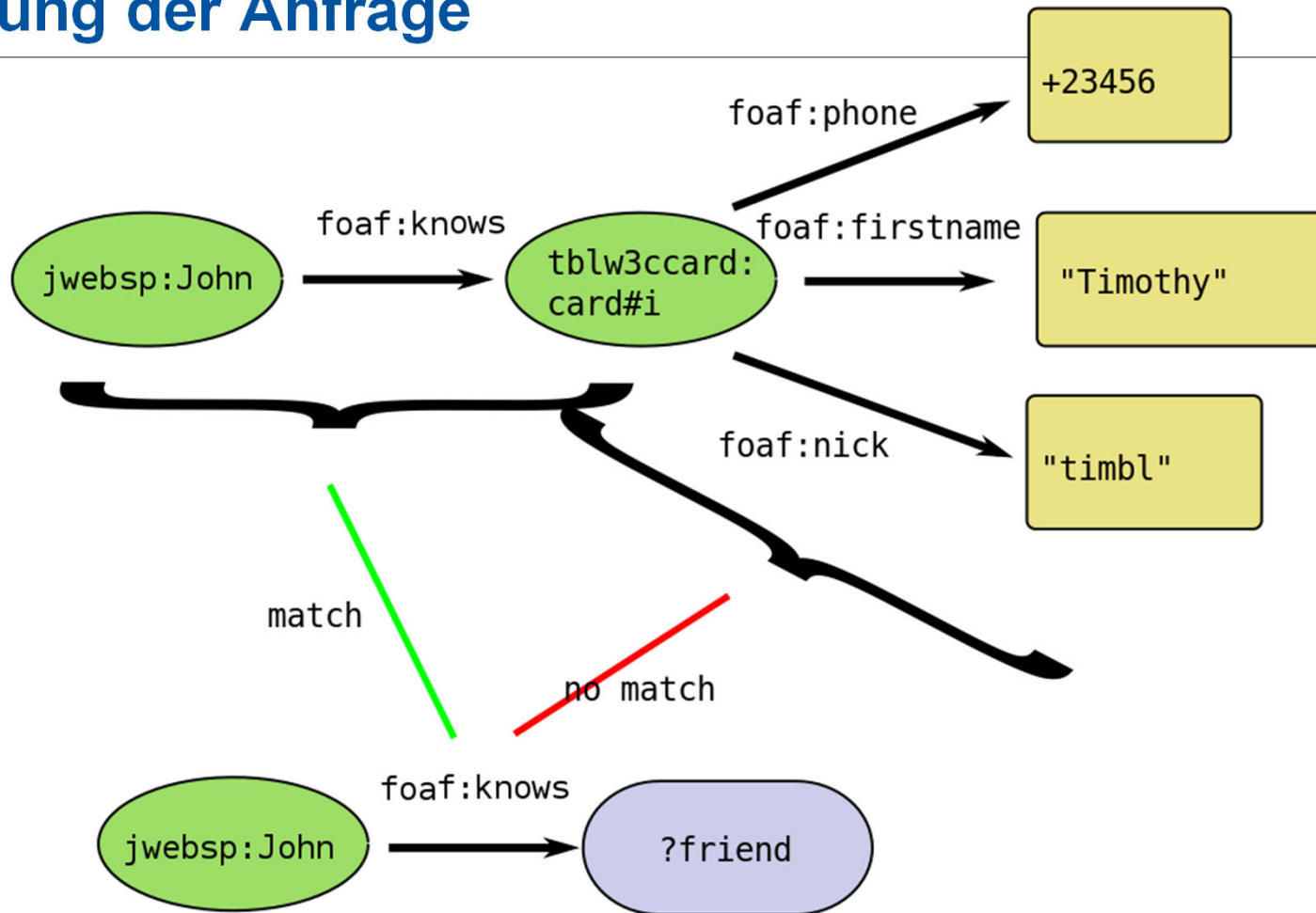


# Eine sehr einfache Anfrage

---



# Beantwortung der Anfrage



# Eine etwas komplexere Anfrage

---



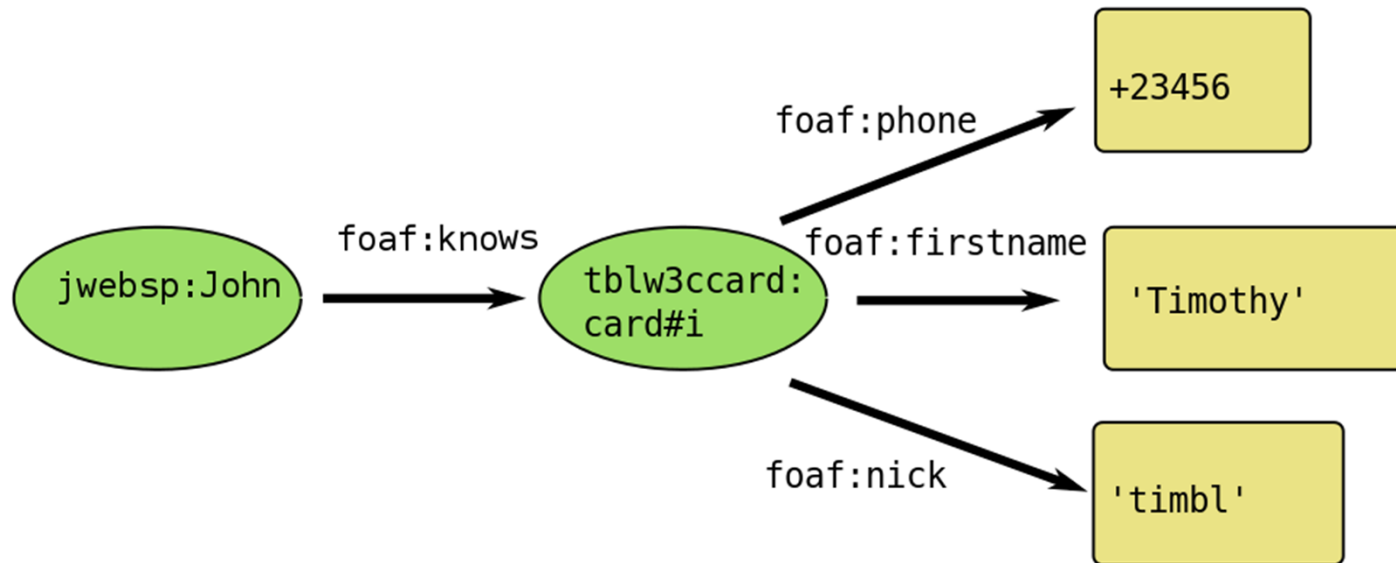
## SPARQL für die Anfrage

---

```
SELECT ?friend ?friendname WHERE {  
  jwebesp:John foaf:knows ?friend .  
  ?friend foaf:firstname ?friendname  
}
```

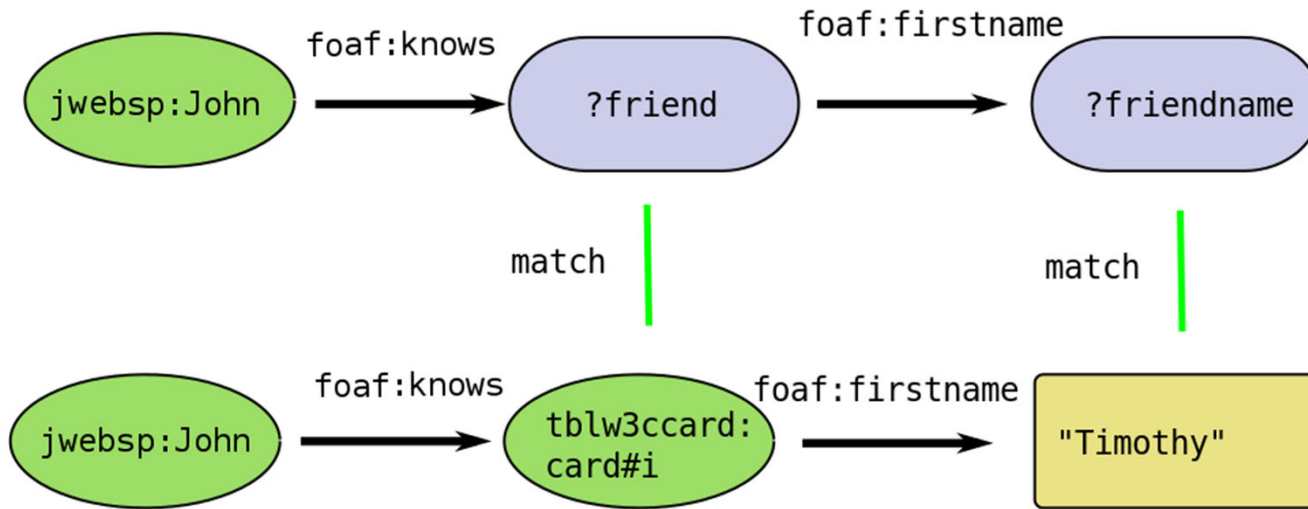
# Ein kleiner Beispiel RDF Graph

```
@prefix wbsp: <http://mywebpace.com/profiles/john/>.  
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>  
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```





# Beantwortung der zweiten Anfrage



?friend		?friendname
-----+		
tblw3ccard:card#i		"Timothy"

# SPARQL Anfrage Möglichkeiten

---

- **SELECT**
  - Ergebnis ist eine Tabelle der Ergebnisse
- **ASK**
  - Ergebnis ist eine boolesche Variable.
  - Wahr, wenn das Graph-Muster wenigstens einmal passt
- **CONSTRUCT**
  - Ergebnis ist das Erzeugen von neuen Tripel nach einem gegebenen Graph-Muster
- **DESCRIBE**
  - Ergebnis ist die Beschreibung von Ressourcen

# FROM Klausel

---

