Alternative Datenmodelle

1. XML als semistrukturiertes Datenmodell
2. RDF und Semantic Web
3. Graph-Datenbanken

Alternative Datenmodelle

1. **XML als semistrukturiertes Datenmodell**
2. RDF und Semantic Web
3. Graph-Datenbanken

Informatik 5 -
Informationssysteme
und Datenbanken

RWTH AACHEN UNIVERSITY

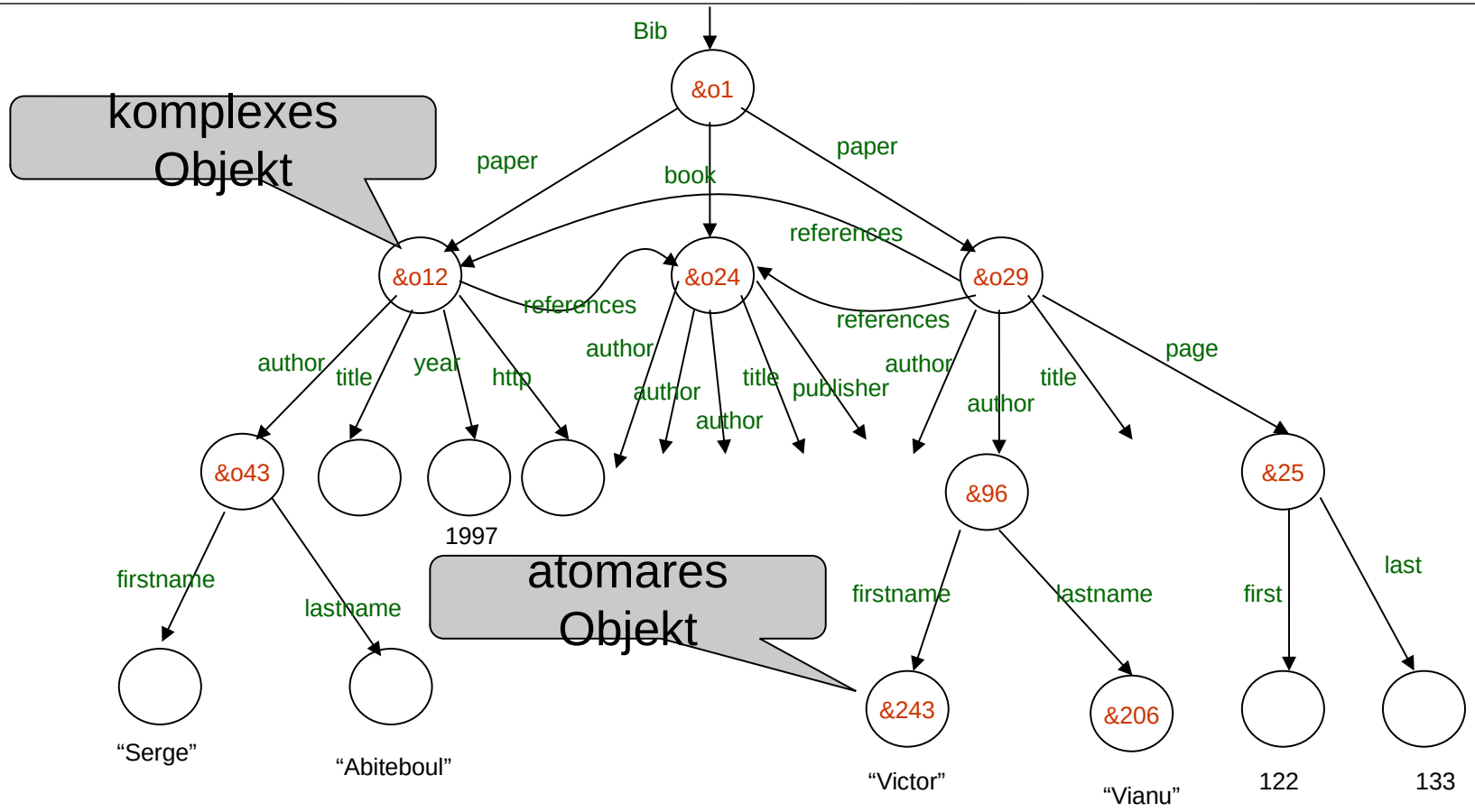
Strukturierte und Unstrukturierte Datenmodelle

- Das **relationale Datenmodell** legt die Struktur der abgespeicherten Daten fest
  - Schema erforderlich
  - ➢ Strukturiertes Datenmodell
- Die **HyperText Markup Language** (HTML) ist eine reine Formatierungssprache bei der Tags formatierungstechnisch interpretiert werden, sie haben keine semantische Bedeutung
  - Einem HTML Dokument ist kein Schema zugeordnet
  - ➢ Unstrukturiertes Datenmodell
- Beispiel: Informationen aus Universitätsdatenbank in HTML

```
<UL>
    <LI>Curie</LI>
    <LI>Sokrates</LI>
</UL>
<UL>
    <LI>Mäeutik</LI>
    <LI>Bioethik</LI>
</UL>
```

  - nur durch Kontextwissen als Professoren- bzw. Vorlesungsliste erkennbar

Semistrukturiertes Datenmodell (1)

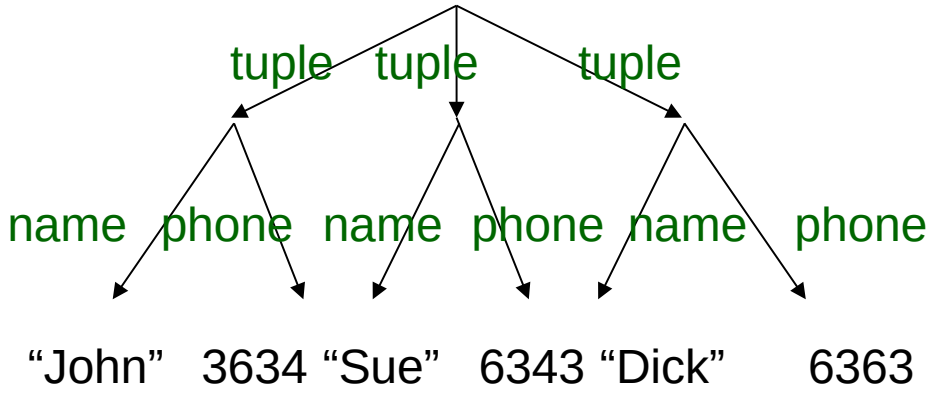


Object Exchange Model (OEM)

[Papakonstantinou et al., ICDE 1995]

Relationale Daten im semistrukturierten Datenmodell

name	phone
John	3634
Sue	6343
Dick	6363



eXtensible Markup Language

- Die **eXtensible Markup Language** (XML) wurde vom W3C konzipiert und standardisiert zur Ergänzung von HTML
- XML ist ein **semistrukturiertes Datenmodell**
  - Selbstbeschreibend, irregulär
  - Daten folgen einem fest vorgegebenen Schema,
  - beinhalten jedoch auch Elemente die nicht dem Schema unterliegen.
- In XML werden kontext- bzw. anwendungsspezifische Tags verwendet, welche die Bedeutung der Elemente angeben (und nicht die Formatierung/Struktur)
- Beispiel: Informationen aus Universitätsdatenbank in XML

```
<Professoren>
                    <ProfessorIn>Curie</ProfessorIn>
                    <ProfessorIn>Sokrates</ProfessorIn>
</Professoren>
<Vorlesungen>
                    <Vorlesung>Mäeutik</Vorlesung>
                    <Vorlesung>Bioethik</Vorlesung>
</Vorlesungen>
```

Ursprünge und Verwendung

- XML als Beschreibungssprache für Dokumente
  (Nachfolger von HTML, SGML, ...)
  - Daher: Flexibilität, Selbstbeschreibend
  - Anfragen: Transformation von Dokumenten, regelbasiert **XSLT**

- XML als Datenaustauschformat
  (als Ersatz für ASCII- oder Binärformate)
  - Semistrukturierte Daten
  - Schema: **XML Schema**
  - Anfragen: **XPath**, **XQuery**

Struktur eines XML Dokuments

- Ein XML Dokument besteht aus zwei Teilen:
  - Einer optionalen Präambel (gibt u.a. die XML Version an)
  - Einem einzigen Wurzelelement, das beliebig viele und tief geschachtelte Unterelemente beinhalten kann

- Beispiel:

```
<?xml version='1.0' encoding='UTF-8'?>

<UniDaten>
        <Professoren>
                <ProfessorIn>Curie</ProfessorIn>
                <ProfessorIn>Sokrates</ProfessorIn>
        </Professoren>
        <Vorlesungen>
                <Vorlesung>Mäeutik</Vorlesung>
                <Vorlesung>Bioethik</Vorlesung>
        </Vorlesungen>
</UniDaten>
```

Präambel

Wurzelelement

Geschachtelte Elemente

XML Terminologie zur Syntax

- Tags: Buch, Titel, Autor, …
- Anfangs-Tag: <Buch>, Ende-Tag: </Buch>
- Elemente: <Buch>…</Buch>, <Autor>…</Autor>
- Elemente sind verschachtelt: Elemente und Subelemente
- Leere Elemente: <ISBN></ISBN>, abgekürzt: <ISBN/>
- Attribute <Buch Jahr='2001'/>
- Verarbeitungshinweise *(*Processing instructions*)* <?… ?>
- Kommentare <!-- … -->

Informationen werden in XML folgendermaßen gespeichert:

- Durch Vorhandensein von Elementen
- Durch Attribute von Elementen
- Durch Inhalte der Elemente

Wohlgeformtheit eines XML-Dokuments

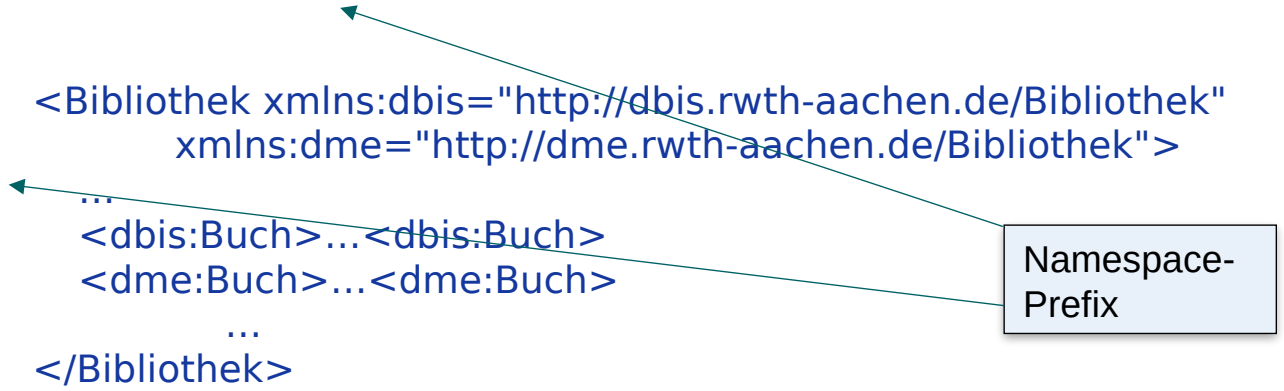
- Ein XML Dokument ohne Schema heißt **wohlgeformt**, falls es die syntaktischen Anforderungen erfüllt, d.h.:
  - Korrekter Prolog/Präambel, gefolgt von einem Wurzel-Element
  - Jedes Element beinhaltet Kind-Elemente, Daten oder ist leer
  - Elemente sind baumartig innerhalb des Wurzelelements verschachtelt
  - Elementaufbau ist korrekt, d.h.
    - einem Anfangs-Tag `<elementName>` folgt
      ein Ende-Tag `</elementName>`
    - Tags verschiedener Elemente überlappen sich nicht (korrekte Schachtelung)
  - Attribute
    - Anfangs-Tags eines Elements können Attribute aufweisen
    - Jedes Attribut wird max. einmal pro Element verwendet (eindeutig innerhalb des Tags)

XML Namespaces

- Um Elemente z.B. aus unterschiedlichen Quellen voneinander abzugrenzen, ist möglich jedem Element ein Namespace zu zuweisen:

  <Bibliothek **xmlns="http://dbis.rwth-aachen.de/Bibliothek"**>…</ Bibliothek > (Nur Identifier – nicht zwangsläufig Download-URL)

- Ein Namespace kann durch die Definition eines Namespace-Prefix verkürzt verwendet werden:



  - Der Prefix ist in allen Unterelementen verwendbar.

