

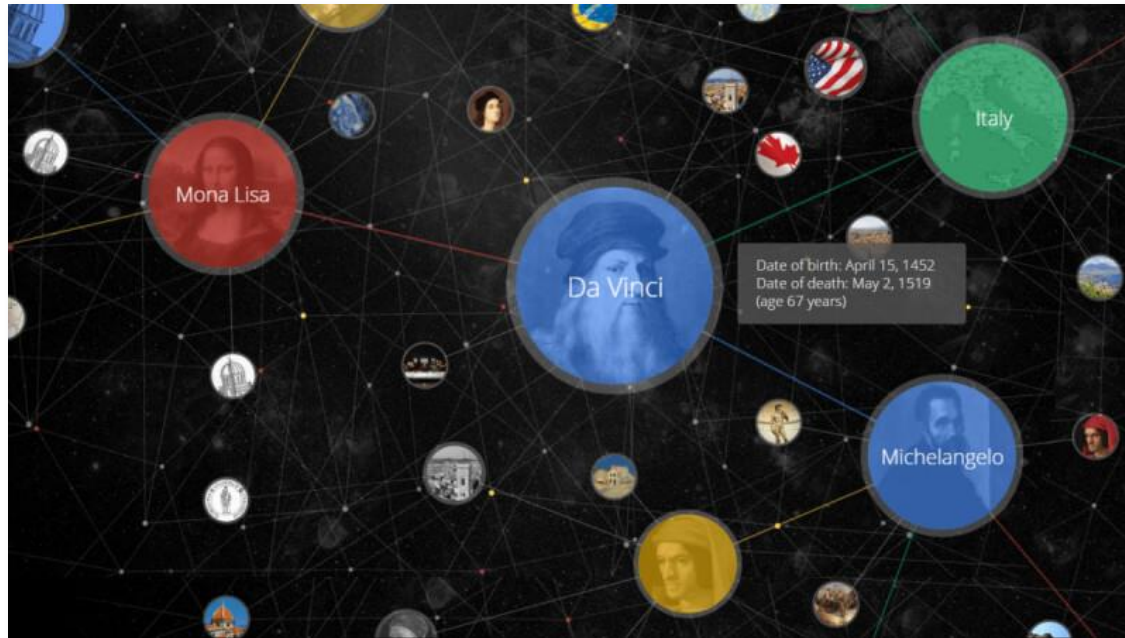
Knowledge Graphs

Knowledge Graphs and Linked Data

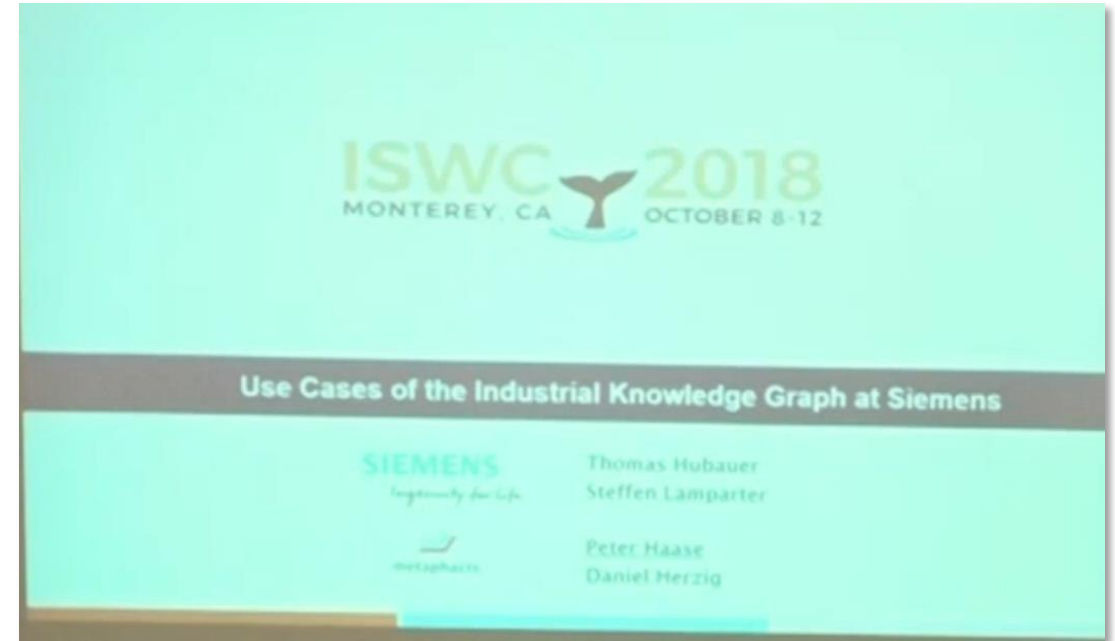
Dr. Tobias Käfer, Deputy Professor at KIT