- Spezifiziert welche Graphen für das Ergebnis berücksichtigt werden sollen
- Optional
  - Falls ausgelassen wird der sogenannte *default* Graph verwendet
  - Falls angegeben, werden nur die angegebenen Graphen berücksichtigt
  - Falls ein oder mehrere “named graph(s)” angegeben sind, können diese in der Anfrage referenziert werden

# WHERE Klausel

---

- Enthält die Graph-Muster
- Konjunktiv
- Variablen werden an die Werte aus den Ergebnissen für die Graph-Muster gebunden
  
- Format der Graph-Muster:
  - Wieder die Subjekt / Prädikat / Objekt Form
  - Variablen können an jeder Position vorkommen.

# Anfrage Modifikatoren

---

- Verändert das Ergebnis einer Anfrage
- LIMIT und OFFSET teilen das Ergebnis auf
- ORDER BY ASC / DESC
- Beispiele:
  - `SELECT * WHERE {.....} LIMIT 10`
    - Nur die ersten 10 Ergebnisse werden zurückgeliefert
  - `SELECT * WHERE {.....} ORDER BY  
ASC(...) LIMIT 10`
    - Nur die ersten 10 Ergebnisse werden zurückgeliefert, aufsteigend alphabetisch sortiert.

# Beispiele für Graph-Muster

---

```
:John foaf:knows :Tim .  
:John foaf:name "John" .  
:Tim foaf:knows :John . :Tim foaf:name "Tim" .
```