XML Schema (XSD)

W3C Recommendation: https://www.w3.org/XML/Schema

- Definiert Struktur von XML-Dokumenten in Form eines wohlgeformten XML Dokuments
  - Nutzbar zur Validierung von ausgetauschten Dokumenten
  - Insgesamt trotzdem mehr Flexibilität als im relationalen Datenmodell

- üblicher Namespace-Prefix `xs:`
- Stellt 19 übliche primitive Datentypen (z.B. `xs:string`) und 25 abgeleitete Datentypen (z.B. `xs:ID`) bereit
- Erlaubt die Definition von komplexen Typen / Elementstrukturen mit
  - festgelegter Reihenfolge (`xs:sequence`)
  - Auswahl von Alternativen (`xs:choice`)
  - Definierter Anzahl (`minOccurs`, `maxOccurs`)

- Alternativen sind DTD und RELAX-NG

XML Schema Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
        xmlns="http://dbis.rwth-aachen.de/Bibliothek"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://dbis.rwth-aachen.de/Bibliothek"
        elementFormDefault="qualified">

 <xs:complexType name="BuchType">
        <xs:sequence>
                <xs:element name="Titel" type="xs:string" />
                <xs:element name="Autor" type="xs:string" />
                <xs:element name="Erscheinungsdatum" type="xs:date" />
        </xs:sequence>
        <xs:attribute name="ISBN" type="xs:string" />
</xs:complexType>

<xs:complexType name="BibliothekType">
        <xs:sequence>
                <xs:element name="Buch" type="BuchType"  minOccurs="0"


maxOccurs="unbounded" />
        </xs:sequence>
</xs:complexType>
<xs:element name="Bibliothek" type="BibiliothekType"/>
</xs:schema>
```

Namespace der von diesem Schema definiert wird

definiert Typ *Buch*

definiert Typ *Bibliothek*

Wurzelelement

XML Information Set

- Das **Information Set** eines XML-Dokuments besteht aus **Informationseinheiten** („information items“) und wird verwendet zur abstrakten Beschreibung von Teilen des XML-Dokuments

- Idee:
  - Information Set = “Baum”
  - Informationseinheit = “Knoten”

- Der XML Information Set-Standard definiert:
  - 11 verschiedene Informationseinheitstypen
  - Eigenschaften dieser Typen
  - Gültige Eltern/Kind-Beziehungen zwischen Informationseinheiten

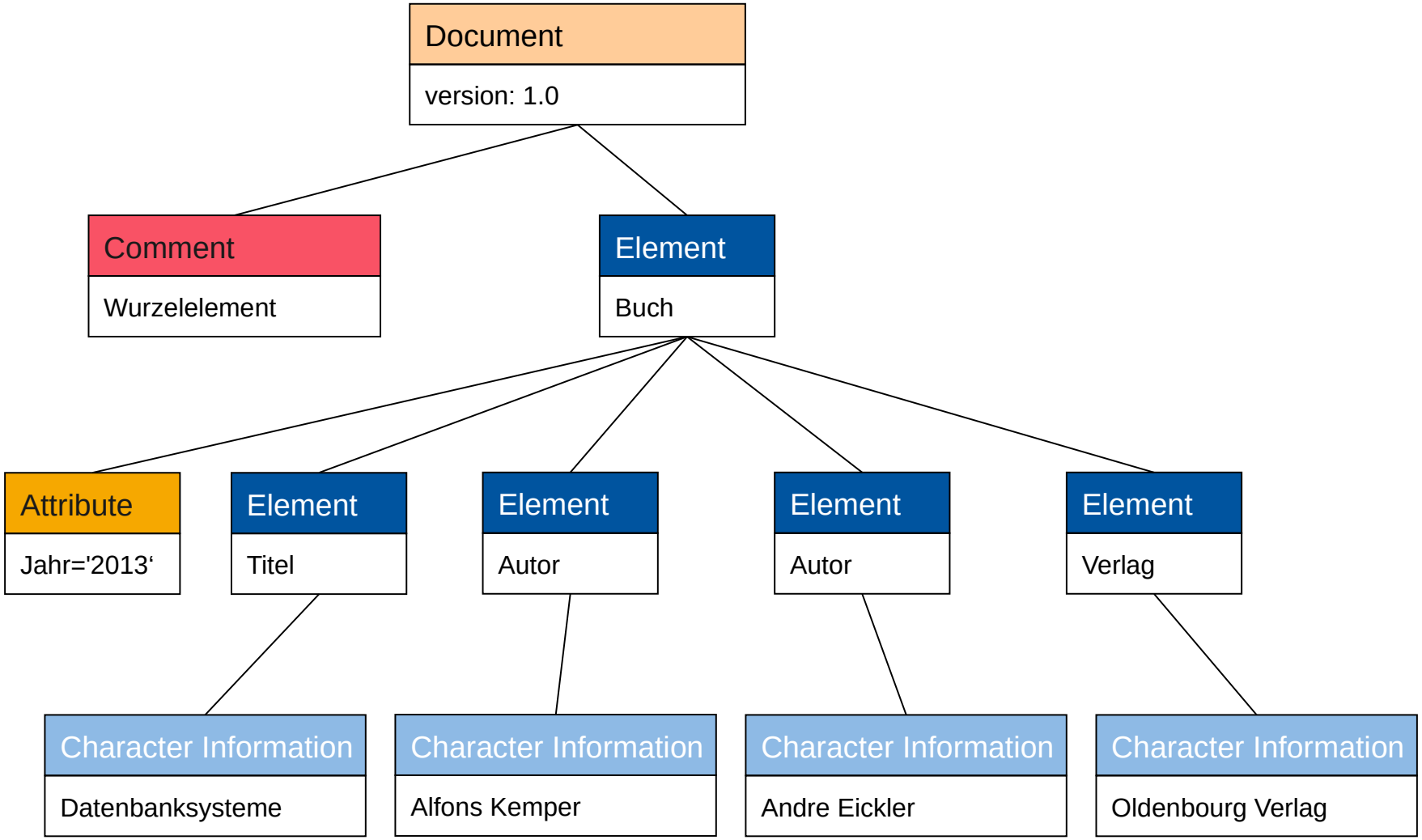
XML Information Set: Informationseinheiten

- Ein XML-Dokument besitzt genau eine Informationseinheit des Typs *document*

- Die Informationseinheit *document* hat als Kinder
  - eine Informationseinheit des Typs *element*,
  - opt. Informationseinheiten des Typs *processing instruction*,
  - opt. Informationseinheiten des Typs *comment*,
  - opt. eine Informationseinheit des Typs document type definition,
  - …

- Zu jedem Element im XML-Dokument existiert eine Informationseinheit des Typs *element*

XML Information Set: XML-Dokument als Baum



Anfrage- und Transformationssprachen

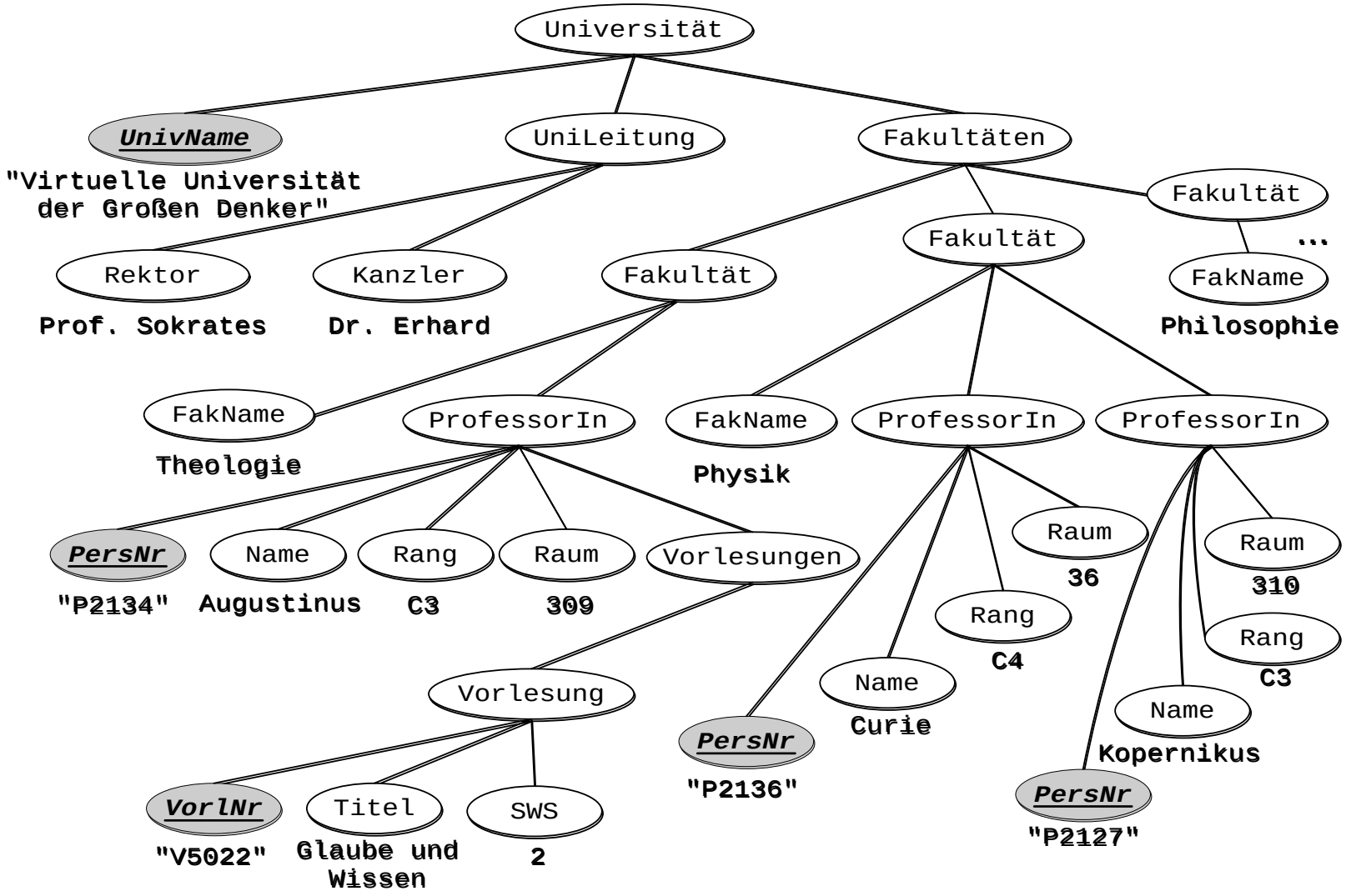
- Motivation
  - Bedarf für flexiblen, wählbar feingranularen Zugriff
  - Vergleichbar mit SQL?

- Anforderungen
  - Klare Semantik
    - Erforderlich für Anfragetransformation und –optimierung
    - Bevorzuge einfache Kernsprache vor benutzerfreundlicher Sprache mit unklarer Semantik
  - Zusammensetzbarkeit
    - Erforderlich für flexible Komposition von Anfragen:
      Output(query1) = Input(query2)
    - Anfrageergebnisse dürfen das Datenmodell nicht verlassen
  - Schema
    - Strukturbewusst
    - Ausnutzung Strukturkenntnis für Typüberprüfung & Optimierung

XPath

- **XPath** ist eine Adressierungs-Sprache zur Lokalisierung von Knoten im XML-Baum
- Grundlage von XPath ist das XML Information Set, d.h. XML-Dokument werden als Baum verschiedener Knotentypen interpretiert
- Das zentrale Konzept von XPath sind **Lokalisierungspfade**, die aus aneinander gereihten **Lokalisierungsschritten** bestehen.
  - Lokalisierungsschritte sind mit dem „/“-Zeichen voneinander getrennt
  - Ein Lokalisierungsschritt selektiert, ausgehend von einem Referenzknoten, eine Knotenmenge
  - Jeder Knoten dieser Menge dient als Referenzknoten für nachfolgende Lokalisierungsschritte in einem längeren Lokalisierungspfad
  - Die selektierten Knotenmengen des letzten Lokalisierungsschritt werden vereinigt und bilden das Ergebnis des gesamten Lokalisierungspfades
  - (Ergebnisknoten sind in Dokumentreihenfolge auszugeben)

Beispiel einer Universität



Lokalisierungspfad

- Sequenz von durch "/" getrennten Lokalisierungsschritten
  - Beispiel: /Universität/UniLeitung/Kanzler
    berechnet alle Kanzler-Elemente alle Universitätsleitungen aller Universitäten

- Sukzessive Auswertung der Lokalisierungsschritte
  - Ausgangspunkt: Referenzknoten (Kontextknoten)
  - Selektiere Knotenmenge auf Basis dieses Knotens
  - Für jeden Knoten aus dieser Menge:
    - werte folgenden Lokalisierungsschritt mit diesem Knoten als Referenzknoten aus
  - Vereinige die berechneten Knotenmengen der folgenden Lokalisierungsschritte
  - Beispiel: /Universität/Fakultäten/ProfessorIn
    berechnet alle ProfessorIn-Elemente aller Fakultäten aller Universitäten

Lokalisierungsschritt

- Ein Lokalisierungsschritt hat die folgende Struktur:

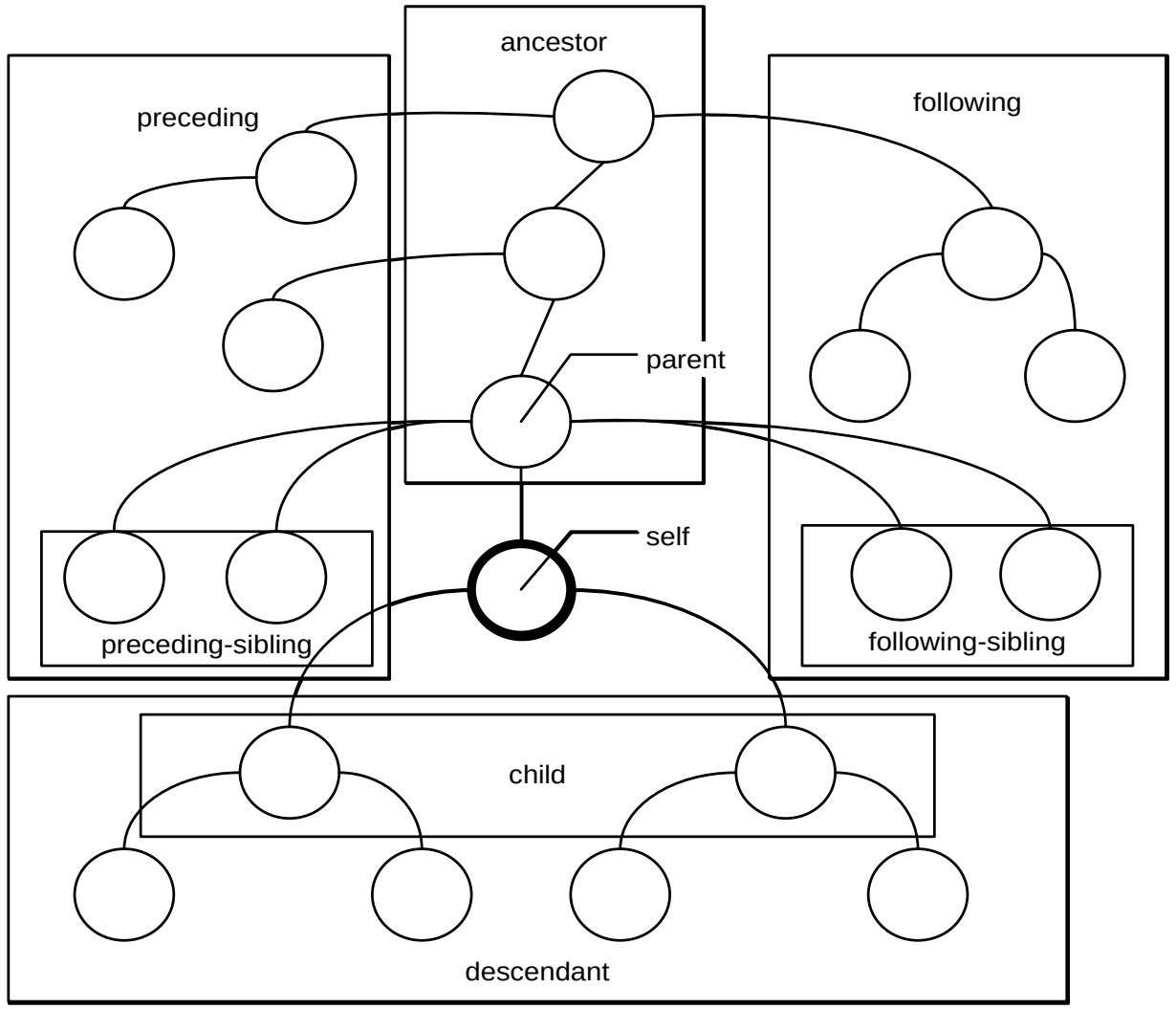
Achse::Knotentest[Prädikat]

- Die **Achse** selektiert eine Menge von Knoten abhängig von der Relation zum Kontextknoten im XML-Baum ("Traversierungs-Richtung"). Sie kann beliebig viele Knoten als Ergebnis haben.
- Der **Knotentest** schränkt eine Achse ein. Es handelt sich hierbei in der Regel um einen einfachen Namenstest.
- Das **Prädikat** überprüft komplexere Bedingungen zur weiteren Einschränkung der Knotenmenge.

XPath: Achsen

- self: Referenzknoten selbst
- attribute: Attributknoten des Referenzknotens
- child: Alle direkten Unterelemente
- descendant: Alle direkten und indirekten Unterelemente
- descendant-or-self: descendant vereinigt self
- parent: Vaterknoten des Referenzknotens
- ancestor: Alle Knoten auf dem Pfad vom Referenzknoten bis zur Wurzel
- ancestor-or-self: ancestor vereinigt self
- following-sibling: In Dokumentreihenfolge nachfolgende Kinder des Elternknotens vom Referenzknoten
- preceding-sibling: In Dokumentreihenfolge vorangehende Kinder des Elternknotens vom Referenzknoten
- Following: Alle Elementknoten nach Referenzknoten in Dokumentreihenfolge, ohne Kindknoten des Referenzknotens
- Preceding: Alle Elementknoten vor Referenzknoten in Dokumentreihenfolge, ohne Kindknoten des Referenzknotens

Visualisierung der Achsen



XPath: Knotentest

- node(): Ist erfüllt für Knoten beliebigen Typs (immer wahr)
- *: Ist erfüllt für Knoten des Typs der Achse
  - Für attribute-Achse: Attributknoten,
  - Ansonsten Elementknoten
- element(<ElementName>): Ist erfüllt für Elementknoten mit angegebenem Namen
- attribute(<AttributName>): Ist erfüllt für Attributknoten mit angegebenem Namen
- ...

XPath: Prädikate

- Knotentest erlaubt nur rudimentäre Filterung von Knoten der Achse
- Prädikate ermöglichen benutzerdefinierte Einschränkungen, sie sind selbst XPath Ausdrücke
- Beispiel: `//ProfessorIn[descendant::Vorlesungen]`
  alle ProfessorInnen mit einem Vorlesungs-Unterelement
- Prädikate müssen nicht am Ende eines Pfadausdruckes stehen, können geschachtelt sein und können boolsche Ausdrücke beinhalten.

- Prädikatauswertung
  - Ausgangspunkt: Knotenmenge aus Achse nach Überprüfung des Knotentests
  - Für jeden Knoten aus dieser Menge:
    - überprüfe Prädikat mit Knoten als Kontextknoten und schließe Knoten ggf. aus
    - wenn Pfadausdruck leere Knotenmenge adressiert, schließe Knoten aus

XPath: Beispiele

- Fakultätsnamen der Universität
  /child::Universität/child::Fakultäten/child::Fakultät/
  child::FakName
  /Universität/Fakultäten/Fakultät/FakName

- Personal-Nummern aller ProfessorInnen
  //ProfessorIn/attribute::attribute(PersNr)

- Ränge der ProfessorInnen, die einen Assistenten haben
  //Assistent/ancestor::ProfessorIn/Rang

Verkürzte XPath-Syntax

- Die wichtigsten Abkürzungen in XPath-Pfadausdrücken sind die folgenden:
- . Der aktuelle Referenzknoten
- .. Vaterknoten des Referenzknotens
- / Bezeichnet den Wurzelknoten und dient als Trennzeichen zwischen den einzelnen Schritten des Pfades
- @ Bezeichnet die Attribute des aktuellen Knotens
- // Alle Nachfahren des aktuellen Knotens einschließlich des Referenzknotens
- [n] Wählt das n-te Element aus

Beispiele:

- Fakultätsnamen der Universität
  /Universität/Fakultäten/Fakultät/FakName
- Personal-Nummern aller ProfessorInnen
  //ProfessorIn/@PersNr
- Alle Vorlesungen der Physik-Fakultät:
  /Universität/Fakultäten/Fakultät[FakName="Physik"]//Vorlesung

XML Query (XQuery)

- Deklarative Anfragesprache für XML-Dokumente
- Vergleichbar zu SQL
- Eine Anfrage in XQuery ist ein Ausdruck, der
  - eine Anzahl von XML-Dokumenten oder -Fragmenten liest
  - eine Sequenz wohlgeformter XML-Fragmente zurückgibt
- Entwurfsziele
  - Ausdrucksstärke
  - Anwendbarkeit in unterschiedlichen Umgebungen
  - Minimalismus und klares Design

- Überblick: http://www.w3.org/XML/Query/
- Spezifikation: http://www.w3.org/TR/xquery/
- Fülle von Beispielen: http://www.w3.org/TR/xquery-use-cases/

XQuery Beispiel

- Verzeichnis aller Vorlesungen im Hochschul-Dokument

