

Vorlesung Semantic Web



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vorlesung im Wintersemester 2012/2013

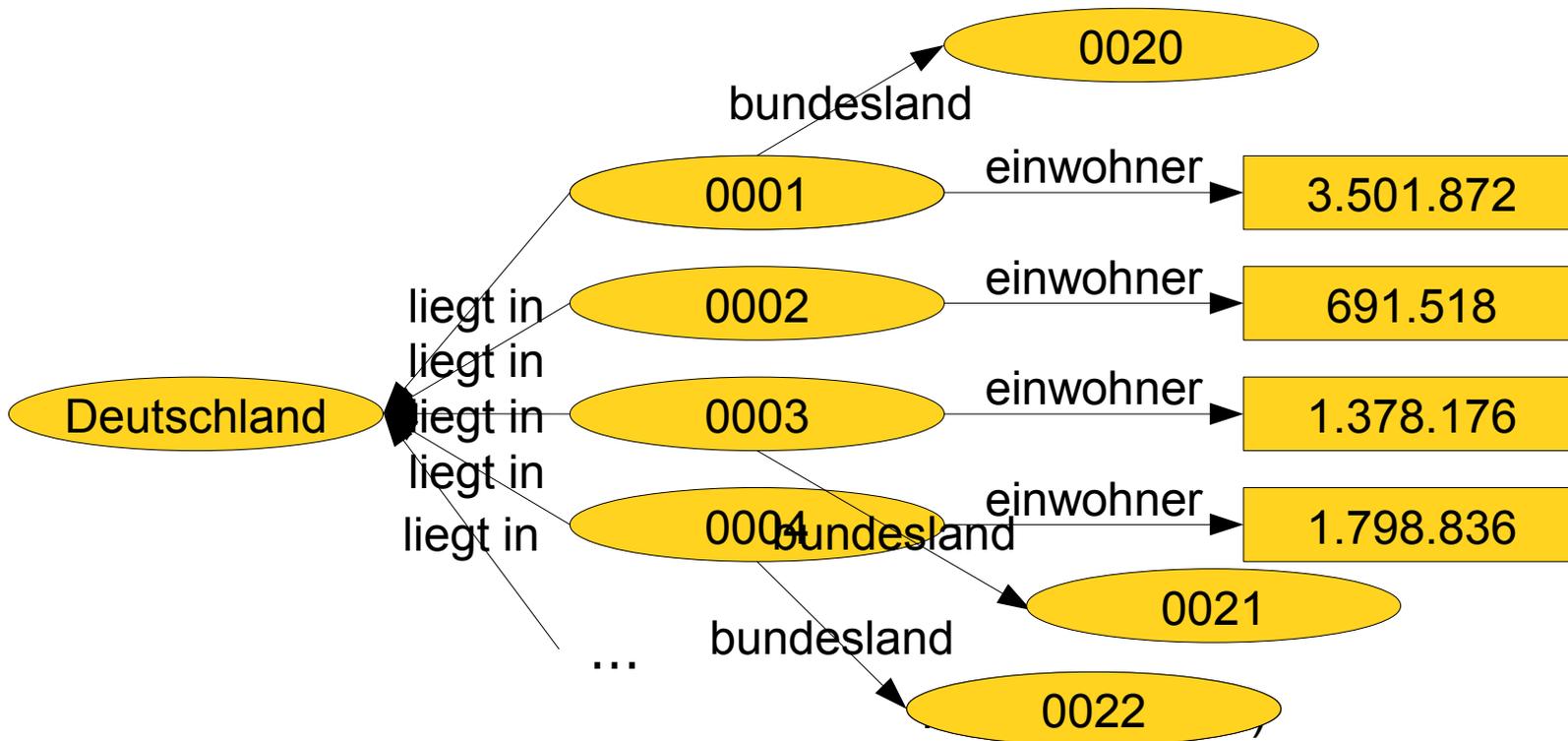
Dr. Heiko Paulheim

Fachgebiet Knowledge Engineering

Was bisher geschah

- Was wir bisher kennen gelernt haben:
 - RDF und RDF Schema als Sprachen
 - Linked Open Data
- Wie wir bisher auf Linked Open Data zugegriffen haben
 - mit Browsern
 - Graphen entlang hangelnd
- ...schauen wir uns das noch mal an!

Ein RDF-Beispielgraph



Aufgabe: in welchen Bundesländern liegen die fünf größten deutschen Städte?

Informationsbeschaffung auf Linked Open Data



- Aufgabe: in welchen Bundesländern liegen die fünf größten deutschen Städte?
- Probieren wir das aus...

Fazit

- So ist das sehr mühselig!
- Geht das nicht irgendwie einfacher?

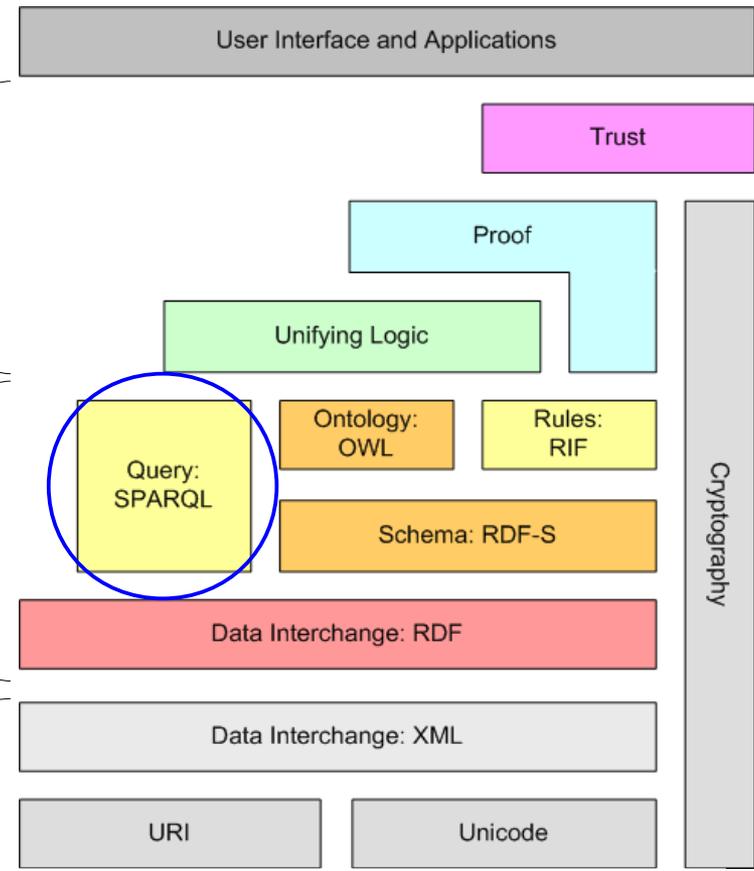
Semantic Web – Aufbau



here be dragons...

Semantic-Web-
Technologie
(Fokus der Vorlesung)