```
SELECT ?name WHERE { :John foaf:name ?name }  
-->      "John"  
SELECT ?friend WHERE { :John foaf:knows ?friend }  
-->      :Tim  
SELECT ?friend ?name WHERE { :John foaf:knows  
?friend . :John foaf:name ?name }  
-->      :Tim "John"  
SELECT ?friendsname WHERE { :John foaf:knows  
?friend . ?friend foaf:name ?friendsname }  
-->      "Tim"
```

# Graph-Muster: Kartesisches Produkt

---

```
:John foaf:knows :Tim .  
:John foaf:name "John" .  
:Tim foaf:knows :John .  
:Tim foaf:name "Tim" .
```

```
SELECT ?person ?friendsname WHERE {  
    ?person foaf:knows ?friend .  
    ?somebody foaf:name ?friendsname  
}
```

```
:John "John"  
:John "Tim"  
:Tim "John"  
:Tim "Tim"
```

# Ressourcen vergleichen

---

- Vergleichen von URIs:
  - foaf:name == <http://xmlns.com/foaf/spec/name>
- Kein Umwandeln von z.B. reservierten Zeichen
  - myns:John%20Doe != myns:John Doe (gibt einen Fehler)
- Großschreibung beachten!
  - foaf:name != <http://xmlns.com/foaf/spec/Name>



## Literale vergleichen

---

- Literale werden Zeichen für Zeichen verglichen
- Wenn Literale einen Datentyp haben, liegt es im Ermessen der SPARQL Engine den Datentyp zu interpretieren und zu vergleichen
  - Wichtig z.B. bei `xsd:int` , `xsd:date`
- Literale mit Language Tag werde auch Zeichen für Zeichen verglichen

# Filter

---

- Verändern das Ergebnis von Graph-Mustern
- Erlauben es Werte zu testen
- Wichtigste Funktion: Restriktionen von Literal Wertebereichen
  - String Vergleich
  - Reguläre Ausdrücke (regular expressions)
  - Numerische Vergleiche
- Tests der Sprache / des Datentyps von Strings
- Das Ergebnis eines String Tests ist entweder wahr, falsch oder “type error”

# Filter: Übersicht

---

- Logische Filter: ! , && , ||
- Mathematisch: + , - , \* , /
- Vergleichs-Operatoren: = , != , > , < , ....
- Tests bzgl. RDF Datenmodell: isURI, isBlank, isLiteral, str, lang, datatype
- Tests bzgl. SPARQL Resultatmenge: bound
- Andere Operatoren: sameTerm, langMatches, regex

## Filter: Strings

---

- `str()`: Wert eines Literals ohne den Datentypen und Tag
- `contains()`: Suche innerhalb eines Literals
- `regex()`: Einsatz einer vollwertigen regular expression

## Filter: String Beispiel

---

```
:John :age 32 ; foaf:name "John"@en .  
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend {?friend foaf:name "John" . }  
→ empty
```

```
SELECT ?friend {?friend foaf:name "Tim" . }  
→ :Tim
```

```
SELECT ?friend {?friend foaf:name ?name .  
FILTER ( str(?name) = "John")}  
→ :John
```

```
SELECT ?friend {?friend foaf:name ?name .  
FILTER contains(?name, "im")}  
→ :Tim
```

## Filter: Sprache und Datentyp

---

- `lang(?x)` : Zugriff auf den Sprach-Bezeichner eines Literals
- `langMatches(lang(?x), "en")`: prüft ob ein gegebener Sprach-Bezeichner mit einem anderen Sprach-Bezeichner übereinstimmt
- `datatype(?x)` : Zugriff auf den Datentyp eines Literals

# Filtern mit numerischen Operatoren

---

```
:John :age 32 ; foaf:name "John"@en .  
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend WHERE {?friend :age ?age  
FILTER (?age>25) }  
-->      :John
```

# Filtern mit logischen Operatoren

```
:John :age 32 ;foaf:name "John"@en .  
:Tim :age 20 ; foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend WHERE {  
  ?friend foaf:name ?name .  
  ?friend :age ?age .  
  FILTER (str(?name) = "Tim" && ?age>25)}
```

```
--> empty
```

```
SELECT ?friend WHERE {  
  ?friend foaf:name ?name .  
  ?friend :age ?age .  
  FILTER (str(?name) = "Tim" || ?age>25)}
```

```
--> :Tim
```

```
--> :John
```



# Optionale Werte in SPARQL

---

- Ähnlich wie ein left join in SQL
- Erlaubt es unvollständige Daten abzufragen
- “OPTIONAL” bearbeitet ein vollständiges Graph-Muster
- Syntax:
  - { Graph-Muster1 } OPTIONAL { Optionales-Muster }

## Beispiel einer Anfrage mit "OPTIONAL" (1)

---

```
:John foaf:knows :Tim ;
      foaf:name "John"
      foaf:phone "+123456"
:Tim  foaf:knows :John ;
      foaf:name "Tim" .
```

```
SELECT ?name ?phone
WHERE   {?person foaf:name ?name .
        ?person foaf:phone ?phone}
```

→ "John" "+123456"

**Suboptimales Ergebnis?**

## Beispiel einer Anfrage mit “OPTIONAL” (2)

---

```
:John foaf:knows :Tim ;
      foaf:name "John"
      foaf:phone "+123456"
:Tim  foaf:knows :John ;
      foaf:name "Tim" .
```

```
SELECT ?name ?phone WHERE {
  ?person foaf:name ?name .
  OPTIONAL {?person foaf:phone ?phone}}
```

```
--> "John" "+123456"
```

```
--> "Tim"
```

# UNION in Graph-Mustern

---

- **Syntax:** {Graph Muster 1} UNION {Graph Muster 2}
- Erlaubt es mehrere Muster zu kombinieren

# SPARQL Beispiel

---