```
<Vorlesungsverzeichnis>
  {for $v in doc("Uni.xml")//Vorlesung
   return $v
  }
</Vorlesungsverzeichnis>
```

XQuery Anfrageresultat

```
<?xml version="1.0" encoding="UTF-8"?>
<Vorlesungsverzeichnis>
  <Vorlesung VorlNr="V5022">
    <Titel>Glaube und Wissen</Titel>
    <SWS>2</SWS>
  </Vorlesung>
  <Vorlesung VorlNr="V5041" Voraussetzungen="V5001">
    <Titel>Ethik</Titel>
    <SWS>4</SWS>
  </Vorlesung>
  <Vorlesung VorlNr="V5049" Voraussetzungen="V5001">
    <Titel>Mäeutik</Titel>
    <SWS>2</SWS>
  </Vorlesung>
  ...
</Vorlesungsverzeichnis>
```

XSL / XSLT

- W3C Recommendation:
  http://www.w3.org/TR/xslt (Version 3.0 seit 06/2017)
- starke Marktdurchdringung (z.B. Bestandteil in allen gängigen Web-Browsern)
- Traditionell wichtigste Anwendungsdomäne:
  - Transformation von XML-Elementen in andere XML- oder Textdokumente (z.B. CSV, TeX)

Matching Expression

```
<xsl:template match="match-expr">
        Inhalt
</xsl:template>
```

- *match-expr* ist ein XPath-Ausdruck. Die Regel ist anwendbar für alle Knoten der Ergebnismenge.
- *Inhalt* ist eine Menge von XSLT-Elementen, die das "Programm" zur Transformation der selektierten Elemente beschreiben

- Einfaches Beispiel:

```
<xsl:template match="/Universität/Vorlesung">
  <xsl:copy-of select="current()"/>
</xsl:template>
```

Anwendbar auf alle Vorlesungs-Unterelemente von Universität

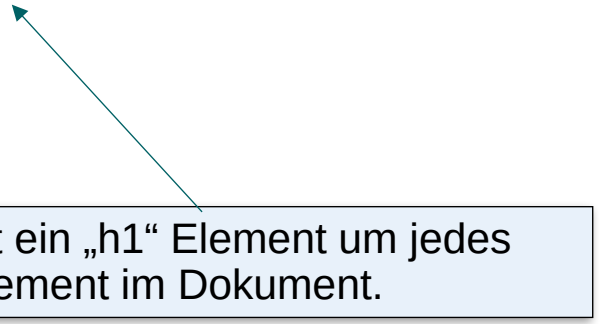
Kopiert alle Elemente in den Ausgabebaum

Ausführung XSLT-Styleheet

- Ausführung eines XSLT-Stylesheets beginnt mit dem Template für die Wurzel des Eingabebaums
- Ausführung weiterer Templates wird durch <xsl:apply-templates> gesteuert
- Neue Elemente und Attribute können in den Ausgabebaum eingefügt werden

- Beispiel:

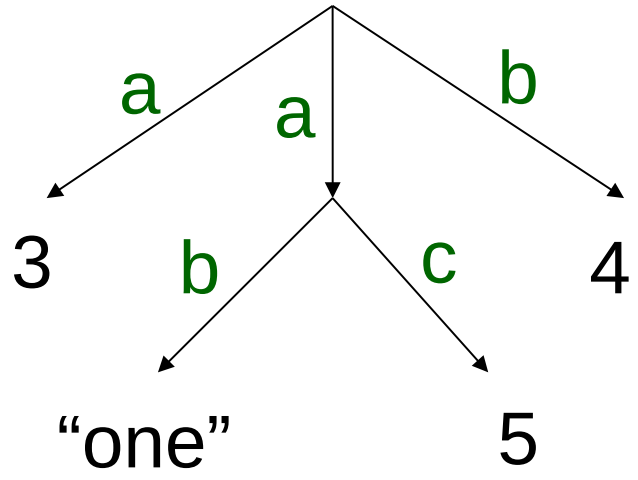
```
<xsl:template match="title">
  <h1><xsl:apply-templates/></h1>
</xsl:template>
```



Strukturelle Rekursion (1)

- Semistrukturierte Daten als geschachtelte Mengen mit Vereinigungs-Operator:

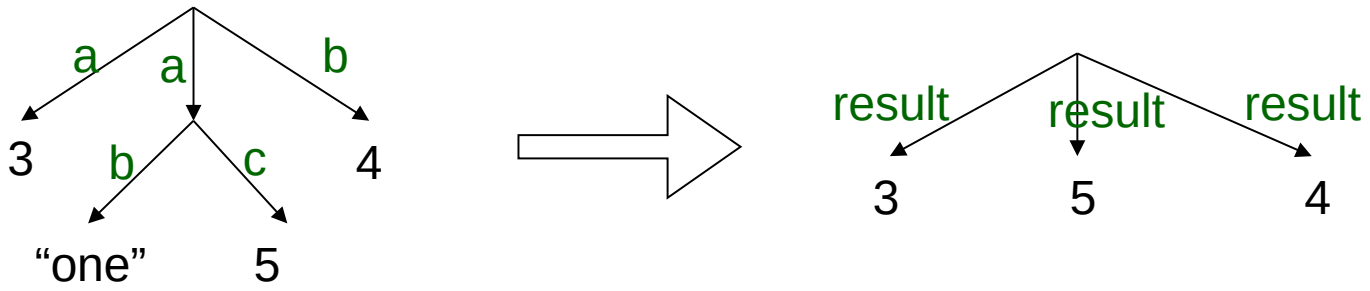
  {a:3, a:{b:”one”, c:5}, b:4} = {a:3} U {a:{b:”one”,c:5}} U {b:4}



Strukturelle Rekursion (2)

Beispiel: Liste alle Integer-Zahlen in den Daten auf

```
f(T1 U T2) =  f(T1) U f(T2)
f({L: T})     =  f(T)
f({})         =  {}
f(V)          =  if isInt(V) then {result: V} else {}
```

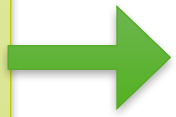


XSLT Beispiel (1)

```
<xsl:template> <xsl:apply-templates/> </xsl:template>

<xsl:template match="a">  <A><xsl:apply-templates/></A>
</xsl:template>

<xsl:template match="b">  <B><xsl:apply-templates/></B>
</xsl:template>

<xsl:template match="c">  <C><xsl:value-of/></C>
</xsl:template>
```

XSLT Beispiel (2)

```
<a> <e> <b> <c> 1 </c>
            <c> 2 </c>
        </b>
        <a> <c> 3 </c>
        </a>
    </e>
    <c> 4 </c>
</a>
```



```
<A> <B> <C> 1 </C>
          <C> 2 </C>
    </B>
    <A> <C> 3  </C>
    </A>
    <C> 4 </C>
</A>
```

XSLT ist Strukturelle Rekursion

Äquivalent zu

```
f(T1 U T2) = f(T1) U f(T2)
f({L: T})    =  if L= c then {C: T}
                else L= b then {B: f(T)}
                else L= a then {A: f(T)}
                else f(T)
f({})        =  {}
f(V)         =  V
```

XSLT
- Nur Bäume
- Schleifen möglich

Strukturelle Rekursion
- Beliebige Graphen
- Stets terminierend

XPath vs. XQuery vs. XSL/XSLT

XPath

- **Adressierungssprache**
- Genutzt von XQuery, XSLT, DOM (Document Object Model) etc
- Auswahl von XML-Knoten durch Lokalisierungspfade
- mächtige Selektionsmechanismen
- Ergebnis:
  - Folge von XML-Knoten
    (Sortierung durch Dokumentordnung vorgegeben)
  - keine Formatierung der Ausgabe möglich

XQuery (XML Query)

- **Anfragesprache**
- nutzt XPath zur Knotenselektion
- ermöglicht Kombination von XML-Knoten-Mengen
- erlaubt komplexe Operationen vergleichbar mit SQL

XSL / XSLT

- **Template- und Transformationssprache**
- erlaubt Transformation von XML Dokumenten in beliebige XML und Textformate

JSON - JavaScript Object Notation

- Alternative zu XML
- Basiert auf einer Teilmenge der Programmiersprache JavaScript
  - Jedoch Programmiersprachen unabhängig
- Einfach zu lesen, zu bearbeiten und zu parsen

- Gemeinsamkeiten mit XML
  - Textbasiert, menschenlesbar
  - Selbst-beschreibend
  - Hierarchisch

- Unterschiede zu XML
  - typisierte Objekte (String, Number, Array, Boolean, Null, Object)
  - Keine Namespaces
  - Kein Schema, keine implizite Validierung

Vergleich XML und JSON am Beispiel