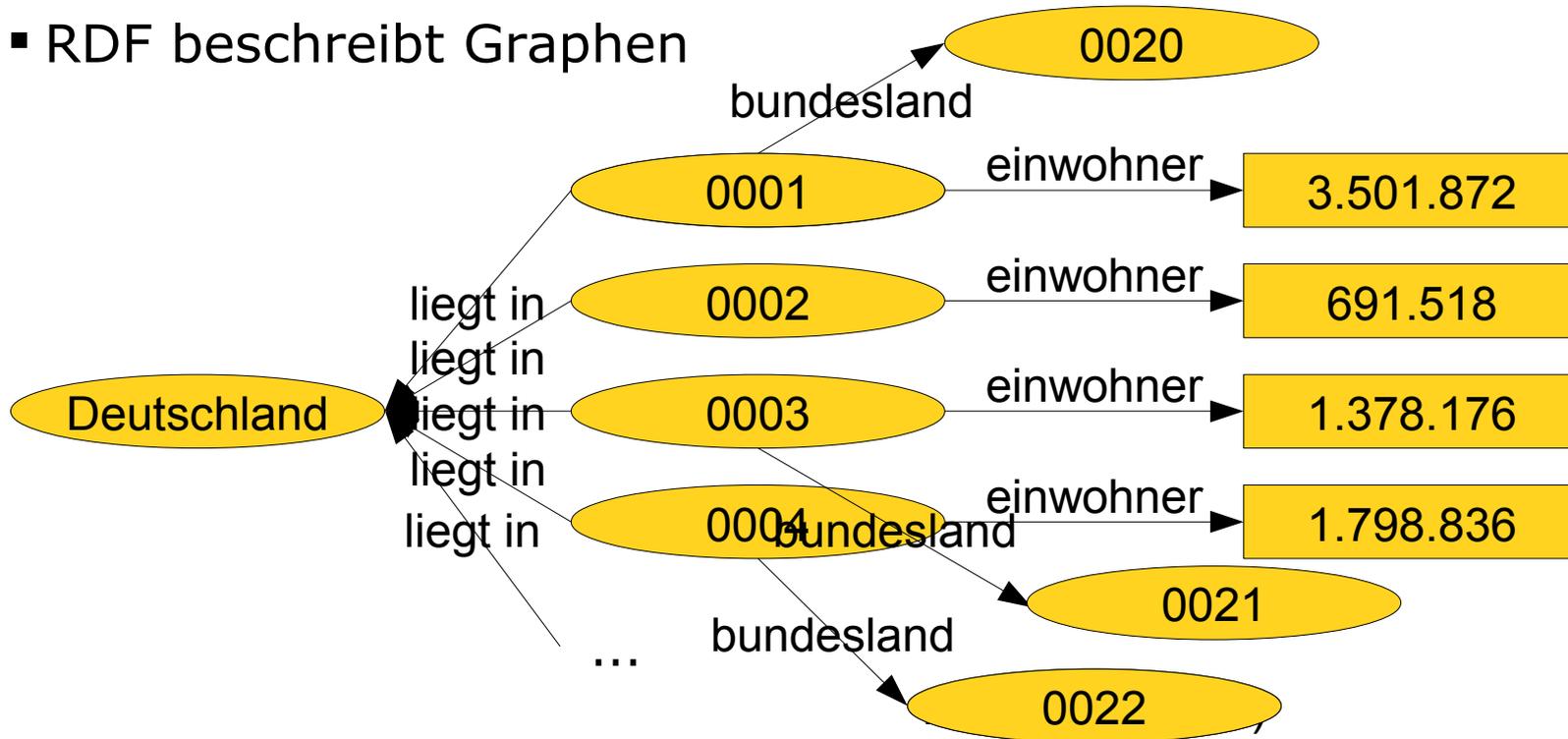
Technische
Grundlagen



Berners-Lee (2009): *Semantic Web and Linked Data*
<http://www.w3.org/2009/Talks/0120-campus-party-tbl/>

Was hätten wir denn gern?

- RDF beschreibt Graphen



Gesucht: eine Abfragesprache für das Semantic Web

- Analog zu SQL für relationale Datenbanken:

```
SELECT      Name, Geburtsdatum FROM Kunden
WHERE       Kundennummer = '00423789'
```

Kundennummer	Name	Geburtsdatum
00183283	Stefan Müller	23.08.1975
00423782	Julia Meyer	05.09.1982
00789534	Gertrud Schäfer	31.03.1953
00423789	Herbert Scholz	02.04.1960
...

Gesucht: eine Abfragesprache für das Semantic Web

- SPARQL: "SPARQL Query Language for RDF"
 - ein rekursives Akronym
- Standardisiert vom W3C (2008)
- Abfragen auf RDF-Graphen
- Übung 1, Aufgabe 3:
 - Merke: XPath und RDF/XML ist eine ganz schlechte Idee!
 - Etwas besseres lernen wir in Kürze kennen...



Hello SPARQL!

▪ Beispiel:

```
SELECT ?child  
WHERE { :Stefan :vaterVon ?child }
```

Ausdrücke mit ?
kennzeichnen
Variablen



SPARQL: Grundkonzepte



- Grundstruktur:

```
SELECT <Variablenliste>  
WHERE { <Muster> }
```

- Variablen mit ?

- Namensräume: wie in RDF/N3:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
SELECT ?person ?name  
WHERE { ?person foaf:name ?name }
```

SPARQL: Grundkonzepte

- Der WHERE-Teil ist ähnlich wie N3-Notation
 - mit Variablen
- `{?p foaf:name ?n }`
- `{?p foaf:name ?n; foaf:homepage ?hp }`
- `{?p foaf:knows ?p1, ?p2 }`

SPARQL: Pattern Matching auf RDF-Graphen

- WHERE-Teil der Abfrage: ein RDF-Graph mit Variablen

```
SELECT ?person1 ?person2
WHERE {
  ?person1 :kennt ?anderePerson .
  ?anderePerson :vaterVon ?person2 .
}
```

über gemeinsame Variablen wird ein komplexes Muster definiert

- Ergebnis:

- ?person1 = :Peter, ?person2 = :Julia



SPARQL: Matching auf Graphen

- Eine Person, die eine Tochter und einen Sohn hat
{ ?p :hatTochter ?t ; :hatSohn ?s . }
- Eine Person, die zwei Personen kennt, die sich untereinander kennen
{ ?p :kennt ?p1 , ?p2 . ?p1 :kennt ?p2 . }
- ~~▪ Eine Person, die zwei Kinder hat
{ ?p :hatKind ?k1, ?k2 . }~~

Achtung: zwei Variablen müssen nicht automatisch an verschiedene Ressourcen gebunden werden!

SPARQL: Blank Nodes

- WHERE-Teil der Abfrage: ein RDF-Graph mit Variablen

```
SELECT ?person1 ?person2 ?anderePerson
WHERE {
    ?person1 :kennt ?anderePerson .
    ?anderePerson :vaterVon ?person2 . }
```

- Ergebnis:

- ?person1 = :Peter, ?person2 = :Julia; ?anderePerson = _:x1

- Blank Node IDs sind nur eindeutig innerhalb des Result Sets!



SPARQL: Matching von Literalen



- Strings

```
{ ?country :name "Deutschland" . }
```

- Vorsicht bei Sprachangaben:

```
{ ?country :name "Deutschland"@de . }
```

→ Die Strings "Deutschland" und "Deutschland"@de sind verschieden!

- Zahlen:

```
{ ?person :alter "42"^^xsd:int . }
```

oder kürzer:

```
{ ?person :alter 42 . }
```

SPARQL: Filter

- Zur weiteren Eingrenzung von Ergebnissen

```
{?person :age ?age . FILTER(?age < 42) }
```

- Vergleichsoperatoren:

= != < > <= >=

- Logische Verknüpfungen:

&& || !

- Personen, die jüngere Geschwister haben

```
{ ?p1 :geschwisterVon ?p2 .  
  ?p1 :alter ?a1 .  
  ?p2 :alter ?a2 .  
  FILTER(?a2 < ?a1) }
```

- Personen, die sowohl jüngere und ältere Geschwister haben

```
{ ?p1 :geschwisterVon ?p2,p3 .  
  ?p1 :alter ?a1 .  
  ?p2 :alter ?a2 .  
  ?p3 :alter ?a3 .  
  FILTER(?a2 < ?a1 && ?a3 > ?a1) }
```

SPARQL: Filter

- Zweiter Versuch: Eine Person, die zwei Kinder hat

```
{ ?p :hatKind ?k1, ?k2 . FILTER( ?k1 != ?k2) }
```
- Schon mal besser als der erste Versuch
→ Variablen werden jetzt unterschiedlich gebunden
- Aber: es gilt immer noch die Non-Unique Naming Assumption
→ Aus
:Peter :hatKind :Julia .
:Peter :hatKind :Stefan .
folgt immer noch nicht, dass Peter zwei Kinder hat!
- Darüber hinaus gilt die Open World Assumption
→ Peter könnte also auch noch mehr Kinder haben

Filter auf Strings

- Suche in Strings: Reguläre Ausdrücke

- Personen, die "Heinz" heißen

```
{?person :name ?n . FILTER(regex(?n, "^Heinz$")) }
```

```
{?person :name ?n . FILTER(regex(?n, "Heinz")) }
```

→ die zweite Variante findet z.B. auch "Karl-Heinz"

- `str`: URIs und Literale als Strings

- ermöglicht u.a. Suche über String-Literale in allen Sprachen

```
{?country :name ?n . FILTER(str(?n) = "Tyskland") }
```

→ wir lernen: Deutschland heißt auch auf norwegisch "Tyskland".

Weitere eingebaute Funktionen



- Typ einer Ressource abfragen:
 - `isURI`
 - `isBLANK`
 - `isLITERAL`
- Datentyp und Sprache eines Literals abfragen:
 - `DATATYPE(?v)`
 - `LANG(?v)`
- Sprache von zwei Literalen vergleichen:
 - `langMATCHES(?v1, ?v2)`
 - **Achtung:** sei `?v1 = "Januar"@DE`, `?v2 = "Jänner"@DE-at`
`LANG(?v1) = LANG(?v2) → false`
`langMATCHES(?v1, ?v2) → true`

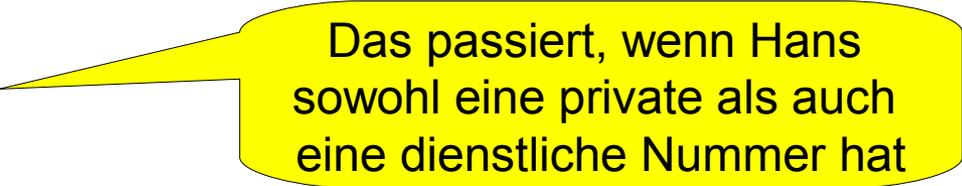
Verknüpfung von Teilmustern

- Finde die private oder dienstliche Telefonnummer