```
:John rdf:type foaf:Person ; foaf:name "John" .  
:Tim  rdf:type foaf:Person ; foaf:name "Tim"  .  
:Jane rdf:type foaf:Person ; rdfs:label "Jane" .
```

```
SELECT ?name WHERE  
  {?person rdf:type foaf:Person . ?person foaf:name ?name}  
--> "John"  
--> "Tim"
```

```
SELECT ?name WHERE  
  {?person rdf:type foaf:Person .  
    {?person foaf:name ?name} UNION {?person rdfs:label ?name}}  
--> "John"  
--> "Tim"  
--> "Jane"
```

# Projektion

---

```
SELECT ?s ?o WHERE {?s ?p ?o}
```

--> only the variables specified, in this case ?s and ?o

```
SELECT * WHERE {.....}
```

--> all variables mentioned in the graph patterns

```
SELECT DISTINCT .....
```

--> eliminates duplicates in the result

# Count

---

- Eine sehr einfache Funktion um zu zählen wie oft eine Variable durch ein Ergebnis gebunden wird

- **Beispiel:**

```
:John foaf:knows :Tim ; foaf:name "John" .  
:Tim foaf:knows :John ; foaf:name "Tim" .
```

```
SELECT count(?person) {?person foaf:name ?name}  
--> 2
```

# SPARQL Anfragen nach Klassen

---

- Vokabulare definieren Klassen
  - foaf:Person
  - foaf:Dokument
- rdf:type assoziiert eine Instanz mit einer Klasse
  - Wird auch mit „a“ abgekürzt:
  - `:John a foaf:Person == :John rdf:type foaf:Person`





# SPARQL in der Praxis

# Screenshot von Wikidata SPARQL Service

The screenshot shows the Wikidata Query Service interface. The query is as follows:

```
1 #Information about Aachen
2 SELECT ?item ?pred ?itemLabel
3 WHERE
4 {
5   wd:Q1017 ?pred ?item
6   SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
7 }
```

The results table shows 667 results in 4019 ms. The table has four columns: item, pred, itemLabel, and predLabel.

item	pred	itemLabel	predLabel
Aachen	skos:altLabel	Aachen	http://www.w3.org/2004/02/skos/core#altLabel
Aachen	skos:altLabel	Aachen	http://www.w3.org/2004/02/skos/core#altLabel
Aix-la-Chapelle	skos:altLabel	Aix-la-Chapelle	http://www.w3.org/2004/02/skos/core#altLabel
Aix-la-Chapelle	skos:altLabel	Aix-la-Chapelle	http://www.w3.org/2004/02/skos/core#altLabel
آخن	skos:altLabel	آخن	http://www.w3.org/2004/02/skos/core#altLabel
亞琛	skos:altLabel	亞琛	http://www.w3.org/2004/02/skos/core#altLabel

# SPARQL Anfragen stellen

---

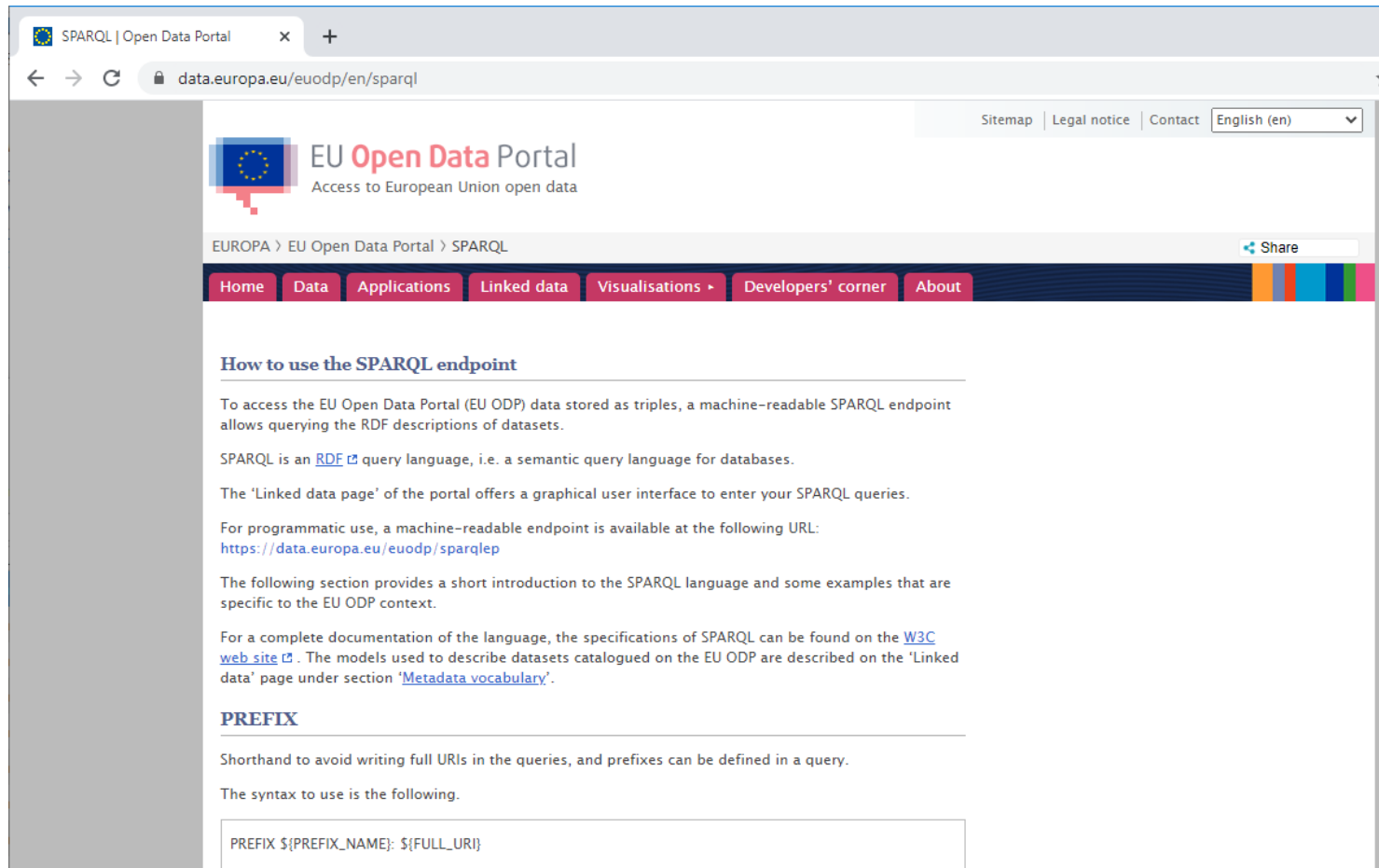
- Benutzte SPARQL um mehr über Entitäten herauszufinden mit „DESCRIBE“