```
<?xml version='1.0' encoding='UTF-8'?>
<UniDaten>
        <Professoren>
  <ProfessorIn>Curie</ProfessorIn>
  <ProfessorIn>Sokrates</ProfessorIn>
        </Professoren>
        <Vorlesungen><!-- WS 2015  -->
  <Vorlesung>Mäeutik</Vorlesung>
  <Vorlesung>Bioethik</Vorlesung>
        </Vorlesungen>
</UniDaten>
```

XML

- Schema-validierbar
- erweiterbar
- XPath / XQuery / XSLT

- Aufgeblasene Darstellung
- Komplexe Serialisierung / Deserialisierung

JSON

- Kompakte Darstellung
- Einfache Serialisierung / Deserialisierung

- Keine Kommentare / Metadaten

```
{
 "Professoren": [
  "Curie",
  "Sokrates“ ],
 "Vorlesungen": [
  "Mäeutik",
  "Bioethik“ ]
}
```

7. Alternative Datenmodelle

Informatik 5 -
Informationssysteme
und Datenbanken
RWTH AACHEN UNIVERSITY

7.2.1 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.
- …

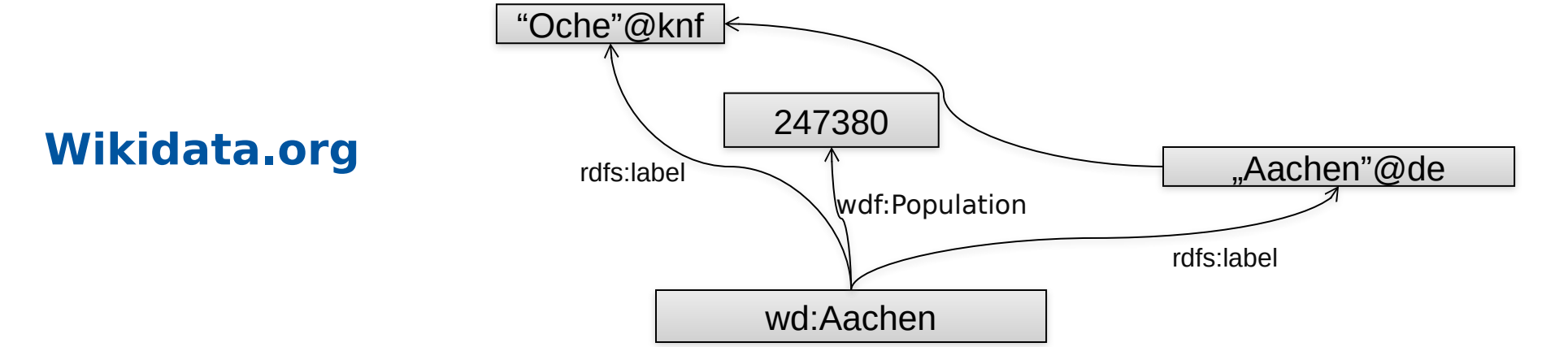
Informationsdarstellung in Tripeln

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

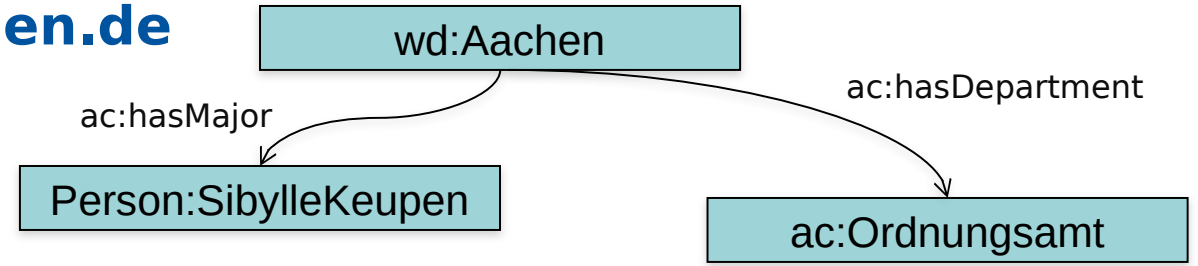
- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.

Wieso Graphen?

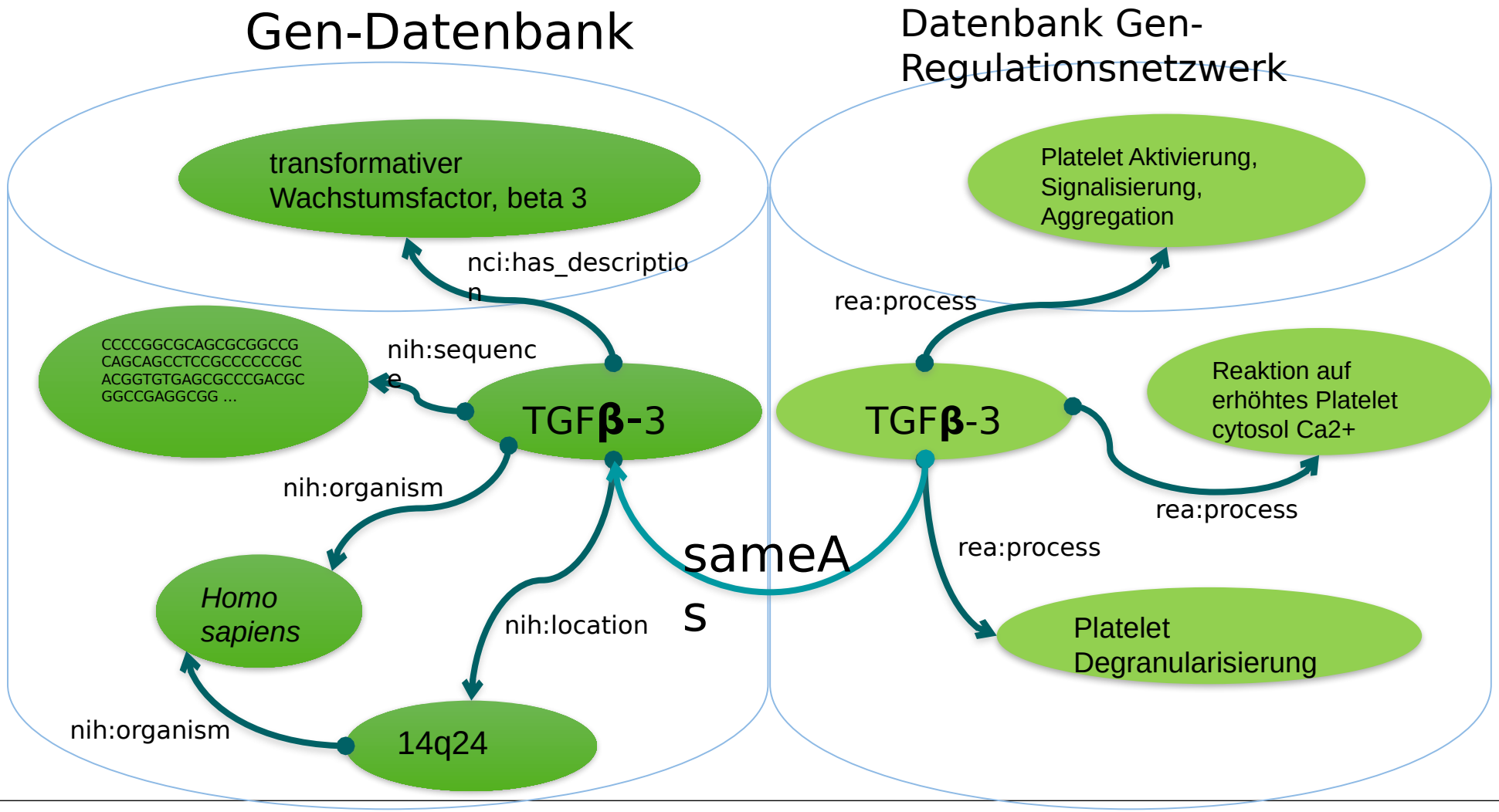
Wikidata.org



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

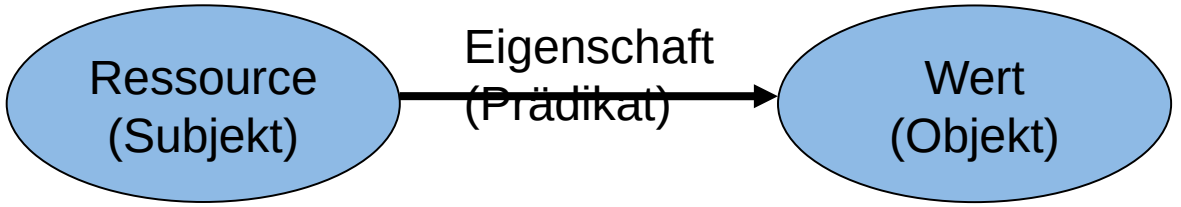
- Ermöglicht das gemeinsame Verständnis für eine Domäne
- Erlauben das Organisieren von Wissen in einem maschinen-lesbaren Format
- Gibt Daten einen auswertbare Bedeutung

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

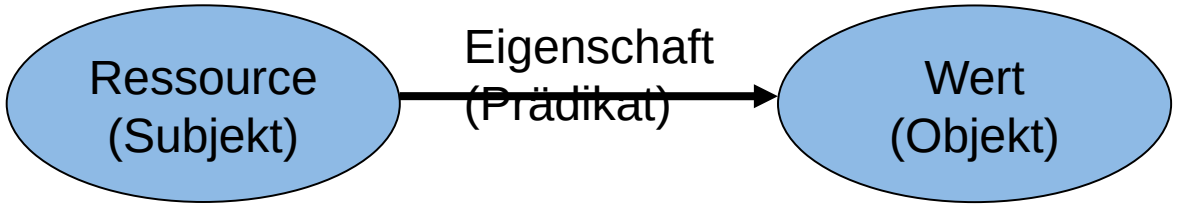
- RDF baut auf den Beziehungen zwischen Ressourcen auf
- Tripel der Form (s, p, o) sind der fundamentale Baustein von RDF
  - **Subjekt**
  - **Prädikat**
  - **Objekt**
- RDF verwendet Identifikatoren vom Web (URIs) um Ressourcen zu identifizieren



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*

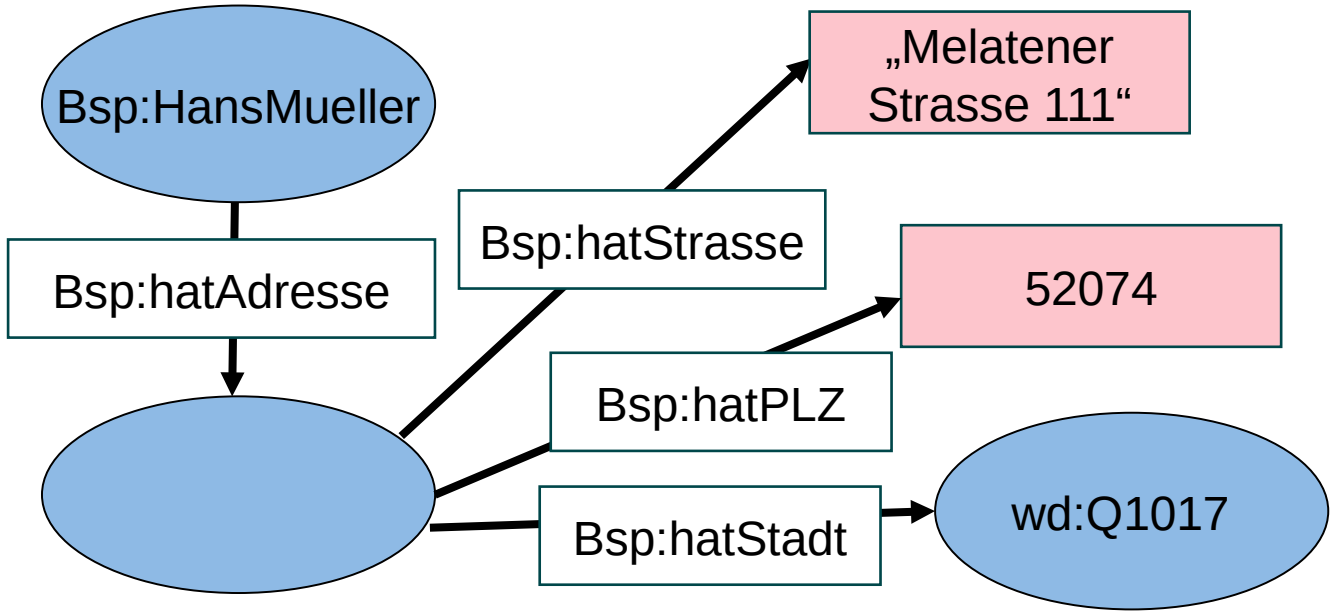
- Beispiel:
  - `"123"^^http://www.w3.org/2001/XMLSchema#int`

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

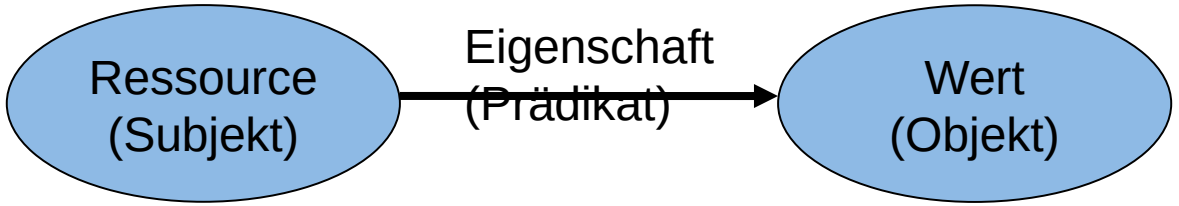
Beispiel für die Verwendung von Blank Nodes

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



Übersicht: Regeln für RDF Tripel

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



7.2.2 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:Q1893149 .
```