AI4INDUSTRY SUMMER SCHOOL, VIRTUAL, 2021

What are Knowledge Graphs? – By Example



Google coined the term Knowledge Graph (2012) [1]



Siemens uses Knowledge Graphs in industrial settings [2]

[1] <https://search.googleblog.com/2012/05/introducing-knowledge-graph-things-not.html> <http://www.google.com/insidesearch/features/search/knowledge.html> (Available in the Web Archive)

[2] Hubauer, Lamparter, Haase, Herzig: Use Cases of the Industrial Knowledge Graph at Siemens. In: Proceedings of the industry track at the 17th ISWC 2018

An Inclusive Definition of a Knowledge Graph

■ A Knowledge Graph is...

... “a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities.” [1]

[1] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, Antoine Zimmermann: “Knowledge Graphs”. <https://arxiv.org/abs/2003.02320> (2020)

What are Knowledge Graph Technologies?

Semantic Web Technologies

- Standardised
- Grounded in formal logic
- > 20 years history
- Built for large-scale integration of data from multiple endpoints

- Considerable adoption

Property Graph Technologies

- Typically proprietary
- Only partially formalised
- Younger
- Built to model things as graph and to access data in one endpoint

- Considerable adoption

When are Semantic Web Technologies Applied?

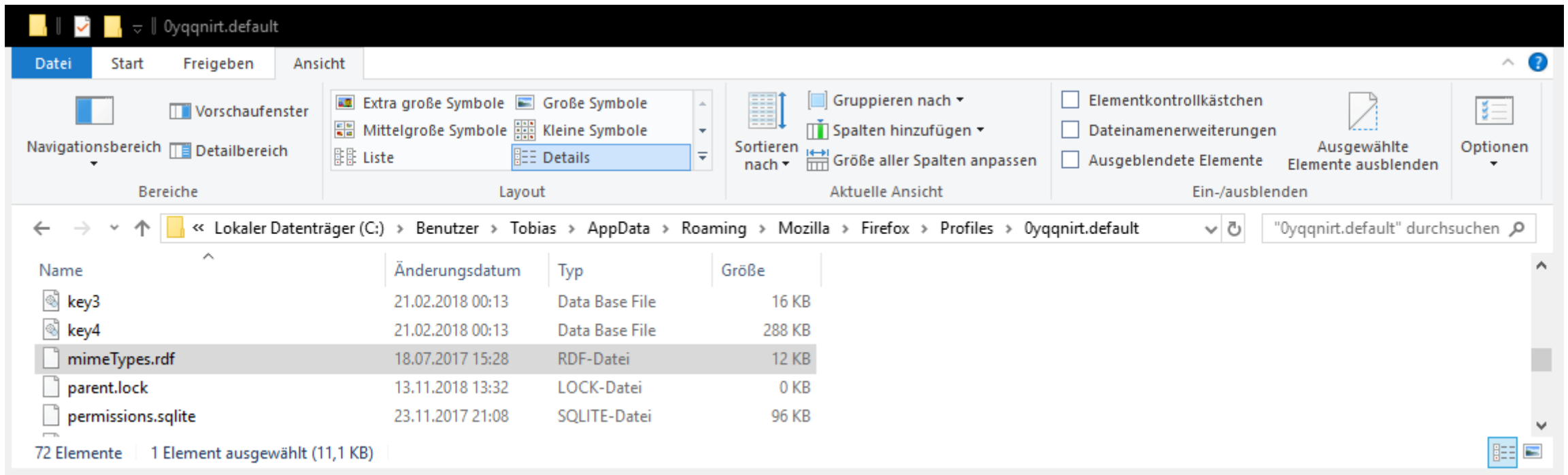
- Graph-based abstraction intuitive in many domains
- Schema heterogeneous and evolving
- Reasoning may be plugged in later
- To integrate different sources
- For data on the web
- ...