```
#Information about Aachen  
Describe wd:Q1017
```

- Benutzte SPARQL um mehr über Entitäten herauszufinden

```
SELECT ?p ?o ?ol  
WHERE {  
  wd:Q1017 ?p ?o .  
  OPTIONAL { ?o rdfs:label ?ol }  
}
```

# EU Open Data Portal



The screenshot shows a web browser window with the address bar displaying `data.europa.eu/euodp/en/sparql`. The page header includes the EU Open Data Portal logo and navigation links for Sitemap, Legal notice, Contact, and a language dropdown set to English (en). A breadcrumb trail shows the path: EUROPA > EU Open Data Portal > SPARQL. A navigation menu contains links for Home, Data, Applications, Linked data, Visualisations, Developers' corner, and About. The main content area is titled "How to use the SPARQL endpoint" and contains the following text:

To access the EU Open Data Portal (EU ODP) data stored as triples, a machine-readable SPARQL endpoint allows querying the RDF descriptions of datasets.

SPARQL is an [RDF](#) query language, i.e. a semantic query language for databases.

The 'Linked data page' of the portal offers a graphical user interface to enter your SPARQL queries.

For programmatic use, a machine-readable endpoint is available at the following URL:  
<https://data.europa.eu/euodp/sparqlp>

The following section provides a short introduction to the SPARQL language and some examples that are specific to the EU ODP context.

For a complete documentation of the language, the specifications of SPARQL can be found on the [W3C web site](#). The models used to describe datasets catalogued on the EU ODP are described on the 'Linked data' page under section '[Metadata vocabulary](#)'.

**PREFIX**

Shorthand to avoid writing full URIs in the queries, and prefixes can be defined in a query.

The syntax to use is the following.

```
PREFIX ${PREFIX_NAME}: ${FULL_URI}
```



# Standards für Vokabulare und Ontologien

# Standards für das Web

---

## 1. RDF – Resource Description Framework

- Ein Graph-basiertes Datenformat: Knoten und Kanten
- Objekte eindeutig identifiziert: URIs
- Information wird verknüpft

## 2. Vokabulare und Ontologien

- Ermöglicht das gemeinsame Verständnis für eine Domäne
- Organisieren von Wissen in einem maschinenlesbaren Format
- Gibt Daten einen auswertbare Bedeutung



# RDF Schema

---

- Was ist RDF Schema?
- Wir können mit RDF Tripeln Fakten ausdrücken:
  - `ex:AlbertEinstein ex:discovered ex:TheoryOfRelativity`
- **Aber wie kann man solche Fakten definieren?**
- Wie können wir definieren das das Prädikat `ex:discovered` eine Person als Subjekt und eine Theorie als Objekt hat?
- Wie können wir die Tatsache ausdrücken, das Albert Einstein ein Forscher war und das jeder Forscher ein Mensch ist?
  
- Solches Wissen wird *schematisches Wissen* genannt
  - Englisch: *schema knowledge* oder *terminological knowledge*
- **RDF Schema** erlaubt es uns dieses Wissen als Model auszudrücken

# RDF Schema (abgekürzt: RDFS)

---

- Von der W3C standardisiert, mit dem Status einer „recommendation“
- Ist selbst ein RDF Vokabular, deswegen ist jedes RDF Schema auch ein RDF Graph
- Das RDF Schema Vokabular ist generisch und nicht an einen bestimmten Anwendungsbereich gebunden
- Erlaubt es die Semantik von RDF Vokabularen zu definieren, welche von Anwendern erstellt wurden (und nicht von der W3C)
- Der Namespace von RDF Schema ist:
  - `http://www.w3.org/2000/01/rdf-schema#`
- Normalerweise wird das Präfix als `rdfs` abgekürzt



# Klassen in RDF Schema

---

Eine *Klasse* ist eine Menge von Dingen oder Entitäten. In RDF sind diese Dinge mit URIs identifiziert

Die Mitgliedschaft einer Entität in einer Klasse ist definiert durch das **rdf:type** Prädikat.

Die Tatsache das `ex:MyBlueVWGolf` ein Mitglied / eine Instanz der Klasse `ex:Car` ist, kann wie folgt ausgedrückt werden:

```
ex:MyBlueVWGolf rdf:type ex:Car .
```

Eine Entität kann eine Instanz mehrerer Klassen sein.