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

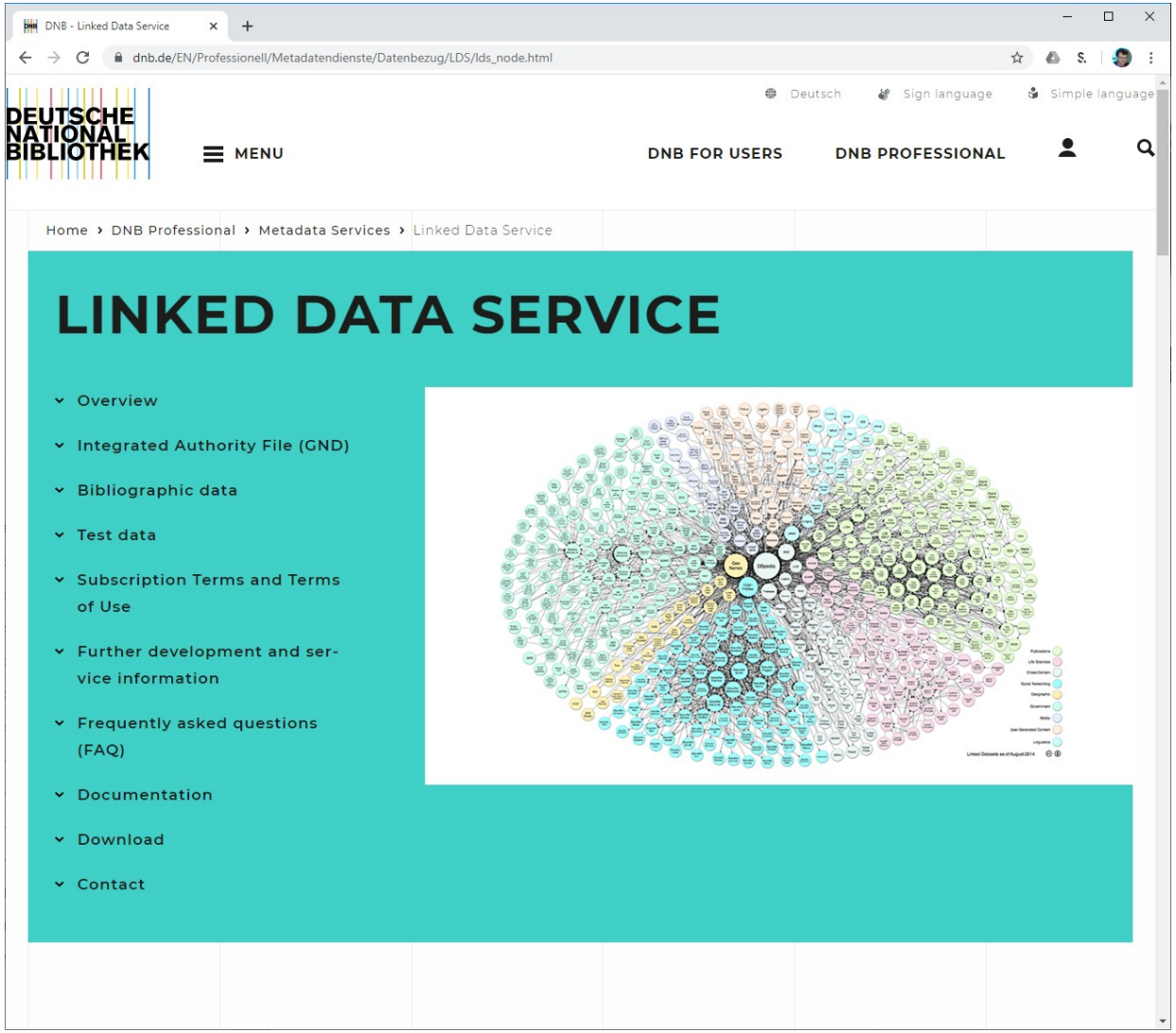
Siehe: https://www.w3.org/TR/turtle/

7.2.3 Beispiele zur Verwendung von RDF Daten

RDF in freier Wildbahn: Schema.org

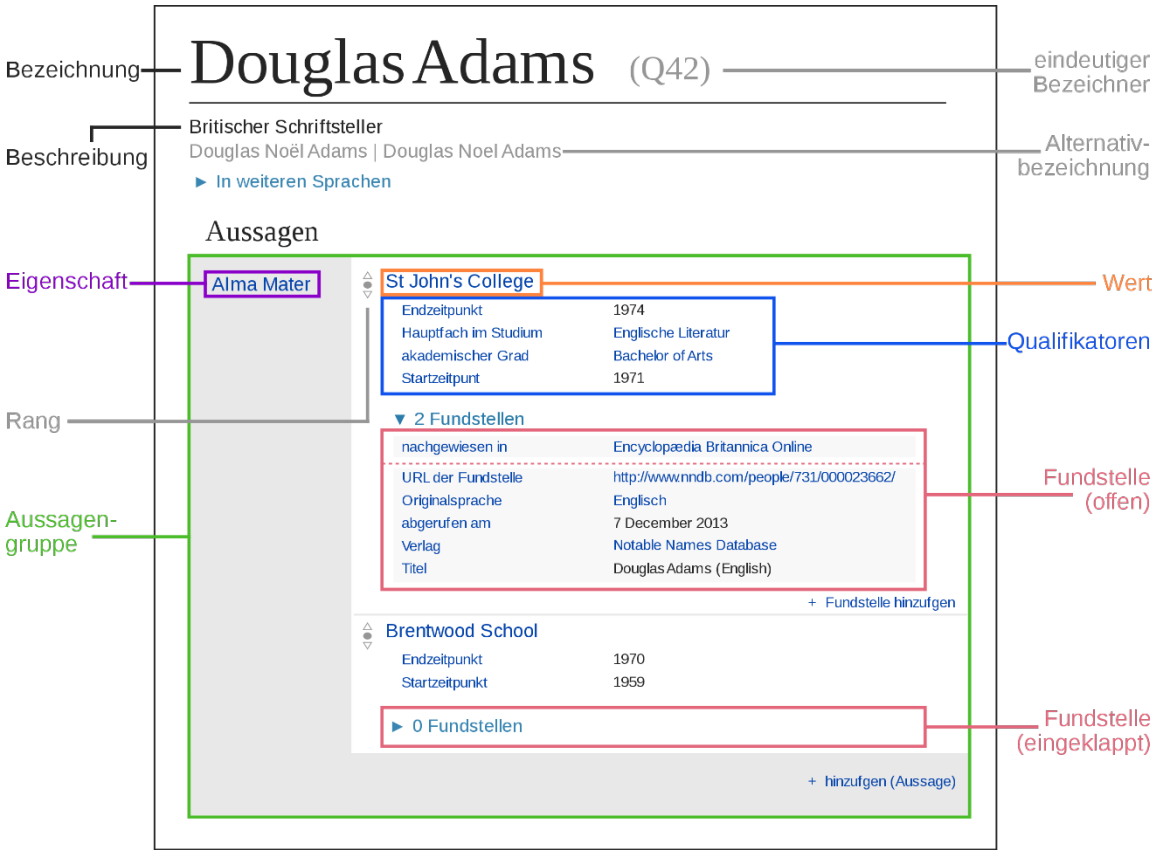
- 2010 von Google, Yahoo!, Microsoft & Yandex gestartet
- Ziele:
  - Vokabulare welches von allen Suchmaschinen unterstützt wird
  - Vereinfacht den Job des Webmasters und der Suchmaschinen-Optimierung
- Vokabular um Web-Seiten zu annotieren
- Kann Entitäten von den folgenden Typen annotieren:
  - „Creative“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

Screenshot von Linked Data Service DNB



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]



[1] https://www.wired.com/story/inside-the-alexa-friendly-world-of-wikidata/
[2] https://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext

7.2.4 SPARQL: Anfrage-Sprache für RDF Graphen

SPARQL: Anfrage-Sprache für RDF Graphen

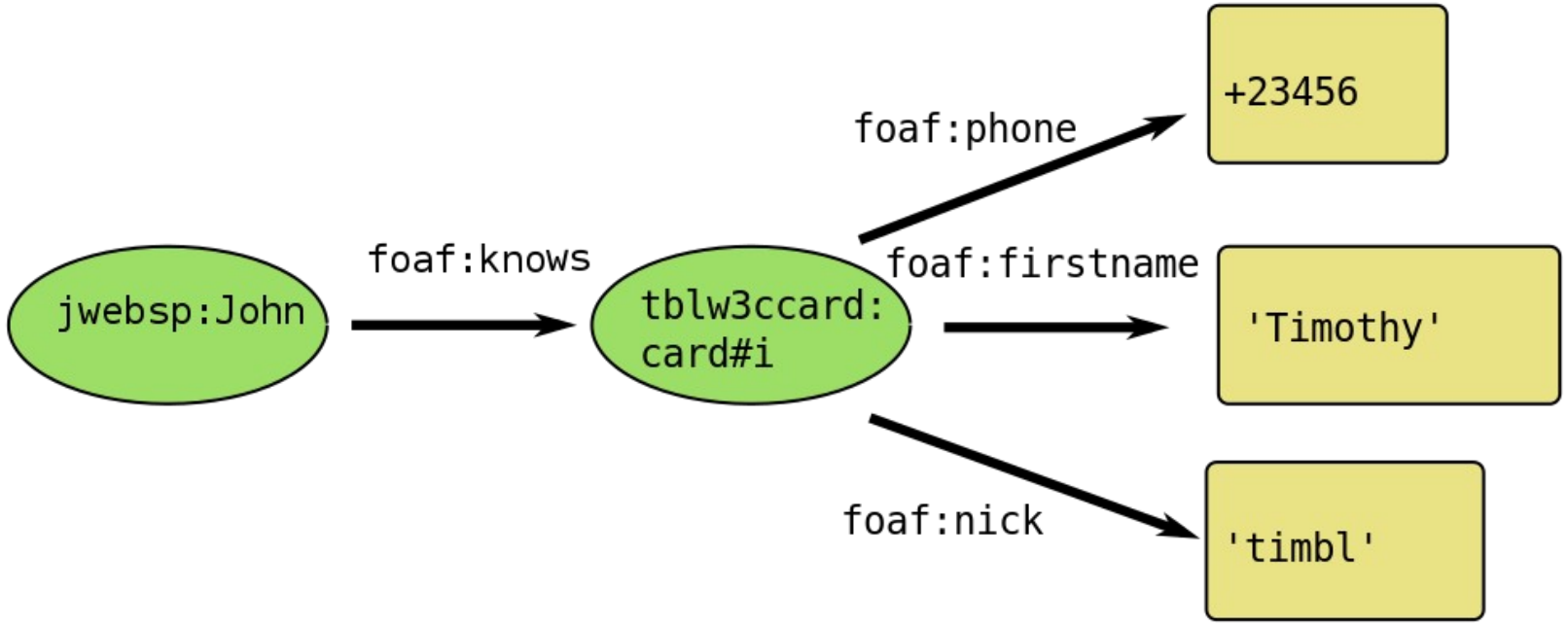
- SPARQL steht für “SPARQL Protocol and RDF Query Language” (gesprochen: “Sparkl”)

- https://www.w3.org/TR/sparql11-query/

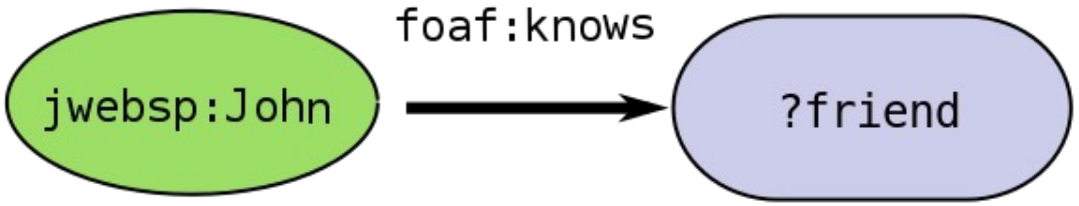
- **Bestandteile einer SPARQL Anfrage:**
  - Definition von Namespaces
  - Anfrage Klausel: Projektion
    - Vier Möglichkeiten: SELECT, ASK, CONSTRUCT, DESCRIBE
  - WHERE Klausel: Selektion durch ein Graph-Muster
  - Anfrage Modifikatoren