```
    { ?p :privatePhone ?nr }  
UNION { ?p :workPhone ?nr }
```

- UNION erzeugt eine Vereinigungsmenge

```
?p = :peter, ?nr = 123;  
?p = :hans, ?nr = 234;  
?p = :hans, ?nr = 345;  
...
```



Das passiert, wenn Hans sowohl eine private als auch eine dienstliche Nummer hat

Optionale Teilmuster

- Finde die Telefonnummer einer Person und, **falls vorhanden**, auch die Faxnummer

```
OPTIONAL { ?p :phone ?tel }  
          { ?p :fax ?fax }
```

- OPTIONAL erzeugt auch ungebundene Variablen

?p = :peter, ?tel = 123, ?fax = 456;

?p = :hans, ?tel = 234, ?fax = ;

?p = :jutta, ?nr = 978; ?fax = 349;

...

Ungebundene Variable:
Hans hat kein Fax
(soweit wir das wissen)

Ungebundene Variablen

- Variablen können auch ungebunden bleiben
- Mit `BOUND(?v)` kann man das abfragen
- Alle Personen, die Telefon oder Fax haben (oder beides):

```
OPTIONAL {?p :phone ?tel . }  
OPTIONAL {?p :fax ?fax . }  
FILTER ( BOUND(?tel) || BOUND(?fax) )
```

Negation

- Häufige Frage mit Bezug auf SPARQL
- Wie geht so etwas:
 - "Finde alle Personen, die keine Geschwister haben."
- Das hat man in SPARQL bewusst nicht direkt vorgesehen
- Warum?
- Open World Assumption
 - wir können das gar nicht wissen!
- Aus dem selben Grund gibt es (noch) kein COUNT

Negation – Hacking SPARQL

- Es gibt dennoch Möglichkeiten
 - im "Giftschrank" von SPARQL...

- Mit `OPTIONAL` und `BOUND`

- Finde alle Personen ohne Geschwister:

```
OPTIONAL {?p :hasSibling ?s . }  
FILTER ( !BOUND(?s) )
```

- Das funktioniert
- man sollte aber immer wissen, was man tut
 - und wie die Ergebnisse zu interpretieren sind!



Negation – Hacking SPARQL



- Wie funktioniert das?
- Ergebnisse vor FILTER:

```
OPTIONAL {?p :hasSibling ?s . }
```

```
?p = :peter, ?s = :julia
```

```
?p = :peter, ?s = :stephan
```

```
?p = :jan, ?s =
```

```
?p = :paul, ?s =
```

Ungebundene Variablen

- Anwendung von FILTER

- `FILTER(!BOUND(?s))`

```
?p = :jan, ?s =
```

```
?p = :paul, ?s =
```

Sortieren der Ergebnisse



- **Sortierung:** ORDER BY ?name
- **Begrenzung:** LIMIT 100
- **Untere Grenze:** OFFSET 200

- **Beispiel: die Personen 101-200, nach Namen sortiert**
 - ORDER BY ?name LIMIT 100 OFFSET 100

- LIMIT/OFFSET **ohne** ORDER BY:
 - Ergebnisse nicht deterministisch
 - Es gibt keine default-Ordnung!

Ausfiltern von Duplikaten

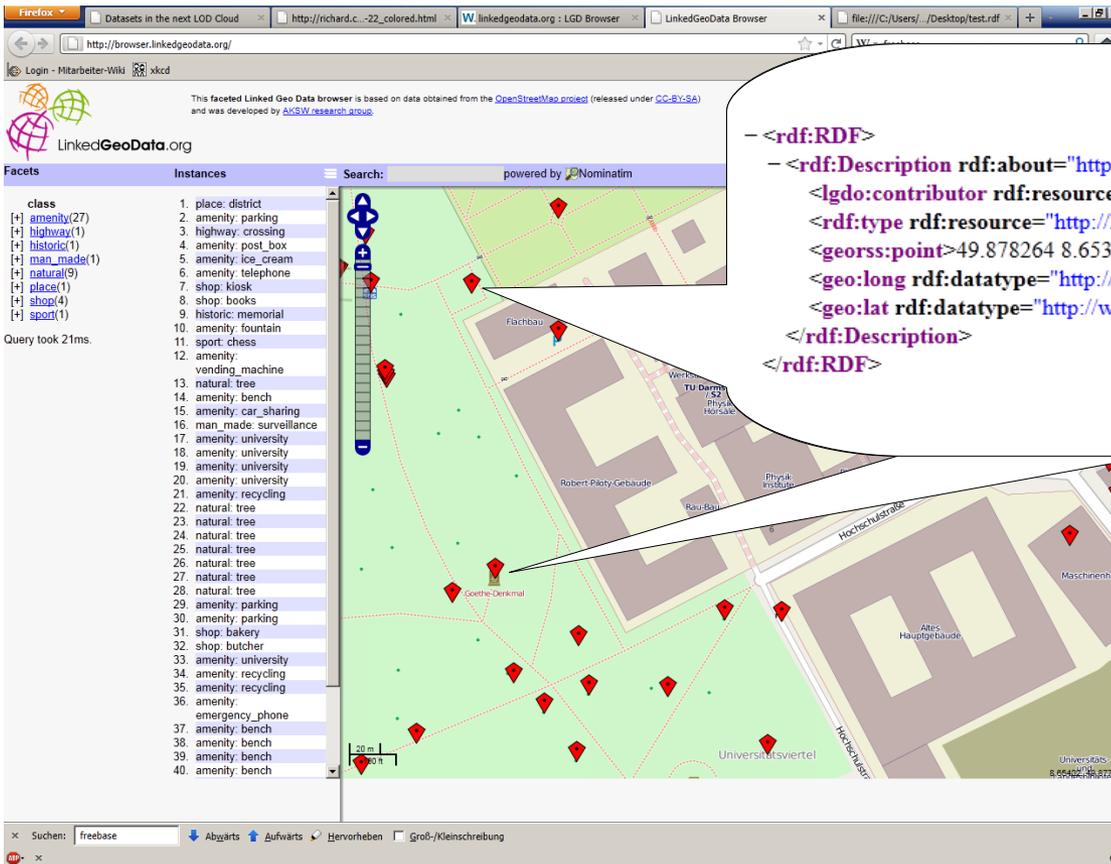


- ```
SELECT DISTINCT ?person
 WHERE { ?person :privatePhone ?nr }
 UNION { ?person :workPhone ?nr }
```
- Bedeutet: Es werden alle Ergebnisse mit identischer Wertebelegung der Variablen ausgefiltert
- Bedeutet nicht: die Personen, die durch die Werte von ?person identifiziert werden, sind tatsächlich verschieden!
- Warum?
  - Non-unique naming assumption

# Custom Built-Ins

- Manche Anbieter von Endpoints erlauben zusätzliche Filter
- sog. Custom Built-Ins
- Beispiel Linked Geo Data

- hat auch ein eigenes User Interface:



```
- <rdf:RDF>
- <rdf:Description rdf:about="http://linkedgedata.org/triplify/node371158882">
 <lgdo:contributor rdf:resource="http://linkedgedata.org/triplify/user61927"/>
 <rdf:type rdf:resource="http://linkedgedata.org/ontology/Chess"/>
 <georss:point>49.878264 8.6537455</georss:point>
 <geo:long rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">8.6537455</geo:long>
 <geo:lat rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">49.878264</geo:lat>
</rdf:Description>
</rdf:RDF>
```

- Abfrage nach Koordinaten

- naiv:

```
WHERE { ?x geo:long ?long; geo:lat ?lat }
FILTER (?long >8.653, ?long < 8.654,
 ?lat >49.878, ?lat < 49.879)
```

- Komplexere Anfragen

- alle Cafés, die sich im Umkreis von 1km  
von einem bestimmten Punkt befinden

```
WHERE { ?x rdf:type lgdo:Cafe; geo:geometry ?geo }
FILTER (bif:st_intersects(?geo,
 bif:st_point(8.653, 49.878), 1))
```

# Custom Built-Ins

- Noch komplexere Anfragen
  - alle Cafés, die sich im Umkreis von 1km von einer Universität befinden

```
WHERE { ?x rdf:type lgdo:Cafe; geo:geometry ?cafegeo .
 ?y rdf:type lgdo:University; geo:geometry ?ugeo . }
FILTER (bif:st_intersects(?cafegeo, ?ugeo, 1))
```

# Weitere Abfragearten: ASK

- Bis jetzt haben wir nur SELECT kennen gelernt
- Mit ASK kann man Ja/Nein-Fragen stellen:  
Gibt es Personen mit Geschwistern?