```
ex:MyBlueVWGolf rdf:type ex:Car .  
ex:MyBlueVWGolf rdf:type ex:GermanProduct .
```

# Hierarchien von Klassen

---

- Klassen können mit dem **rdfs:subClassOf** Prädikat in Hierarchien gegliedert werden
- Jede Instanz von *ex:Car* ist auch ein *ex:MotorVehicle*

```
ex:Car      rdfs:subClassOf      ex:MotorVehicle .
```

# Implizites Wissen 1

---

- Aus der Schema Definition folgt auch implizites Wissen

```
ex:MyBlueVWGolf      rdf:type          ex:Car .  
ex:Car               rdfs:subClassOf  ex:MotorVehicle .
```

- Daraus folgt als logische Konsequenz das folgende Statement:

```
ex:MyBlueVWGolf      rdf:type          ex:MotorVehicle .
```

## Implizites Wissen 2

---

- Die folgenden Tripel

```
ex:Car          rdfs:subClassOf    ex:MotorVehicle .
ex:MotorVehicle rdfs:subClassOf    ex:Vehicle .
```

- Implizieren das folgende Statement als eine logische Konsequenz:

```
ex:Car          rdfs:subClassOf    ex:Vehicle
```

- Wir sehen also das *rdfs:subClassOf* **transitiv** ist

## Wir definieren uns eine Klasse

---

- Jede URI welche eine Klasse kennzeichnet, ist eine Instanz von **rdfs:Class**.
- Um eine eigene Klasse zu definieren ist folgendes Tripel notwendig:

```
ex:Car          rdf:type          rdfs:Class .
```

- Darüber hinaus gilt, das *rdfs:Class* selbst eine Instanz von *rdfs:Class* ist:

```
rdfs:Class     rdf:type          rdfs:Class .
```

# Äquivalenz von Klassen

---

- Um auszudrücken das zwei Klassen äquivalent sind, können folgende Tripel benutzt werden:

```
ex:Car          rdfs:subClassOf      ex:Automobile .  
ex:Automobile  rdfs:subClassOf      ex:Car .
```

- Woraus das folgende Tripel folgt:

```
ex:Car rdfs:subClassOf ex:Car .
```

- Daraus folgt auch das *rdfs:subClassOf* reflexiv ist

# Vordefinierte RDFS Klassen

---

Neben `rdfs:Class` sind auch andere Klassen vordefiniert:

- **`rdfs:Resource`** ist die Klasse aller Dinge. Es ist die Superklasse aller anderen Klassen.
- **`rdf:Property`** ist die Klasse aller Prädikate.
- **`rdf:Datatype`** ist die Klasse aller Datentypen, und jede Instanz dieser Klasse ist eine Subklasse von *`rdfs:Literal`*.
- **`rdfs:Literal`** ist die Klasse aller Literale. Jedes Literal mit einem Datentyp ist auch eine Instanz von *`rdfs:Datatype`*.
- **`rdf:langString`** ist die Klasse aller Literale mit einem Sprachbezeichner. Die Klasse ist selbst eine Instanz von *`rdfs:Datatype`* und eine Subklasse von *`rdfs:Literal`*.
- **`rdf:XMLLiteral`** ist die Klasse der XML Literale. Es ist eine Subklasse von *`rdfs:Literal`* und eine Instanz von *`rdfs:Datatype`*.
- **`rdf:Statement`** ist die Klasse der RDF Tripel. Jedes RDF Tripel ist eine Instanz dieser Klasse, mit den Eigenschaften *`rdf:subject`*, *`rdf:predicate`* und *`rdf:object`*.

## Wir definieren uns ein Prädikat

---

- So wie Klassen definiert werden, werden auch Prädikate definiert:

```
ex:drives rdf:type rdf:Property .
```

- Mit diesem neuen Prädikat können wir ausdrücken das Max einen bestimmten VW Golf fährt:

```
ex:Max ex:drives ex:MyBlueVW Golf .
```



# Hierarchische Prädikate

---

- Mit `rdfs:subPropertyOf` kann eine Hierarchie von Prädikaten definiert werden:

```
ex:drives    rdfs:subPropertyOf    ex:controls .
```

- Zusammen mit dem Statement:

```
ex:Max       ex:drives             ex:MyBlueVWGolf .
```

- Folgt daraus:

```
ex:Max       ex:controls           ex:MyBlueVWGolf .
```

## Range und Domain für Prädikate

---

- Jedes Prädikat hat Domain und Range, welche spezifizieren zu welcher Klasse das Subjekt und das Objekt des Tripels gehören müssen.

```
ex:Max      ex:drives  ex:MyBlueVWGolf .
^^^^^^          ^^^^^^^^^^^^^^^^^^^
Domain                Range
```

- Definiert mit **rdfs:domain** und **rdfs:range**

```
ex:drives rdfs:domain  ex:Person .
ex:drives rdfs:range  ex:Vehicle .
```

- Das gleiche für Datentypen:

```
ex:hasAge rdfs:range xsd:nonNegativeInteger .
```

# Die Semantik von *Domain* und *Range*

---

- Hinweise zur Semantik von *Domain* und *Range*:
- “Weil *ex:MyBlueVWGolf* mit *ex:drives* verwendet wurde, wissen wir das es ein *ex:Vehicle* ist, zusätzlich zu allen anderen Klassenzugehörigkeiten die es haben mag.”

# Prädikate mit mehreren *Range* Zugehörigkeiten

---

- Aus den folgenden Aussagen

```
ex:drives    rdfs:range    ex:Car .  
ex:drives    rdfs:range    ex:Ship .
```

- Schließen wir das der *Range* von *ex:drives* sowohl ein *ex:Car* als auch ein *ex:Ship* ist, also **beides!**
- Eine bessere Möglichkeit um auszudrücken das das Objekt eines Tripels sowohl ein Auto als auch ein Schiff sein **kann**, ist wie folgt:

```
ex:Car       rdfs:subClassOf    ex:Vehicle .  
ex:Ship      rdfs:subClassOf    ex:Vehicle .  
ex:drives    rdfs:range         ex:Vehicle .
```

# Implizites Wissen aus *Domain* und *Range*

---

- Sobald wir domain und range definiert haben, müssen wir auf unbeabsichtigte Folgen achten.
- Aus diesem Schema

```
ex:isMarriedTo    rdfs:domain    ex:Person .  
ex:isMarriedTo    rdfs:range     ex:Person .  
ex:RWTH           rdf:type       ex:Institution .
```

- und dem zusätzlichen Statement:

```
ex:Max            ex:isMarriedTo  ex:RWTH .
```

- Folgt als logische Konsequenz:

```
ex:RWTH           rdf:type       ex:Person .
```

# Reifikation 1

---

- Wie kann man in RDF die folgende Information ausdrücken?
  - “Der Kommissar vermutet das der Butler den Gärtner getötet hat.”

```
ex:Detective    ex:supposes    "The butler killed the gardener" .  
ex:Detective    ex:supposes    ex:theButlerKilledTheGardener .
```

- Beide Varianten sind nicht zufriedenstellend.
- Was wir wirklich wollen, ist direkt über diese Tripel Aussagen zu machen:

```
ex:Butler    ex:killed    ex:Gardener .
```

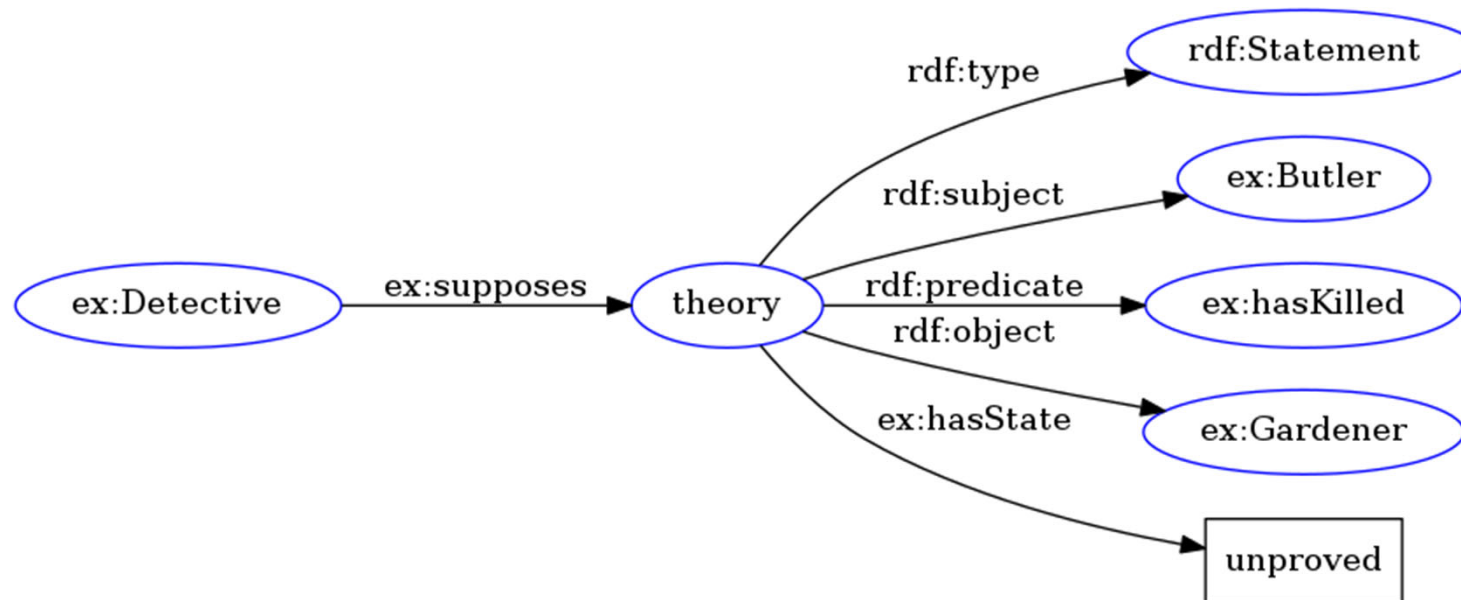
## Reifikation 2

---

- Mit der Klasse `rdf:Statement` erlaubt RDF Schema in einem Tripel eine Aussage über ein anderes Tripel zu machen.
- Dazu werden die folgenden Prädikate verwendet:
  - **`rdf:subject`** definiert eine *`rdfs:Resource`*, die das Subjekt eines Tripels ist
  - **`rdf:predicate`** definiert eine *`rdf:Property`*, welche das Prädikat eines Tripels ist
  - **`rdf:object`** definiert eine *`rdf:Resource`* welche ein Objekt eines Tripels ist
- Das folgende Beispiel zeigt wie ein RDF Tripel als Ressource beschrieben wird, und wie Aussagen über das Tripel gemacht werden (z.B. das die Theorie noch nicht bewiesen wurde).