Ein kleiner Beispiel RDF Graph

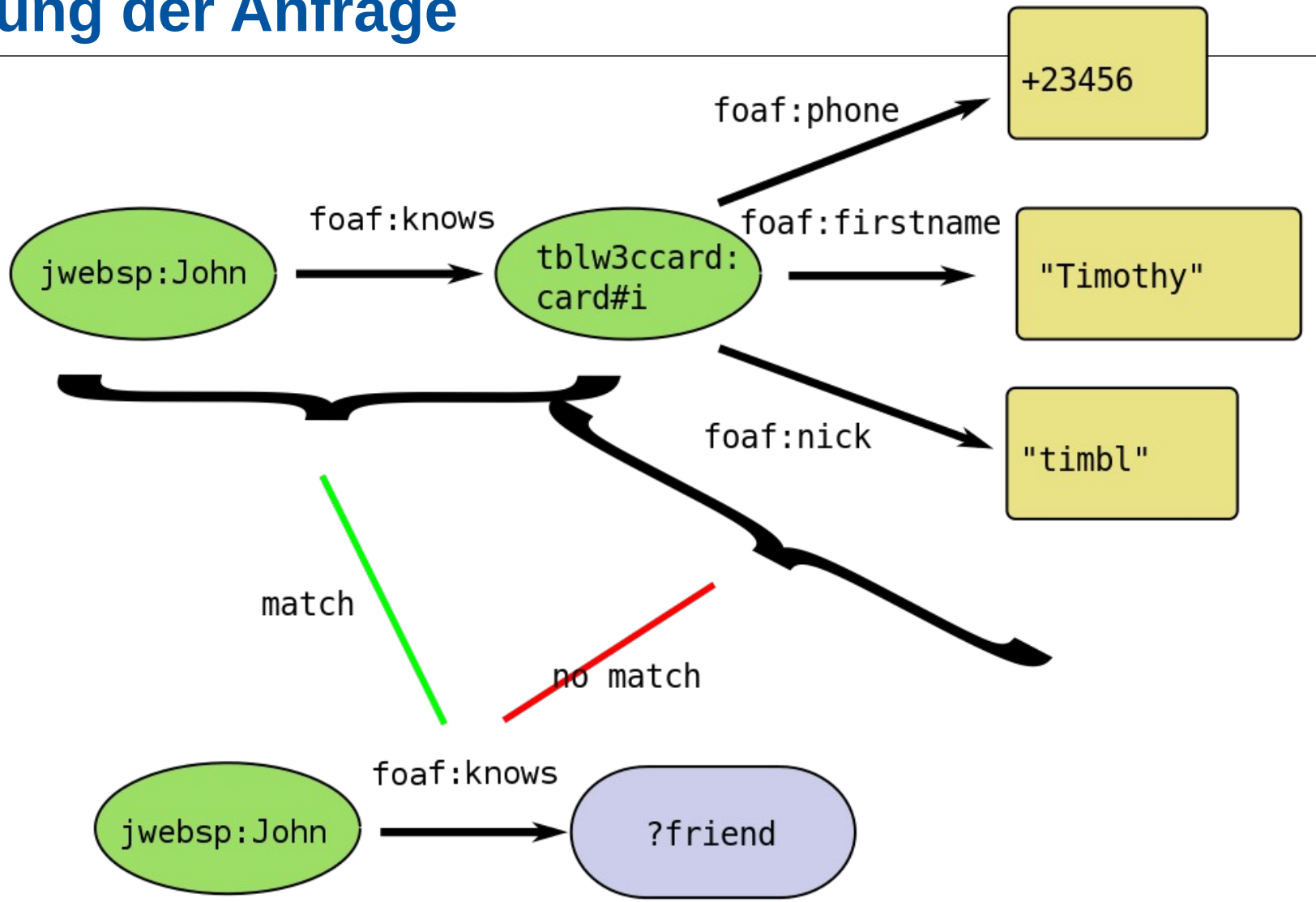
```
@prefix wbsp: <http://mywebspace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```



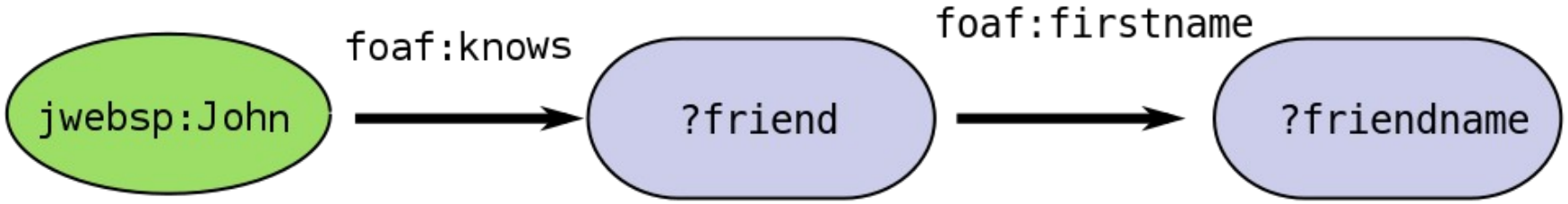
Eine sehr einfache Anfrage



Beantwortung der Anfrage



Eine etwas komplexere Anfrage

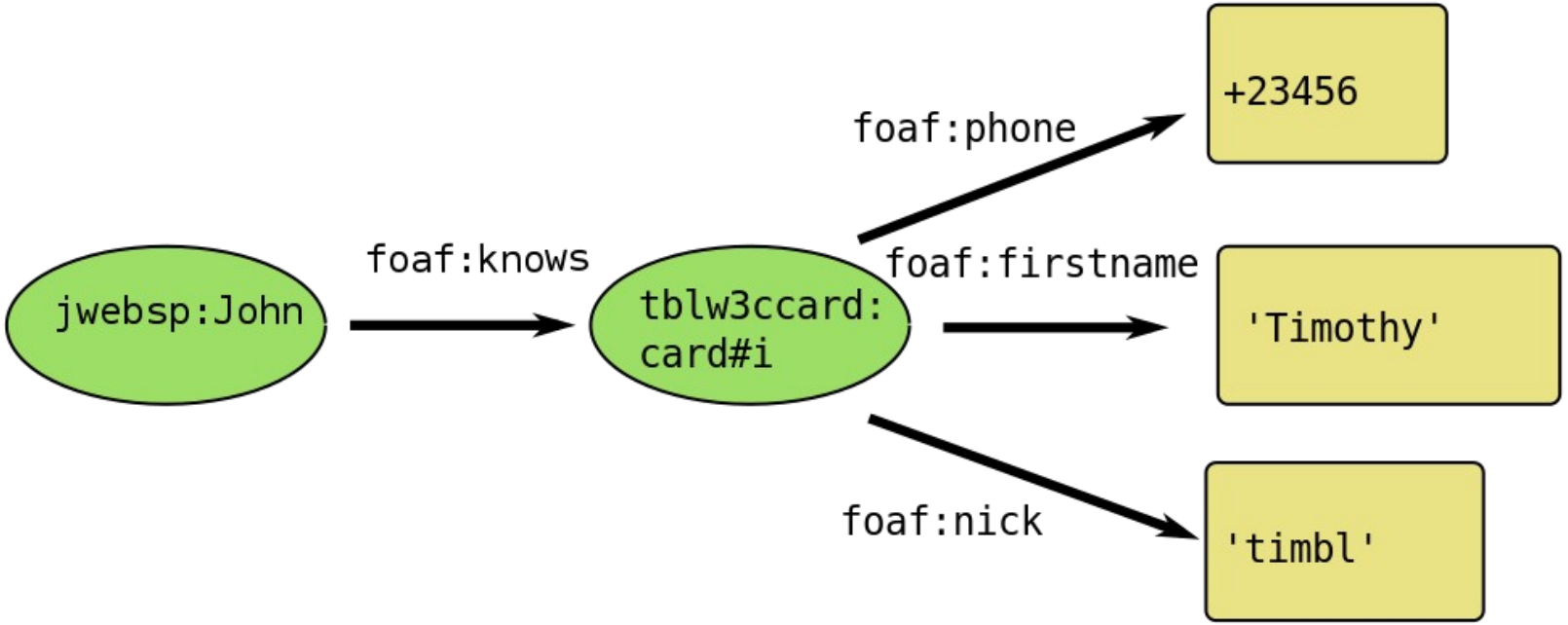


SPARQL für die Anfrage

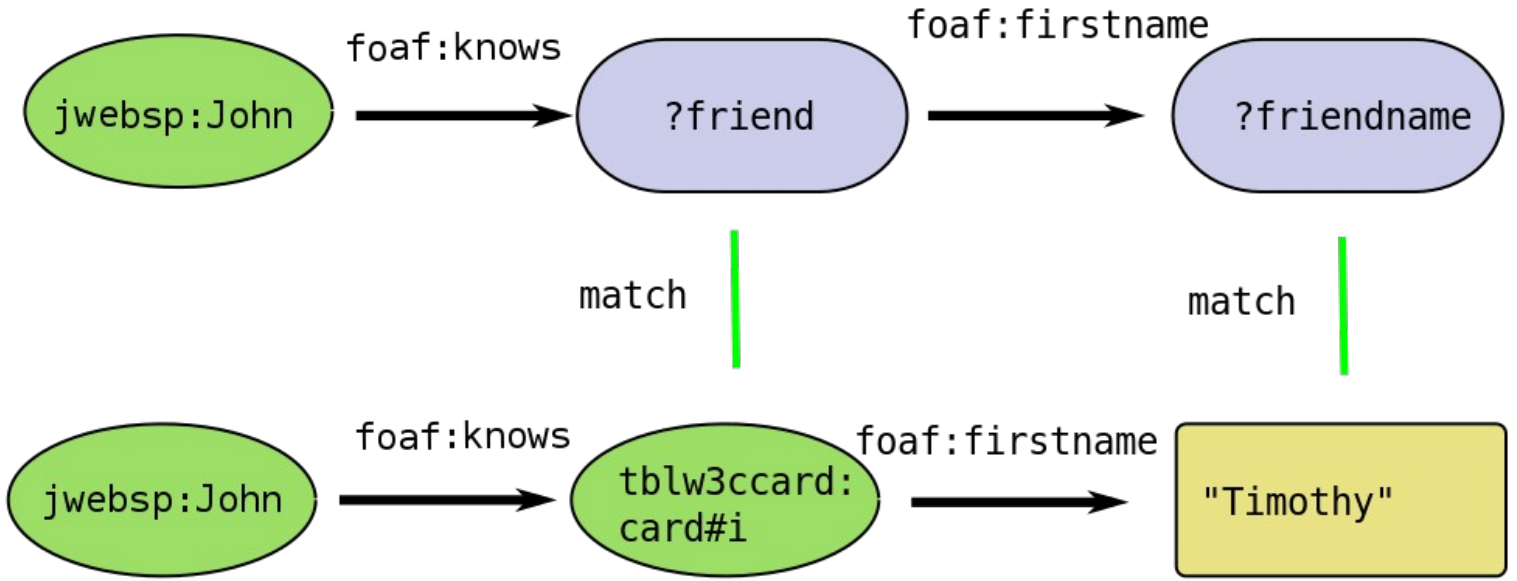
```
SELECT ?friend ?friendname WHERE {
 jwebsp:John foaf:knows ?friend .
 ?friend foaf:firstname ?friendname
}
```

Ein kleiner Beispiel RDF Graph

```
@prefix wbsp: <http://mywebspace.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"

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
```

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

```
-->    empty
```

```
SELECT ?friend {
       ?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
     {?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 {
                                ?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
```

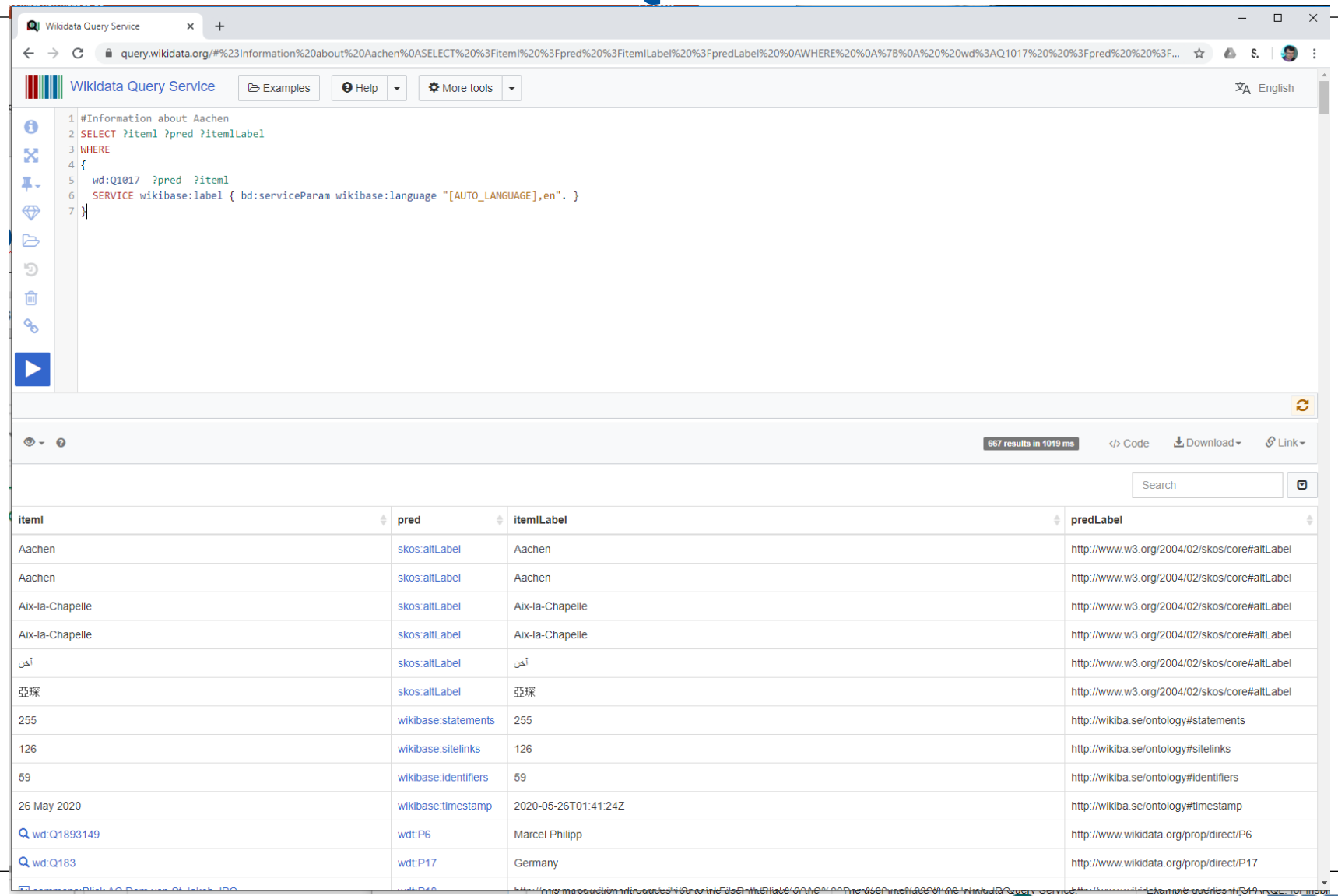
SPARQL Anfragen nach Klassen

- 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`

7.2.5 SPARQL in der Praxis

Screenshot von Wikidata SPARQL Service



SPARQL Anfragen stellen

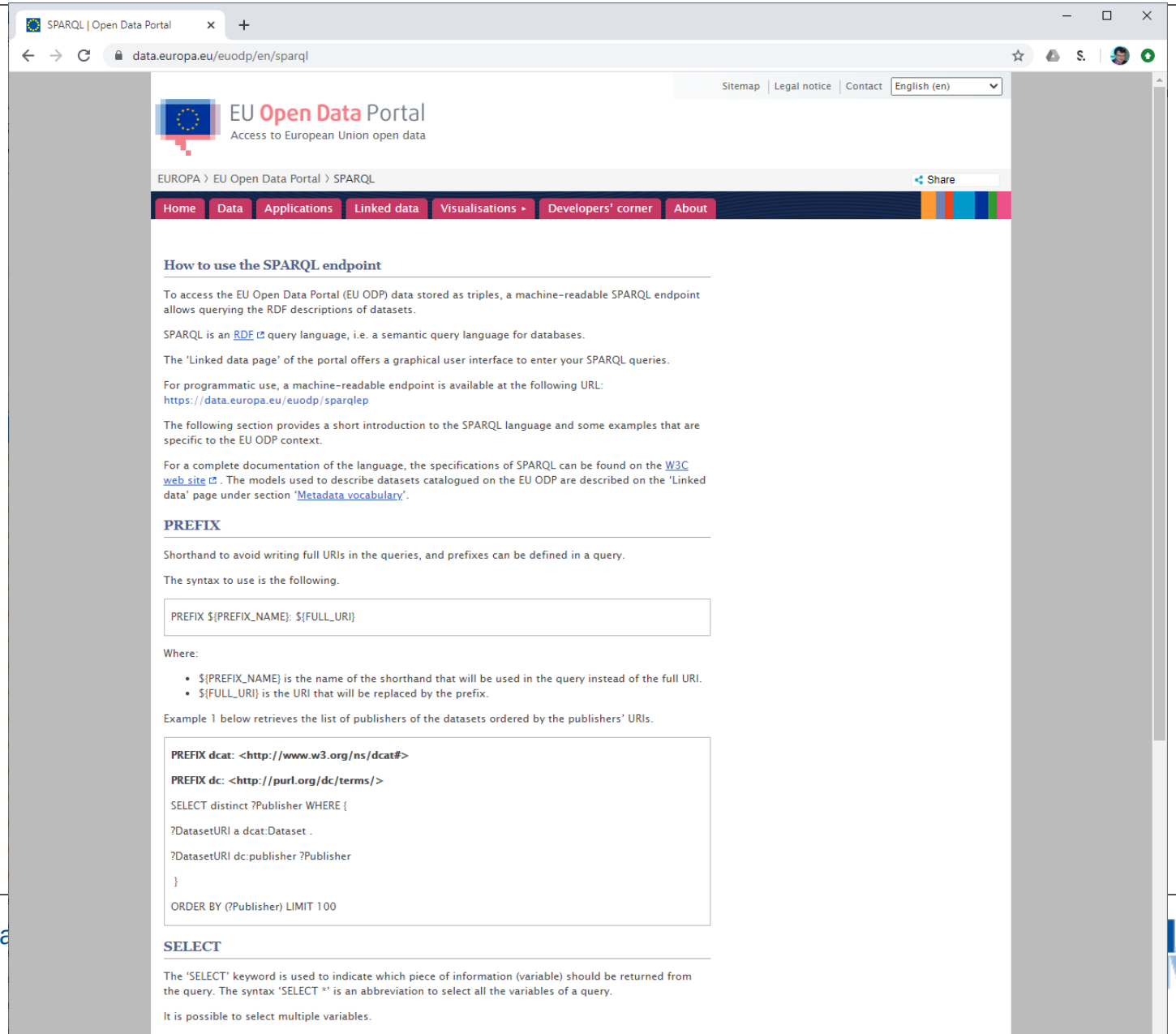
- Benutzte SPARQL um mehr über Entitäten herauszufinden mit „DESCRIBE“