Where Are All Those Semantic Technologies?

*“Semantic technology vendors [...] are beginning to learn that their customers don’t want to hear about ontologies, inference rules, and other nuances of the semantic technologies underlying their products. [...] As a result of this dynamic, **semantic technologies are being absorbed into the platform and hidden from users.** This trend will continue as more and more platforms add semantic capabilities and adopt semantic standards.”*

Gartner: “Finding Meaning in the Enterprise: A Semantic Web and Linked Data Primer”, 2011

Are You Using Semantic Technologies?



My Firefox profile folder

Who Else is Using Semantic Technologies?



Collected by Prof. Frank van Harmelen

“Who’s using knowledge graphs?” Only 9 out of 10 of the most value-creating companies in the world

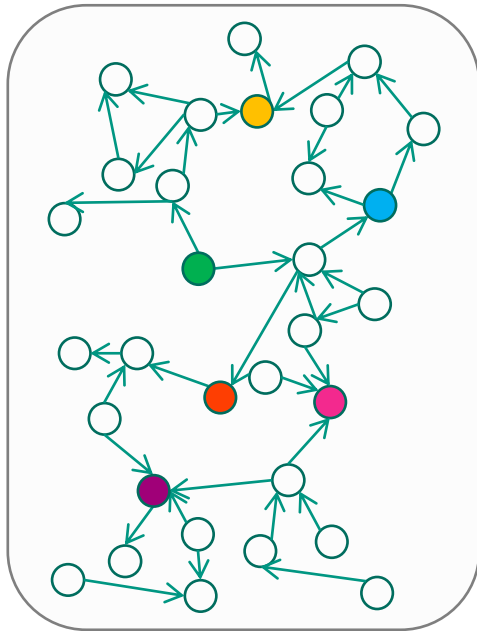
	Company name	Location	Industry	Change in market cap 2009-2018 (\$bn)	Market cap 2018 (\$bn)
Known knowledge graph builders	1 Apple	United States	Technology	757	851
	2 Amazon.Com	United States	Consumer Services	670	701
	3 Alphabet	United States	Technology	609	719
	4 Microsoft Corp	United States	Technology	540	703
Operator of Taobao and AliBot KG builder	5 Tencent Holdings	China	Technology	483	496
	6 Facebook	United States	Technology	383(1)	464
Known KG builders	7 Berkshire Hathaway	United States	Financial	358	492
	8 Alibaba	China	Consumer Services	302(1)	470
	9 JPMorgan Chase	United States	Financials	275	375
	10 Bank of America	United States	Financials	263	307

(1) Change in market cap from IPO date
 (2) Market cap at IPO date
 Source: Bloomberg and PwC analysis

<https://www.linkedin.com/pulse/beyond-low-code-hype-knowledge-graph-driven-alan-morrison>

Three Buzzwords in Context

Knowledge Graphs



The practice of using graphs for data management

Semantic Web



The vision of intelligent agents that operate on graph-structured data on the web and understand humans

Tim Berners-Lee et al. (2001). "The Semantic Web". Scientific American. 2841 (5): 34.

Linked Data



A set of practices to use Semantic Web technologies for publishing data on the web