```
ASK {?p :hasSibling ?s . }
```

- Die Antwort ist *true* oder *false*
  - wobei *false* heißt, dass keine passenden Daten gefunden wurden
  - darf man nicht falsch interpretieren (Open World Assumption)

# Weitere Abfragearten: DESCRIBE



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Alle Eigenschaften einer Ressource:

```
DESCRIBE <http://dbpedia.org/resource/Berlin>
```

- Auch mit WHERE-Klausel

```
DESCRIBE ?city WHERE { :Hans :livesIn ?city . }
```

- Ermöglicht das Explorieren eines Datensets, dessen Struktur unbekannt ist
- Achtung: Nicht-normativ, Ergebnisse variieren je nach Implementierung!



# Weitere Abfragearten: CONSTRUCT

- Erzeugen eines neuen RDF-Graphen

```
CONSTRUCT
{ ?x rdfs:seeAlso <http://dbpedia.org/resource/Berlin> . }
WHERE { <http://dbpedia.org/resource/Berlin> ?y ?x .
 FILTER (isURI(?x)) }
```

- Das Ergebnis ist ein kompletter RDF-Graph
  - z.B. zur Weiterverarbeitung

# SPARQL: Zusammenfassung

- Abfragesprache für RDF
- Pattern-Matching auf Graphen
- Mit SPARQL kann man nach Informationen suchen, nicht nur über Graphen wandern
  
- Abfrageergebnisse unterliegen der Semantik von RDF!
  - Open World Assumption
  - Non-unique naming assumption



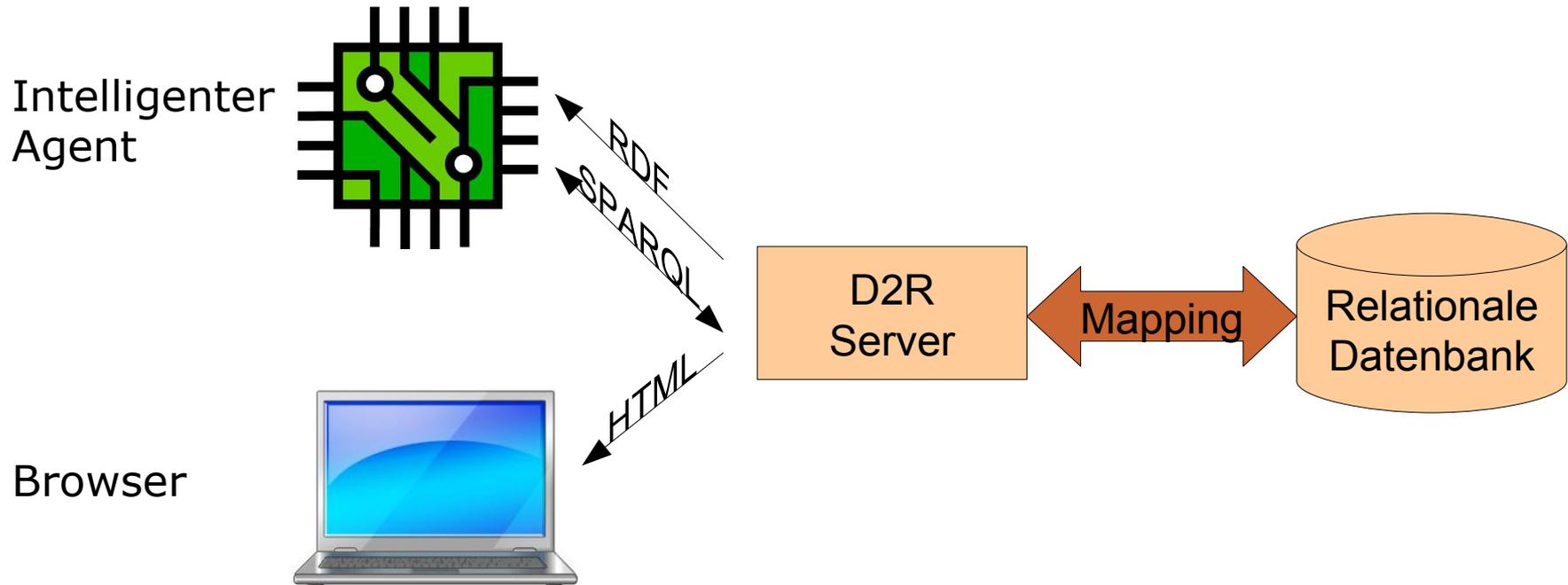
# Implementierung

- Interface SPARQLQuery
  - Hat eine Methode (query())
  - Vom W3C komplett spezifiziert
- Umsetzung:
  - als Web Service (WSDL)
  - als HTTP-Schnittstelle
- Eine solche Schnittstelle nennt man *SPARQL Endpoint*

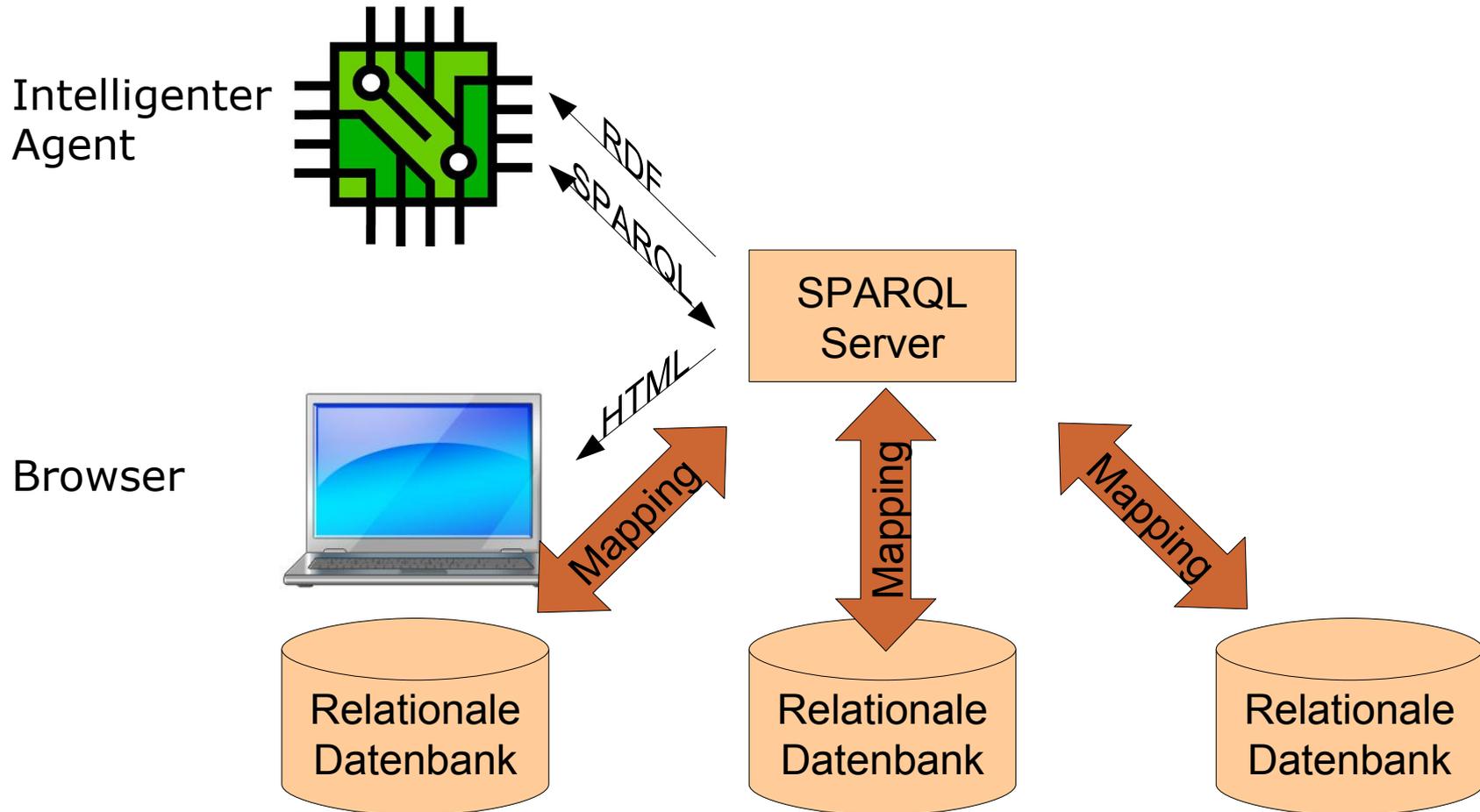


# Implementierung

- Viele Linked Open Data Server haben auch eine SPARQL Schnittstelle:



# Anwendung: Datenintegration



# Anwendung: Datenintegration



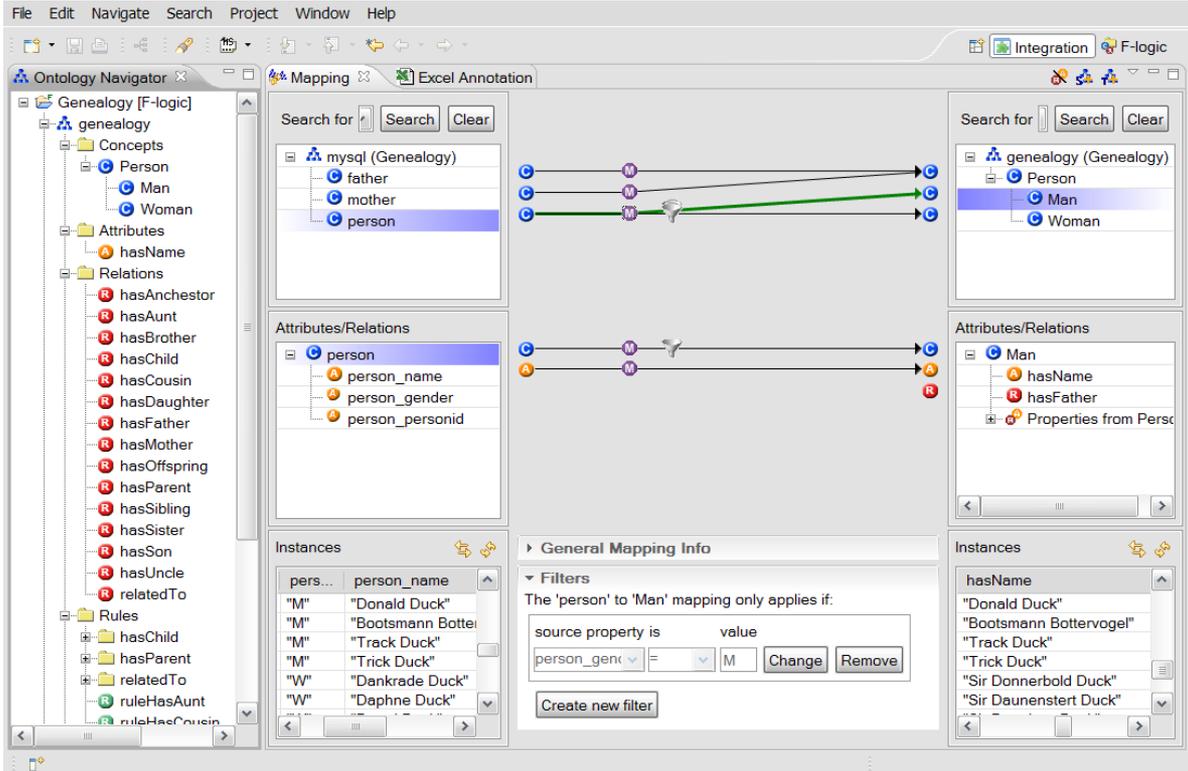
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Viele Datenbanken hinter einer SPARQL-Schnittstelle
- Nur noch ein RDF-Schema statt vieler DB-Schemata
- Zusammenhänge zwischen Datenbanken entdecken
  
- Funktioniert prinzipiell auch mit anderen Quellen
  - Strukturierte Dokumente (XML, CSV, ...)
  - Anwendungen, z.B. mit Web-Service-Schnittstellen



# Anwendung: Datenintegration

- Beispiel für ein kommerzielles Produkt: OntoStudio/OntoBroker



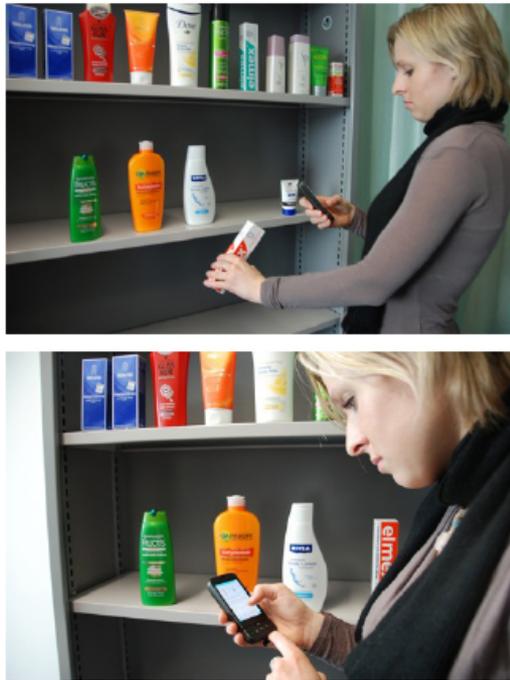
The screenshot displays the OntoStudio/OntoBroker interface with the following components:

- Ontology Navigator:** Shows a tree structure for 'Genealogy [F-logic]' with categories like Concepts (Person, Man, Woman), Attributes (hasName), and Relations (hasAncestor, hasAunt, hasBrother, hasChild, hasCousin, hasDaughter, hasFather, hasMother, hasOffspring, hasParent, hasSibling, hasSister, hasSon, hasUncle, relatedTo). It also includes a 'Rules' section with 'ruleHasAunt' and 'ruleHasCousin'.
- Mapping:** Shows a mapping between 'mysql (Genealogy)' and 'genealogy (Genealogy)'. The 'mysql' side has classes 'father', 'mother', and 'person'. The 'genealogy' side has classes 'Person', 'Man', and 'Woman'. Arrows indicate mappings: 'father' to 'Person', 'mother' to 'Person', and 'person' to 'Man'.
- Attributes/Relations:** Shows the 'person' class from 'mysql' with attributes 'person\_name', 'person\_gender', and 'person\_personid'. The 'Man' class from 'genealogy' has attributes 'hasName' and 'hasFather'.
- Instances:** A table lists instances for 'person' in 'mysql':

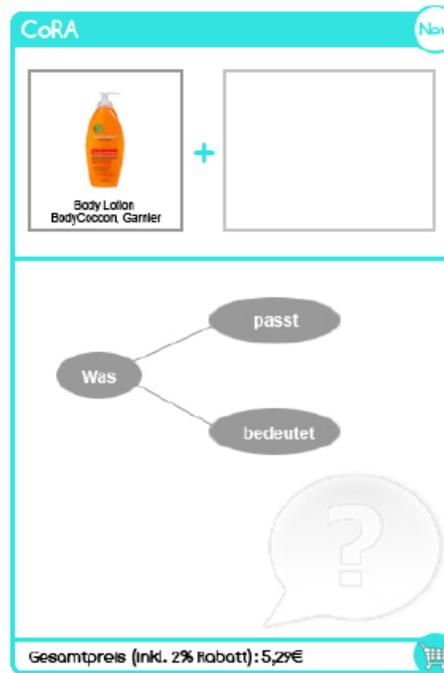
| pers... | person_name        |
|---------|--------------------|
| "M"     | "Donald Duck"      |
| "M"     | "Bootsmann Botter" |
| "M"     | "Track Duck"       |
| "M"     | "Trick Duck"       |
| "W"     | "Dankrade Duck"    |
| "W"     | "Daphne Duck"      |
- General Mapping Info:** Shows a filter for the 'person' to 'Man' mapping: 'The 'person' to 'Man' mapping only applies if: source property is person\_gender, value M. Buttons for 'Change' and 'Remove' are present.
- Instances (genealogy):** Lists instances for 'hasName': "Donald Duck", "Bootsmann Bottervogel", "Track Duck", "Trick Duck", "Sir Donnerbold Duck", "Sir Daunenstert Duck".

OntoPrise (2011): <http://www.ontoprise.de/>

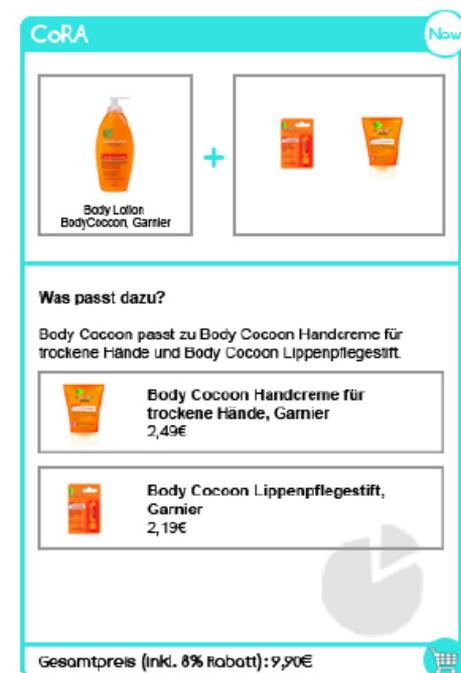
# Beispiel: Integration mehrerer Produktdatenbanken



**Fig. 5.** Subject with CoRA in front of a product shelf



**Fig. 6.** Step-by-step composition of a question



**Fig. 7.** Presentation of the answer

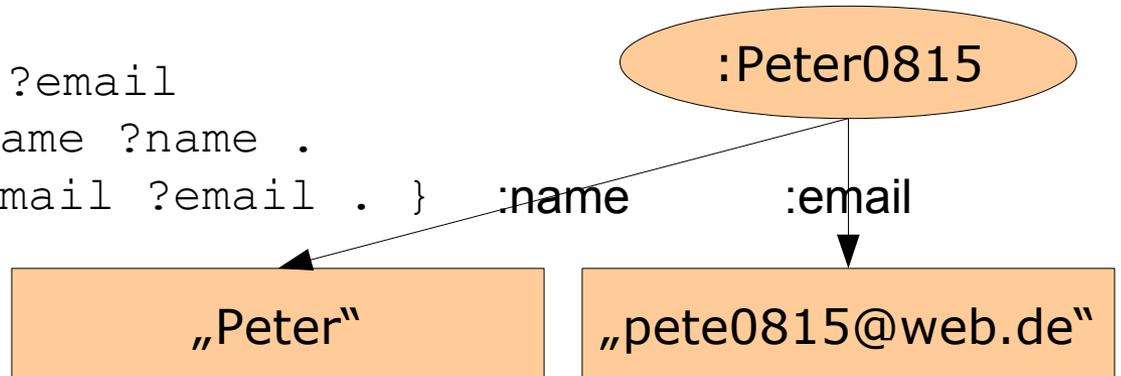
Janzen et al. (2010): Linkage of Heterogeneous Knowledge Resources within In-store Dialogue Interaction. In: International Semantic Web Conference 2010.

# Umsetzung von SPARQL auf SQL

- Daten sind in relationalen DB gespeichert
  - werden mit SQL angesprochen
- Query-Interface: SPARQL
  - Umsetzung von SPARQL auf SQL benötigt
  - je nach Implementierung unterschiedlich

## ▪ Beispiel:

```
SELECT ?person ?name ?email
WHERE {
 ?person :name ?name .
 ?person :email ?email . } :name :email
```



# Umsetzung von SPARQL auf SQL

- Recap: Beispiel naiver Tripel-Store

| Subjekt                 | Prädikat                  | Objekt                  |
|-------------------------|---------------------------|-------------------------|
| <http://foo.bar/Peter>  | <http://foo.bar/vaterVon> | <http://foo.bar/Stefan> |
| <http://foo.bar/Peter>  | <rdf:type>                | <http://foo.bar/Person> |
| <http://foo.bar/Stefan> | <rdf:type>                | <http://foo.bar/Person> |
| <http://foo.bar/Peter>  | <http://foo.bar/vaterVon> | <http://foo.bar/Julia>  |
| <http://foo.bar/Peter>  | <http://foo.bar/kennt>    | _:genID01               |
| _:genID01               | <http://foo.bar/vaterVon> | <http://foo.bar/Markus> |
| ...                     | ...                       | ...                     |

# Umsetzung von SPARQL auf SQL



- Naiver Triple-Store
- SPARQL-Beispiel:

```
SELECT ?person ?name ?email
WHERE {
 ?person :name ?name .
 ?person :email ?email . }
```

- wird zu

```
SELECT t1.subjekt AS person, t1.objekt AS name,
 t2.objekt AS email
FROM triples AS t1, triples AS t2
WHERE t1.predicate = "foo:name"
 AND t2.predicate = "foo:email"
 AND t1.subjekt = t2.subjekt
```

# Umsetzung von SPARQL auf SQL



- Recap: Property Table

| Subjekt    | rdf:type   | foo:vaterVon | foo:kennt |
|------------|------------|--------------|-----------|
| foo:Peter  | foo:Person | foo:Stefan   | NULL      |
| foo:Peter  | foo:Person | foo:Julia    | NULL      |
| foo:Stefan | foo:Person | NULL         | _:genID01 |
| _:genID01  | foo:Person | foo:Markus   | NULL      |
| ...        | ...        | ...          |           |

# Umsetzung von SPARQL auf SQL



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Property Table

- SPARQL-Beispiel:

```
SELECT ?person ?name ?email
WHERE {
 ?person :name ?name .
 ?email :email ?email . }
```

- wird zu

```
SELECT subjekt AS person, name, email
FROM properties
```

# Umsetzung von SPARQL auf SQL

- Recap: Vertikale Partitionierung

| Subjekt   | rdf:type   |
|-----------|------------|
| foo:Peter | foo:Person |
| _:genID01 | foo:Person |
| ...       | ...        |

| Subjekt   | foo:vaterVon |
|-----------|--------------|
| foo:Peter | foo:Stefan   |
| foo:Peter | foo:Julia    |
| _:genID01 | foo:Markus   |
| ...       | ...          |

| Subjekt    | foo:kennt |
|------------|-----------|
| foo:Stefan | _:genID01 |
| ...        | ...       |

# Umsetzung von SPARQL auf SQL



- Vertikale Partitionierung
- SPARQL-Beispiel:

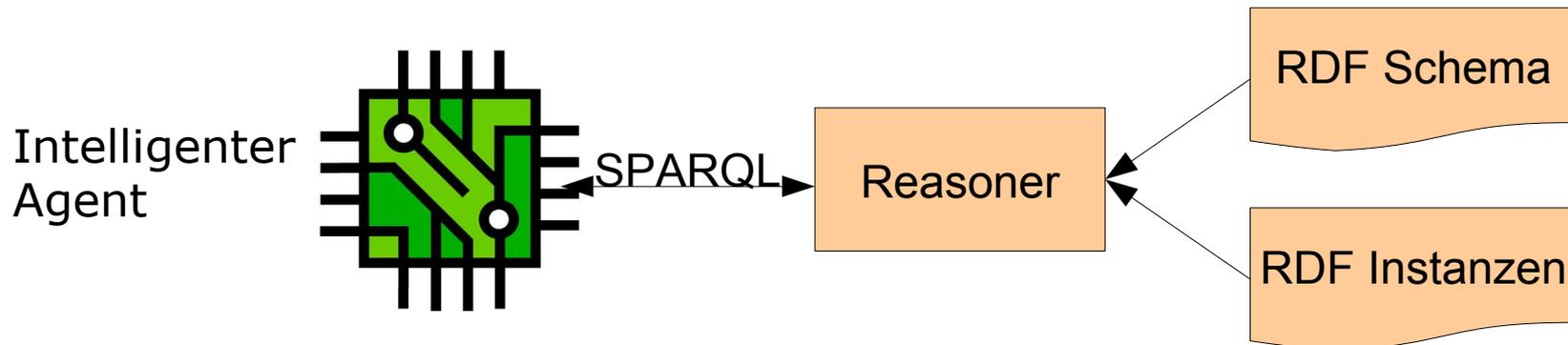
```
SELECT ?person ?name ?email
WHERE {
 ?person :name ?name .
 ?email :email ?email . }
```

- wird zu

```
SELECT t1.subjekt AS person, t1.name, t2.email
FROM name_table AS t1, email_table AS t2
WHERE t1.subjekt = t2.subjekt
```

# Kombination von SPARQL & Reasoning

- Reasoning mit RDF Schema haben wir schon kennen gelernt
- Viele Reasoner haben auch eine SPARQL-Schnittstelle



# Kombination von SPARQL & Reasoning

## ▪ Beispiel-Ontologie

```
:Country a rdfs:Class .
:City a rdfs:Class .
:locatedIn a rdf:Property .
:capitalOf rdfs:subPropertyOf :locatedIn .
:capitalOf rdfs:domain :City .
:capitalOf rdfs:range :Country .

:Madrid :capitalOf :Spain .
:Barcelona :locatedIn :Spain .
:Spain rdfs:label "Spanien"@de .
```

# Kombination von SPARQL & Reasoning

- Finde die Hauptstadt von Spanien

- z.B. so:

```
SELECT ?x WHERE { ?x :capitalOf :Spain . ?x a :City . }
```

```
SELECT ?x WHERE { ?x :capitalOf ?y . ?x a :City .
 ?y rdfs:label "Spanien"@de . }
```

- aber nicht so:

```
SELECT ?x WHERE { ?x :capitalOf ?y . ?x a :City .
 ?y rdfs:label "Spanien" . }
```

# Kombination von SPARQL & Reasoning

- Finde alle Städte in Spanien

- z.B. so:

```
SELECT ?x WHERE { ?x :locatedIn :Spain . ?x a :City . }
```

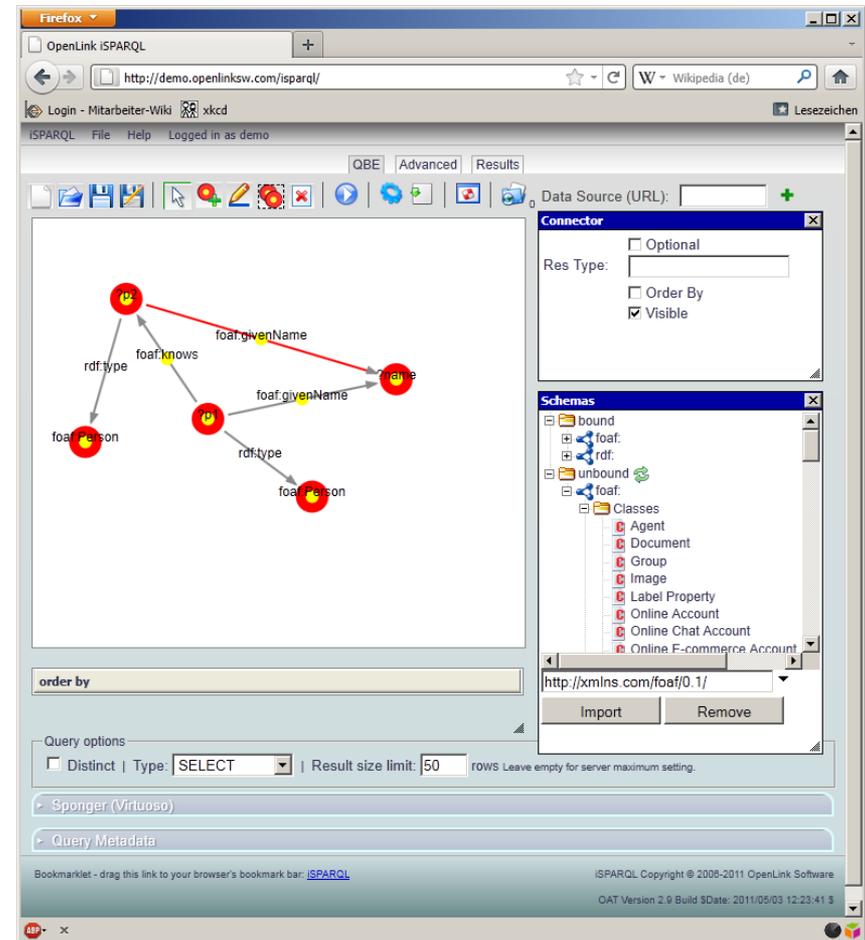
```
SELECT ?x WHERE { ?x :locatedIn ?y . ?x a :City .
 ?y rdfs:label "Spanien"@de . }
```

- Finde alle Städte in Spanien, die nicht Hauptstadt von Spanien sind

```
SELECT ?x WHERE { ?x :locatedIn :Spain . ?x a :City .
 OPTIONAL { ?x :capitalOf ?y }
 FILTER (!BOUND(?y) || ?y != :Spain)
```

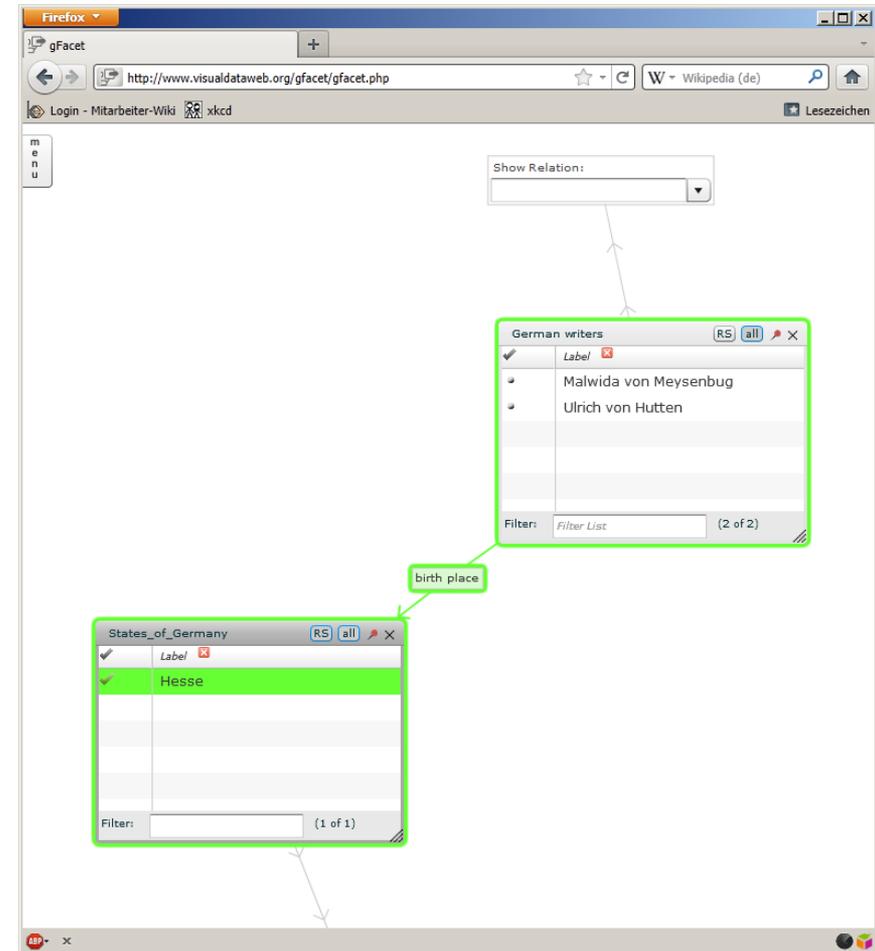
# Visuelle Interfaces

- iSPARQL (2007)
- Query-by-Example
- Klassen und Relationen  
aus Schema per Drag and Drop
- Universität Zürich



# Visuelle Interfaces

- gFacet (2008)
- Exploration von Linked Open Data
- Einschränkungen
  - ausgehend von einem Konzept
  - nur die Kanten, die die Objekte tatsächlich besitzen
- Universität Duisburg-Essen



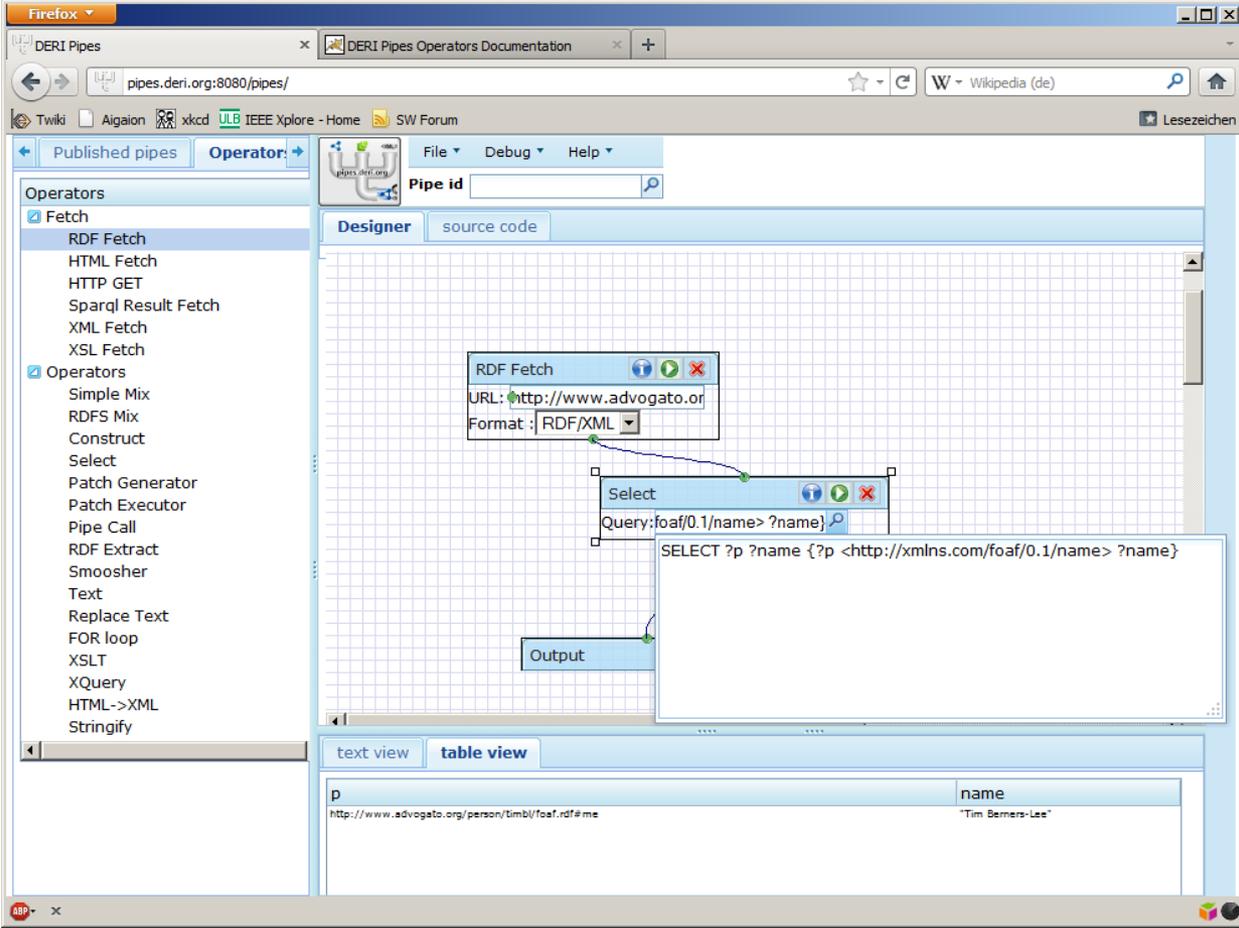
# DERI Pipes



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Daten-Mashup
- Hat auch RDF-Bausteine:
  - RDF von Adresse holen
  - Daten in RDF umwandeln
  - Verschiedene RDF-Dokumente zusammenführen
  - SPARQL-Abfragen

# DERI Pipes



The screenshot displays the DERI Pipes web application. The browser window shows the URL `pipes.deri.org:8080/pipes/`. The interface includes a sidebar with a list of operators, a central designer workspace, and a table view at the bottom.

**Operators List:**

- Fetch
  - RDF Fetch
  - HTML Fetch
  - HTTP GET
  - Sparql Result Fetch
  - XML Fetch
  - XSL Fetch
- Operators
  - Simple Mix
  - RDFS Mix
  - Construct
  - Select
  - Patch Generator
  - Patch Executor
  - Pipe Call
  - RDF Extract
  - Smoocher
  - Text
  - Replace Text
  - FOR loop
  - XSLT
  - XQuery
  - HTML->XML
  - Stringify

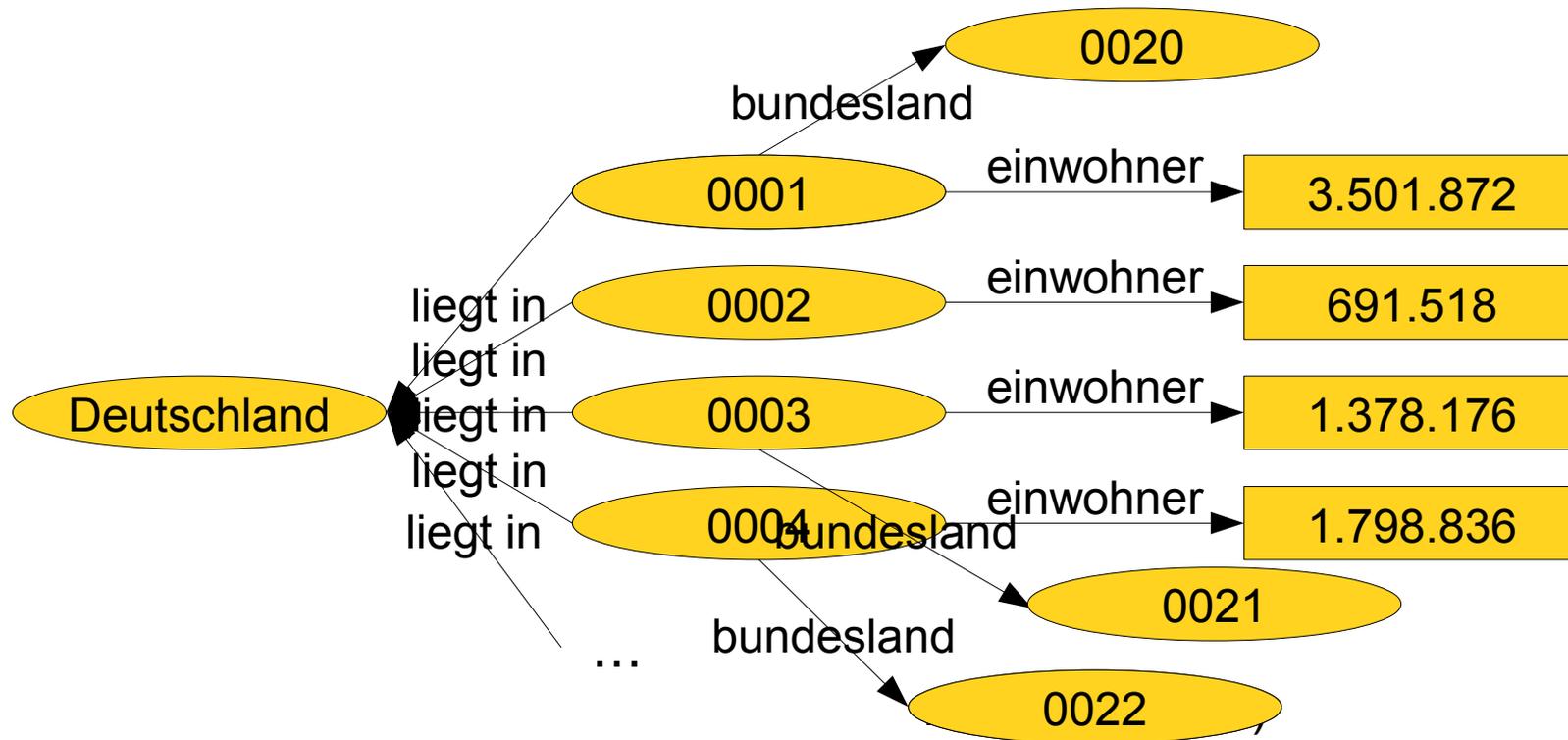
**Designer Workspace:**

- RDF Fetch:** URL: `http://www.advogato.org`, Format: `RDF/XML`
- Select:** Query: `foaf/0.1/name > ?name`
- Output:** A box representing the final output of the pipeline.

**Table View:**

| p                                                             | name            |
|---------------------------------------------------------------|-----------------|
| <code>http://www.advogato.org/person/timbl/foaf.rdf#me</code> | Tim Berners-Lee |

# Zurück zum Ausgangsproblem



# Zurück zum Ausgangsproblem

- Aufgabe: in welchen Bundesländern liegen die fünf größten deutschen Städte?
- Jetzt geht das ganz bequem:

```
SELECT DISTINCT ?bundesland
WHERE
 { ?s locatedIn :Germany .
 ?s bundesland ?l . ?l rdfs:label ?bundesland .
 ?s einwohner ?e}
ORDER BY DESC(?e)
LIMIT 5
```

# Zusammenfassung

- SPARQL
  - eine Abfragesprache für RDF-Daten
  - Beschreibung von (Sub-)Graphen-Mustern
- Built-ins sind möglich
- Weiterverarbeitung im Triple Store
- Kombination mit Reasoning
- Abfrageergebnisse
  - unterliegen Open World Assumption
  - Verneinung daher nur auf Umwegen

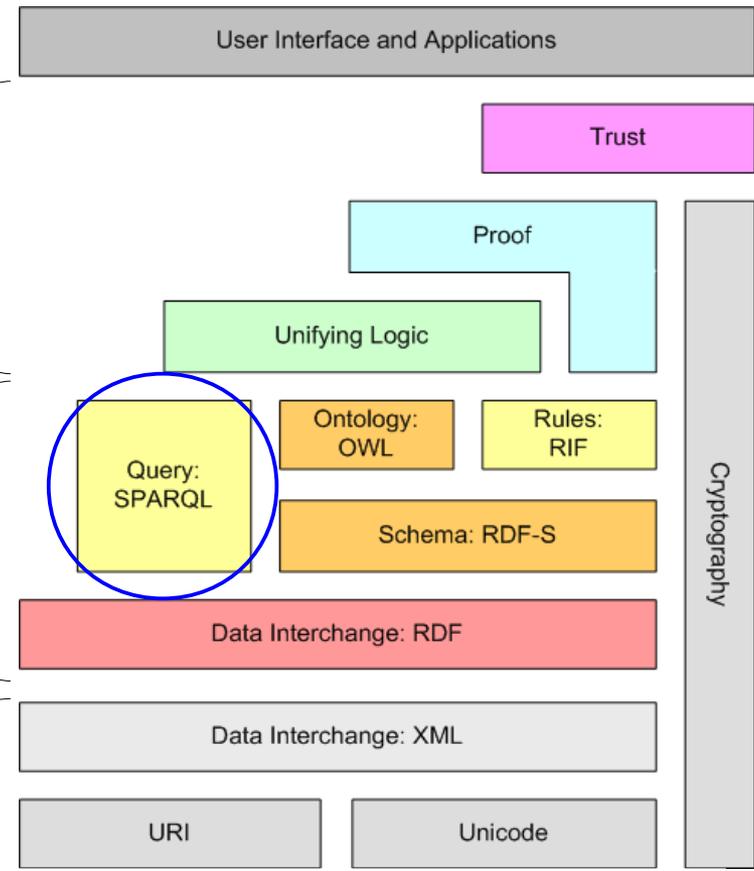
# Semantic Web – Aufbau



here be dragons...

Semantic-Web-  
Technologie  
(Fokus der Vorlesung)

Technische  
Grundlagen



Berners-Lee (2009): *Semantic Web and Linked Data*  
<http://www.w3.org/2009/Talks/0120-campus-party-tbl/>

# Vorlesung Semantic Web



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Vorlesung im Wintersemester 2012/2013

Dr. Heiko Paulheim

Fachgebiet Knowledge Engineering