```
ex:Detective      ex:supposes      :theory .
:theory          rdf:type          rdf:Statement .
:theory          rdf:subject      ex:Butler .
:theory          rdf:predicate    ex:hasKilled .
:theory          rdf:object       ex:Gardener .
:theory          ex:hasState      "unproved" .
```

# Reifikation Beispiel als Graph



Namespaces:  
rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
ex: <http://www.example.org/>  
<http://www.example1.org/>



## Reifikation 3

---

- Bitte Beachten Sie das die folgende Aussage **nicht** aus dem vorherigen Beispiel hervorgeht:

```
ex:Butler    ex:hasKilled    ex:Gardener .
```

- Das erlaubt es in RDF Aussagen zu machen über andere Aussagen, welche falsch oder unbewiesen sind.

## Zusätzliche Informationen

---

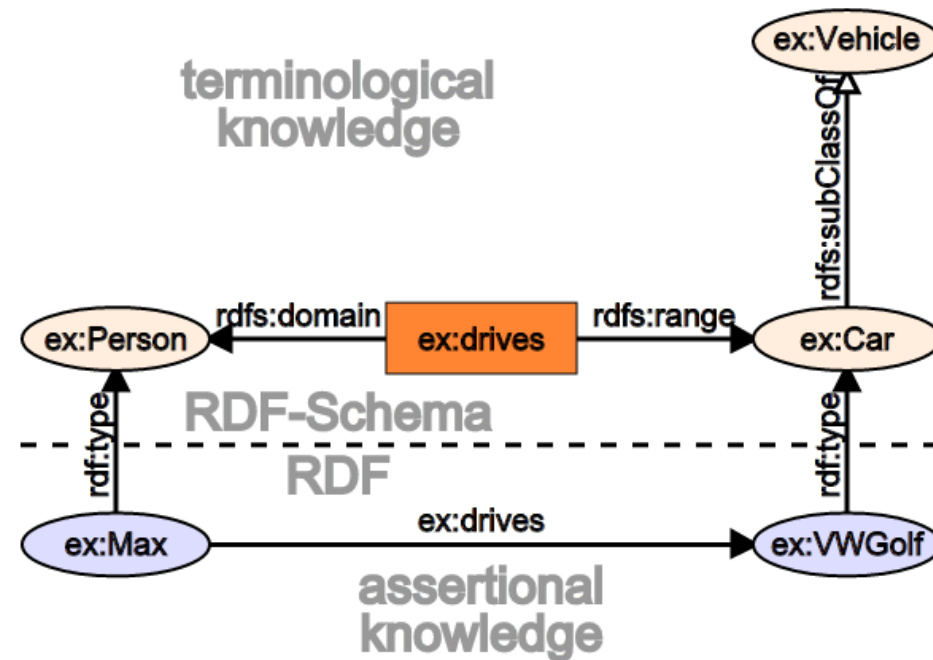
- RDF Schema erlaubt es weitere Informationen über eine Ressource mit den folgenden Prädikaten auszudrücken:
- **rdfs:label** gibt einer Ressource einen Namen (“human readable name”)
- **rdfs:comment** gibt einer Ressource eine Erklärung oder einen Kommentar.
- **rdfs:seeAlso** gibt eine URI an, bei der weitere Informationen zu einer Ressource gefunden werden können.
- **rdfs:isDefinedBy** gibt eine URI an, welche eine Ressource definiert. (**rdfs:isDefinedBy** ist eine Subproperty von **rdfs:seeAlso**).

```
ex:VWGolf    rdfs:label    "VW Golf" .
ex:VWGolf    rdfs:comment  "The VW Golf is a popular german car..." .
ex:VWGolf    rdfs:seeAlso  http://www.wikipedia/VW_Golf .
ex:VWGolf    rdfs:isDefinedBy http://www.Volkswagen.de/ex2:Vw_golf .
```

- Der Vorteil diese Prädikate zu verwenden, ist, dass die zusätzliche Information auch als RDF ausgedrückt wird.

# Vergleich: RDF und RDF Schema

- RDF Schema ist RDF zusammen mit einem Vokabular um Wissen auszudrücken (schematisches Wissen).
- Das Diagramm zeigt die verschiedenen konzeptuellen Ebenen.



# Einschränkungen von RDFS

---

- RDF Schema kann als leichtgewichtige Sprache zur Definition von Vokabularen und Ontologien verwendet werden.
- Jedoch hat RDF Schema auch einige Einschränkungen bzgl. der Definition von Vokabularen und Ontologien.
- RDF Schema erlaubt es nicht die folgenden Sachverhalte auszudrücken:
  - **Negationen von Ausdrücken:** z.B. die Domain eines Prädikats darf eine bestimmte Klasse nicht enthalten.
  - **Keine Einschränkung der Kardinalität:** z.B. zwischen 0 und 1  
*example:isMarriedTo* Prädikate pro Person.
  - **Keine Mengen von Klassen:** Es ist nicht möglich auszudrücken, das eine Domain auf mehrere Klassen zutrifft. Dann benötigen wir eine neue Superklasse.
  - **Keine Metadaten für ein Schema:** z.B. keine Versionsnummer.

# Zusammenfassung RDF Schema

---

- RDF Schema drückt **Wissen** durch die **Definition von Klassen und Prädikaten** aus (schematisches Wissen)
- Klassen und Prädikate (“properties”) können in **Hierarchien** angeordnet werden.
- Für Prädikate können **Domain** (Einschränkung des Subjekts) und **Range** (Einschränkung des Prädikats) definiert werden
- Ein Schema erlaubt es auf **implizit definiertes Wissen** zu schließen (“inference of implicit knowledge”).
  
- RDF Schema kann für “**leichtgewichtige**” **Vokabulare** und Ontologien verwendet werden, ist aber nicht so mächtig wie z.B. OWL. (OWL wird **nicht** in dieser Vorlesung behandelt.)

## Zusammenfassung: Semantic Web

---

- RDF: das Datenmodell des Semantic Web
- Serialisierung von RDF Graphen
- Beispiele zur Verwendung von RDF Daten
- SPARQL: Anfrage-Sprache für RDF Graphen
- SPARQL in der Praxis
- Standards für Vokabulare und Ontologien im Semantic Web