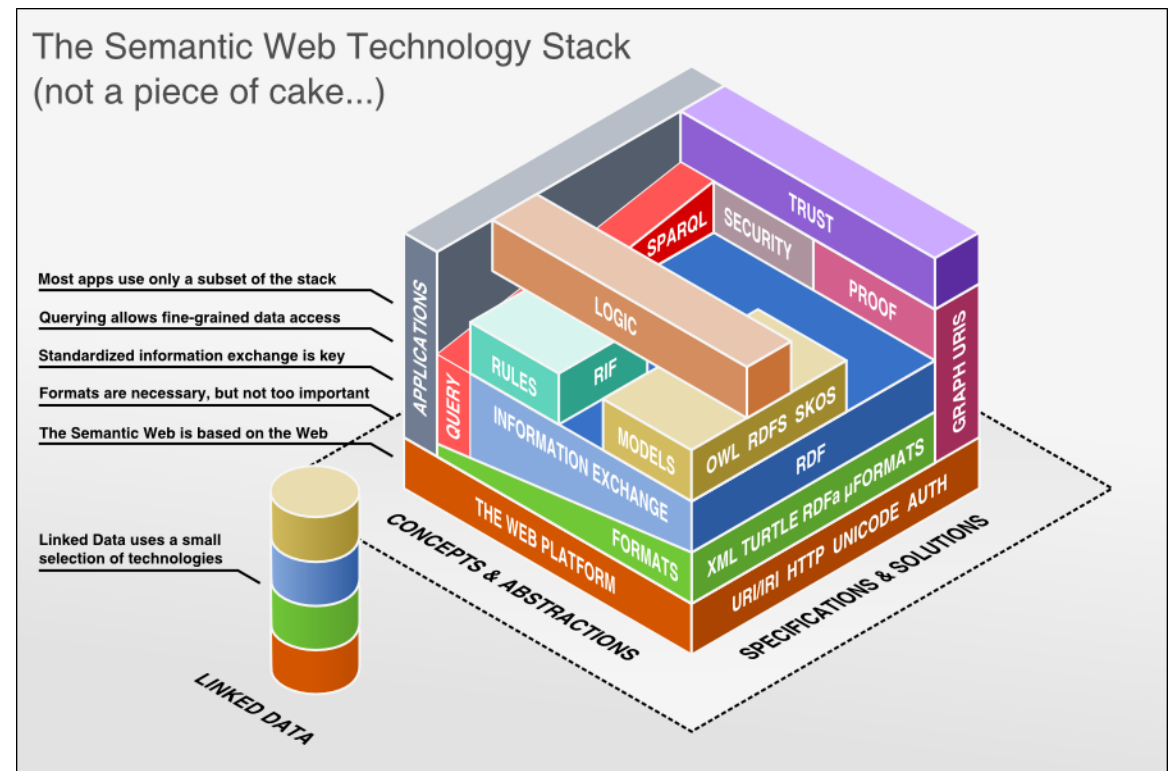
Tim Berners-Lee presenting Linked Data. TED CC-BY-ND

The Linked Data Principles Determine Our Agenda

- Technologies from the Linked Data Principles:
 - URI
 - HTTP
 - RDF(S)
 - SPARQL

- Extensions for Write Access
- Rules for Reasoning, Link Following, and Programming

- Technologies to build systems with Distributed Knowledge Graphs



Source: http://bnode.org/media/2009/07/08/semantic_web_technology_stack.png

Linked Data Principles

- Postulated by Tim Berners-Lee in 2006.

*“The Semantic Web isn't just about putting data on the web. It is about **making links**, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data.”¹*



- Collection of best practices governing the publication and consumption of data on the web
- Aim: unified method for describing and accessing resources
- Later we will also see how to manipulate resource state

¹ <http://www.w3.org/DesignIssues/LinkedData.html>

Linked Data Principles¹

1. Use URIs to **name *things***.
 - Things are not only documents, but also people, locations, concepts, etc.
2. Use HTTP URIs so that users can **look up** those names.
 - Users refer to humans and machine agents alike.
3. When someone looks up a URI, **provide *useful information***, using the standards (**RDF, RDFS, SPARQL**).
 - What “useful” means depends on the data publisher (but the data publisher should return the “useful” data in RDF).
4. Include **links to other URIs**, so that they can **discover** more *things*.