```
#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



7.2.6 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 maschinen-lesbaren 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:MyBlueVWGolf .
```

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:

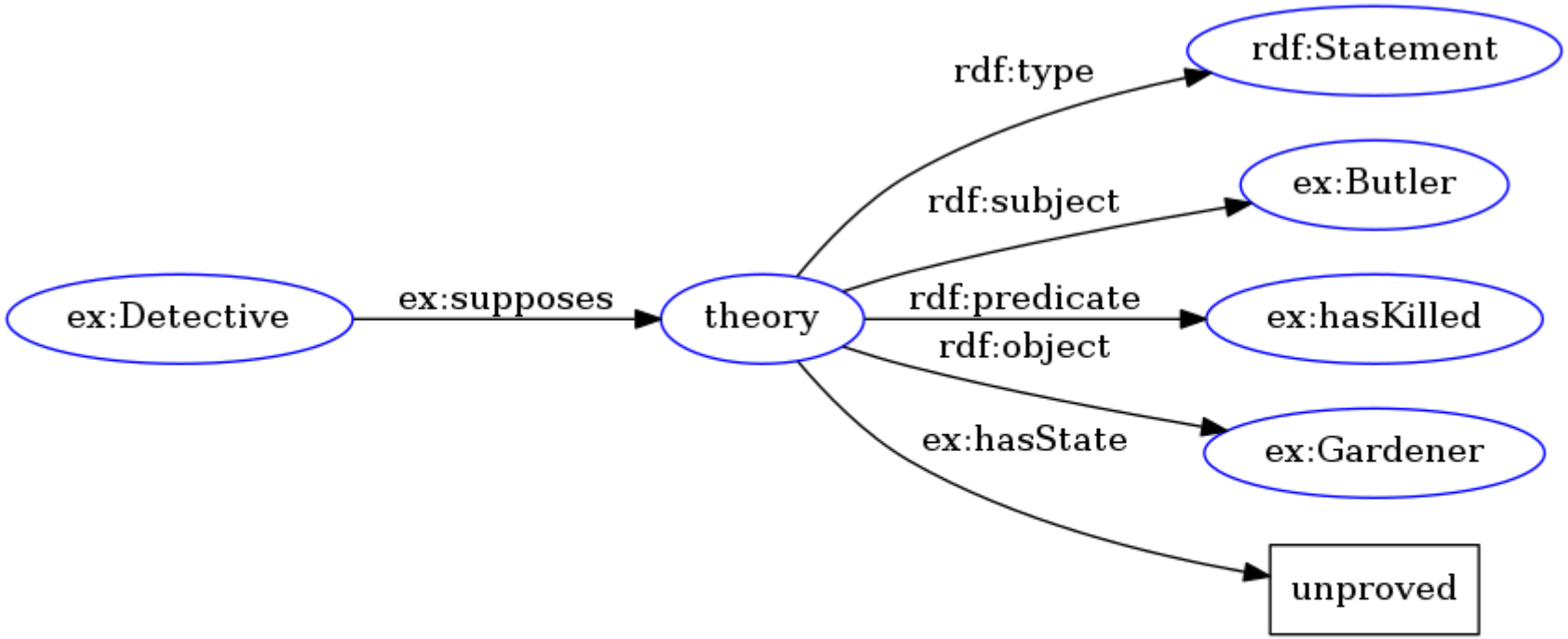
```
ex:Butler   ex:killed   ex:Gardener .
```

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 eine Namen (“human readable name”)
- **rdfs:comment** gibt einer Ressource eine Erklärung oder 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, das die zusätzliche Information auch als RDF ausgedrückt wird.

7.2.8 Modellierungsbeispiel aus der Chemie

Definitionen: Säure und Base

- Wir wollen ein System implementieren, welches in der Lage ist die notwendige Menge von Säure oder Base zu berechnen, um eine gegebene, chemische Lösung zu neutralisieren. Für diese Informationen definieren wir unser eigenes Schema.

- Wir fangen mit Basen und Säuren an:

```
ex:Acid rdf:type rdfs:Class .
ex:Base rdf:type rdfs:Class .
```

- Beide können als chemische Bindungen gesehen werden. Deswegen definieren wie dafür eine eigene Klasse mit zwei Unterklassen:

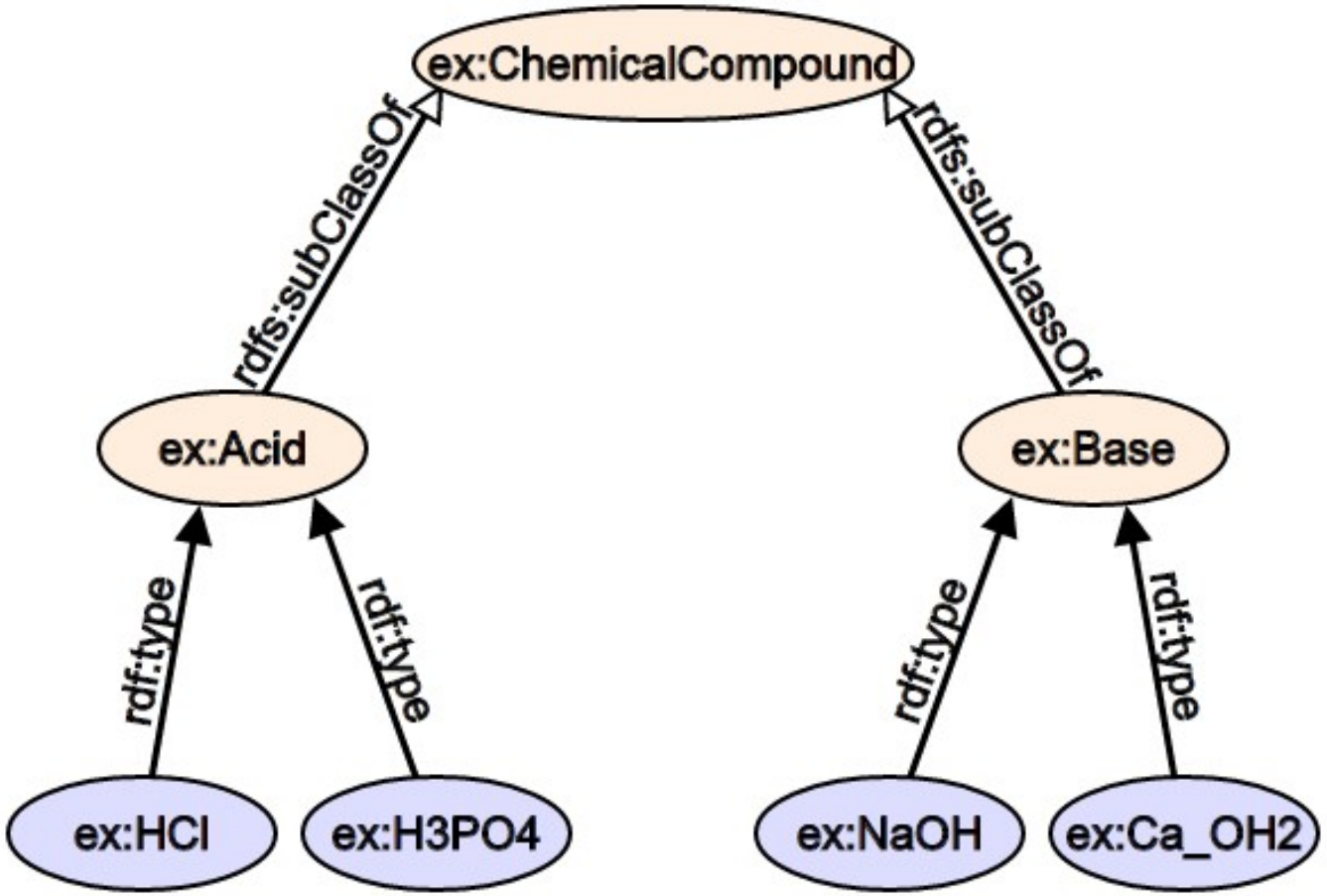
```
ex:ChemicalCompound   rdf:type            rdfs:Class .
ex:Acid               rdfs:subClassOf     ex:ChemicalCompound .
ex:Base               rdfs:subClassOf     ex:ChemicalCompound .
```

Definieren von Instanzen

- Als nächstes definieren wir ein paar Instanzen der Klassen für Basen und Säuren. Zum Beispiel:
  - Chlorwasserstoff (ex:HCl)
  - Phosphorsäure (ex:H3PO4)
  - Natriumhydroxid (ex:NaOH)
  - Calciumhydroxid (ex:Ca_OH2).

```
ex:HCl          rdf:type        ex:Acid .
ex:H3PO4        rdf:type        ex:Acid .
ex:NaOH         rdf:type        ex:Base .
ex:Ca_OH2       rdf:type        ex:Base .
```

Diagramm zum Beispiel



Definition der „Molaren Masse“

- Nachdem wir die Klassen definiert haben, wollen wir eine typische Eigenschaft aus der Chemie zu der Klasse hinzufügen. Zum Beispiel die Masse in Molar.
- Zuerst definieren wir ein Prädikat:

```
ex:hasMolarMass rdf:type rdfs:Property .
```

- Jede chemische Bindung kann eine molare Masse haben.

```
ex:hasMolarMass   rdfs:domain   ex:ChemicalCompound .
```

- Die molare Masse kann als eigener Datentyp definieren werden. Aber für unser kleines Beispiel reicht es eine Gleitkomma-Zahl zu verwenden.

```
ex:hasMolarMass   rdfs:range xsd:float .
```

- Ein Beispiel:

```
ex:NaOH   ex:hasMolarMass   "39.9971" .
```

Zusätzliche Prädikate

- Einer der wichtigsten Unterschiede zwischen Säuren und Basen, ist das Säuren Protonen abgeben können, während Basen Protonen aufnehmen können. Wir wollen zwei Prädikate definieren mit denen wir ausdrücken können wie viele Protonen aufgenommen oder abgegeben werden können

```
ex:canDonateProtons rdf:type     rdfs:Property .
ex:canDonateProtons rdfs:domain  ex:Acid .
ex:canDonateProtons rdfs:range   xsd:integer .

ex:canAcceptProtons rdf:type     rdfs:Property .
ex:canAcceptProtons rdfs:domain  ex:Base .
ex:canAcceptProtons rdfs:range   xsd:integer .
```

- Mit diesen neuen Prädikaten können wir zwischen Basen und Säuren anhand ihrer “Stärke” unterscheiden.