¹ <http://www.w3.org/DesignIssues/LinkedData.html>

Principle 1: Use URIs as Names for Things

- Point on a distinct resource when you share information
- Linked Data follows a *resource*-centered view of data modelling
- Resources are the basic concept of web architecture

- Example:

- Assume we would identify a book via its ISBN (9-781497-364783)
- Using the ISBN scheme from RFC 3187¹ we can use `urn:isbn:9-781497-364783` as resource name for the book



¹ <http://ietf.org/rfc/rfc3187.txt>

Compact URIs (CURIEs)

- We will work a lot with URIs, but full URIs can be unwieldy
- Thus, there is a syntax for abbreviated URIs¹ called Compact URIs, or CURIEs for short²
- CURIEs consist of a prefix (“namespace”) and a local reference (“local part”)
- Assume we declare the prefix `abc` with a value of `http://example.org/doc.ttl#`
- With the prefix `abc` declared, the CURIE `abc:Berlin` expands to `http://example.org/doc.ttl#Berlin`

¹ <http://www.w3.org/TR/curie/>

² CURIEs are an extension to QNames, which are used to abbreviate attribute URIs in XML documents

URIs in Relative Form

- In contrast to absolute HTTP URIs (those starting with `http://` and including a hostname), HTTP URIs can also occur in relative form
- They have to be interpreted *relatively* to an absolute URI
- A URI-reference is either a URI or a relative reference¹
- We can also use the notation known from file systems: “.” refers to the current directory, while “..” refers to the parent directory²

Relative reference	Base URI	Resolves to the URI
<code>research/</code>	<code>http://example.edu/</code>	<code>http://example.edu/research/</code>
<code>./academics/</code>	<code>http://example.edu/research/</code>	<code>http://example.edu/research/academics/</code>
<code>../academics/</code>	<code>http://example.edu/research/</code>	<code>http://example.edu/academics/</code>
<code>#people</code>	<code>http://example.edu/research/</code>	<code>http://example.edu/research/#people</code>
	<code>http://example.edu/doc</code>	<code>http://example.edu/doc</code>

¹ <http://tools.ietf.org/html/rfc3986#section-4.1>

² for detailed technical instructions and further examples: <http://tools.ietf.org/html/rfc3986#section-5.2>

Principle 2: Use HTTP URIs to Allow for Lookup

- Given an identifier for a thing (URI), use HTTP as a mechanism to retrieve more information about that thing
- That is, we require some form of mapping between a
 - **URI as name** (identifying a book, a person, a place or a chemical element) and a
 - **URI as location** (identifying a machine-readable description about the book, the person, the place or the chemical element).

Principle 2: Use HTTP URIs to Allow for Lookup



- Assume we want to know more about a URI-defined resources, say for our book having the URI `urn:isbn:9-781497-364783`
- With the ISBN you can go to your local bookstore, and a clerk there can look up the ISBN in their catalogue
- Or you type the ISBN into a search box of an online bookstore or of a library, to get more information about the book
- Ultimately, there will be a query to a database of things identified via an ISBN, maintained by some organisation

Principle 2: Use HTTP URIs to Allow for Lookup

- HTTP URIs provide an inherent mechanism for lookup and unites logical and physical address
- You can type an identifier into your browser and immediately get some information back → tight connection between identifier and source
- E.g. <http://www.w3.org/People/Berners-Lee/card> is the URI of Tim Berners-Lee's machine-readable homepage
- No additional information or mediator is needed to access information
- Just type HTTP URI into browser and access HTML, JPEG, PNG, GIF, MP4 files – any content that can be serialised into bytes

Referencing a Resource, Dereferencing a URI

- Referencing a resource is easy: just write the URI
- But what about dereferencing?
- How do you get the referenced resource?
- What do you get?

Referencing a Resource, Dereferencing a URI¹

- The act of retrieving a representation of a resource identified by a URI is known as **dereferencing** that URI
- Applications, such as browsers, render the retrieved representation for the user
- Most web users do not distinguish between a resource and the rendered representation they receive by accessing it
- **Information resources** associated with a **resource** need to have their own URIs
- They are themselves distinct resources and provide representations

¹ <https://www.w3.org/2001/tag/doc/httpRange-14/2007-05-31/HttpRange-14>

Referencing a Resource, Dereferencing a URI

- It is important to differentiate between a resource and an informational document **about** that resource¹
- As you cannot retrieve the resource via your browser, a representation is needed