```
ex:HCl         ex:canDonateProtons  "1" .
ex:H3PO4       ex:canDonateProtons  "3" .
ex:NaOH        ex:canAcceptProtons  "1" .
ex:Ca_OH2      ex:canAcceptProtons  "2" .
```

Definition einer chemischen Lösung

- Unser System muss mit Lösungen umgehen können, die verschiedene Mengen von chemischen Bindungen enthalten können.
- Also müssen wir eine Klasse ex:Solution und ein Prädikat für die Masse einer Lösung definieren.

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

ex:hasMass    rdf:type    rdfs:Property .
ex:hasMass    rdfs:domain ex:Solution .
ex:hasMass    rdfs:range  xsd:float .
```

Definition: Zutat 1

- Unser System muss mit Lösungen umgehen können, die verschiedene Mengen von chemischen Bindungen enthalten können.
- Jedoch reicht es nicht wenn wir ausdrücken können das eine chemische Bindung Teil einer Lösung ist. Wir müssen auch ausdrücken welche Menge der Substanz in der Lösung ist.
- Deswegen definieren wir eine Klasse ex:Ingredient und ein Prädikat um die Menge in Prozent zu beschreiben.

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

ex:hasAmount  rdf:type    rdfs:Property .
ex:hasAmount  rdfs:domain ex:Ingredient .
ex:hasAmount  rdfs:range  xsd:float .
```

Definition: Zutat 2

- Zusätzlich braucht diese Klasse 2 Prädikate um mit einer ex:Solution und einem ex:ChemicalCompound verbunden zu werden.

```
ex:isPartOf rdf:type    rdfs:Property .
ex:isPartOf rdfs:domain ex:Ingredient .
ex:isPartOf rdfs:range  ex:Solution .

ex:contains rdf:type    rdfs:Property .
ex:contains rdfs:domain ex:Ingredient .
ex:contains rdfs:range  ex:ChemicalCompound .
```

Beispiel zur Verwendung unseres Schemas 1

- Mit unserem Beispiel Schema können wir jetzt alle Informationen bzgl. der folgenden, sehr typischen Aufgabenstellung aus der Chemie ausdrücken:
  - *Sie haben 100g einer Lösung aus 20% Phosphorsäure. Wieviel 10% Natriumhydroxid benötigen Sie um diese Lösung zu neutralisieren ?*
- Als erstes drücken wir die Informationen aus der Aufgabenstellung mit unserem Schema als **Input** für das Programm aus:

```
ex:GivenSolution     rdf:type       ex:Solution .
ex:GivenSolution     ex:hasMass     "100" .

ex:PhosAcidIng20Perc rdf:type       ex:Ingredient .
ex:PhosAcidIng20Perc ex:hasAmount   "20" .
ex:PhosAcidIng20Perc ex:contains    ex:H3PO4 .
ex:PhosAcidIng20Perc ex:isPartOf    ex:GivenSolution .

ex:SearchedSolution  rdf:type       ex:Solution .

ex:SodiumHyIng10Perc rdf:type       ex:Ingredient .
ex:SodiumHyIng10Perc ex:hasAmount   "10" .
ex:SodiumHyIng10Perc ex:contains    ex:NaOH .
ex:SodiumHyIng10Perc ex:isPartOf    ex:SearchedSolution .
```

Beispiel zur Verwendung unseres Schemas 2

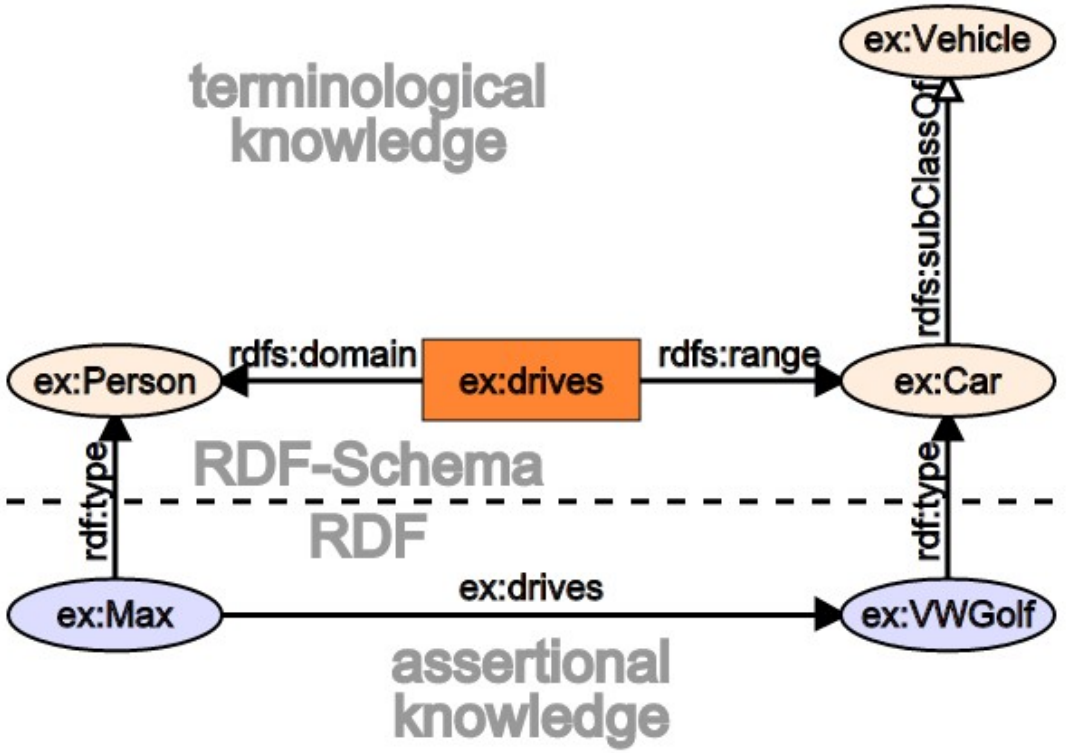
- Das System kann aufgrund des Input der vorherigen Folie, folgende Daten aus einer Chemie-Wissens-Datenbank abfragen:

```
ex:H3PO4 ex:hasMolarMass       "97.995" .
ex:H3PO4 ex:canDonateProtons   "3" .

ex:NaOH   ex:hasMolarMass      "39.9971" .
ex:NaOH   ex:canAcceptProtons  "1" .
```

- Zusammengenommen, kann das System folgende Lösung für die Aufgabenstellung berechnen (Chemie-Details wurden ausgelassen)
  - *Es werden 244.9g der Natriumhydroxidlösung benötigt um die gegebene Phosphorsäurelösung zu neutralisieren.*

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
- Anwendungs-Beispiel aus der Chemie

7. Alternative Datenmodelle

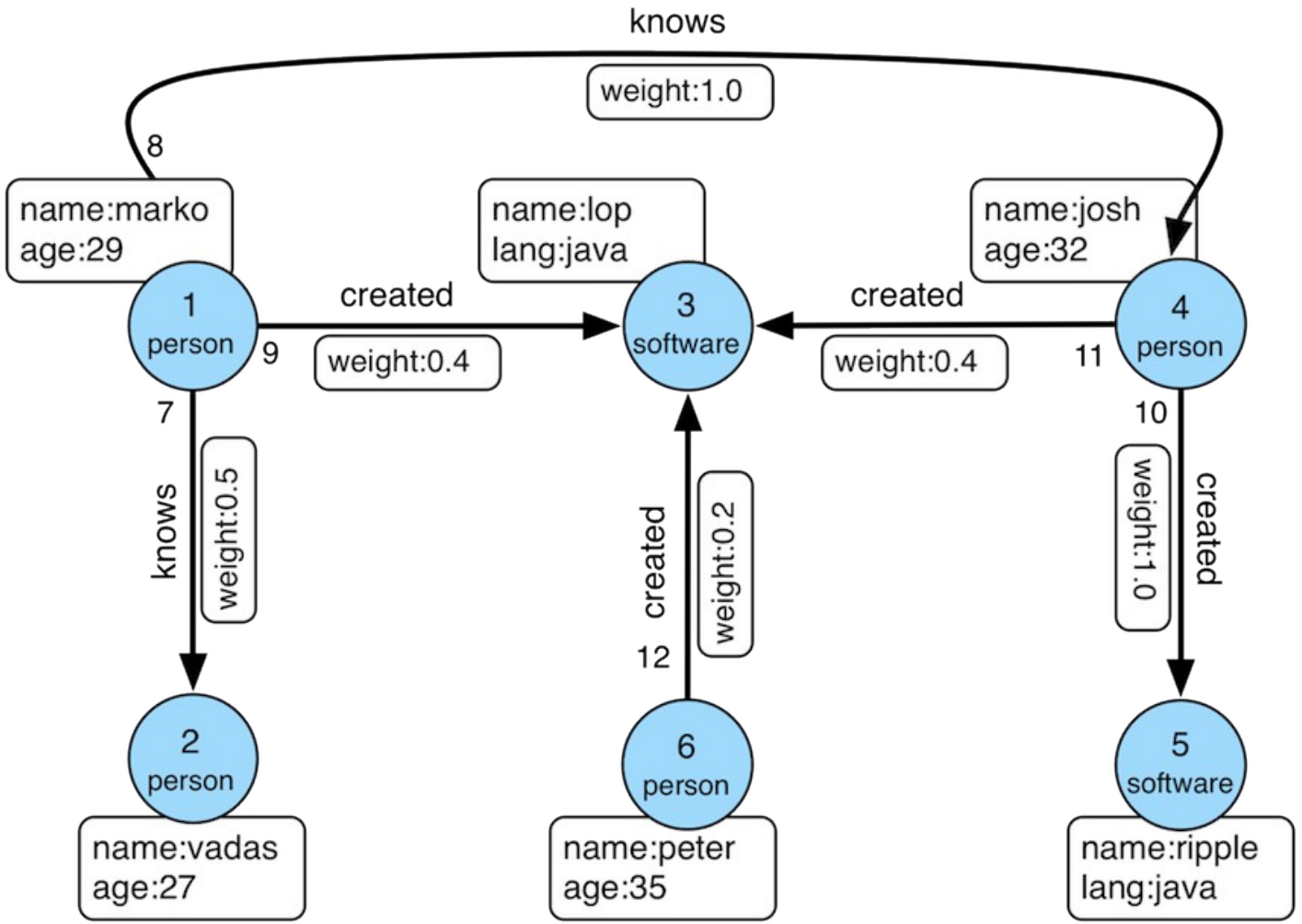
TODO

7.1 XML als semistrukturiertes Datenmodell

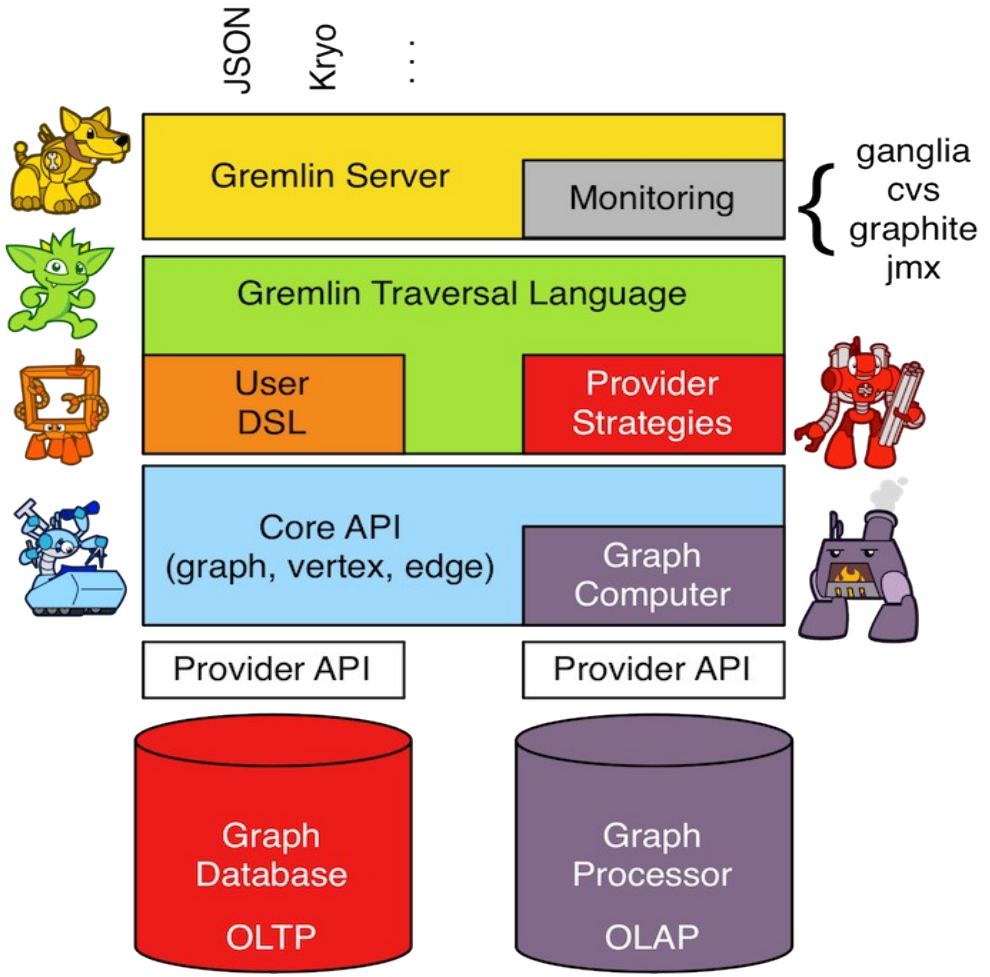
7.2 RDF und Semantic Web

7.3 Graph-Datenbanken

The apache tinkerpop „modern“ example graph



Graph Definition


JSON
Kryo
...
Gremlin Server
Monitoring
ganglia
cvs
graphite
jmx
Gremlin Traversal Language
User DSL
Provider Strategies
Core API (graph, vertex, edge)
Graph Computer
Provider API
Provider API
Graph Database
OLTP
Graph Processor
OLAP