<http://example.org/eiffel-data#Tower>

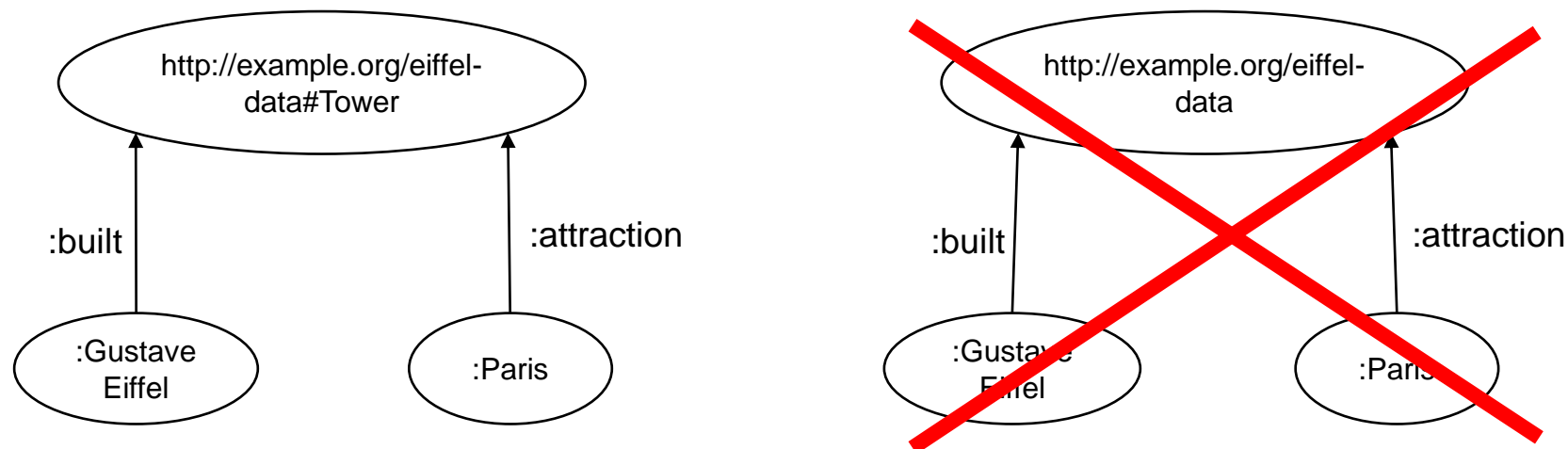


<http://example.org/eiffel-data>

¹ Talking about differentiation: this is also not the Eiffel Tower itself. It is a picture of the Eiffel Tower and the picture's URI is [https://upload.wikimedia.org/wikipedia/commons/thumb/8/85/Tour_Eiffel_Wikimedia_Commons_\(cropped\).jpg/360px-Tour_Eiffel_Wikimedia_Commons_\(cropped\).jpg](https://upload.wikimedia.org/wikipedia/commons/thumb/8/85/Tour_Eiffel_Wikimedia_Commons_(cropped).jpg/360px-Tour_Eiffel_Wikimedia_Commons_(cropped).jpg)

Referencing a Resource, Dereferencing a URI

- As the document about the resource is also a resource itself, it needs its own URI (Information Resource)
- To reference the „Eiffel Tower“, only the URI of the “**resource**” is used:



Referencing a Resource, Dereferencing a URI

- A user that wants information about a given resource might not know the URI of the describing document (the associated information resource)
- In the Semantic Web, two possibilities for providing the information resource of a resource are used: “hash URIs” and “slash URIs”

Resource vs. Information Resource

Hash URIs

- Retrieving the document's URI by stripping off the hash of a hash URI



`http://example.org/karlsruhe-data#Palace`

Resource



`http://example.org/karlsruhe-data`

Information
Resource

Resource vs. Information Resource

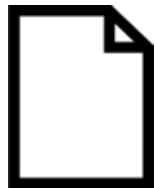
Slash URIs

- Retrieving the document's URI by an automated HTTP redirect (303)



http://dbpedia.org/resource/Karlsruhe_Palace

Resource



http://dbpedia.org/data/Karlsruhe_Palace.ttl

Information
Resource

Resource vs. Information Resource

Thing

http://dbpedia.org/resource/Karlsruhe_Palace

Resource

identifies

describes

Information
Resource

http://dbpedia.org/data/Karlsruhe_Palace.ttl

This document describes the Karlsruhe Palace

```
@prefix dbo: <http://dbpedia.org/ontology/> .  
@prefix dbr: <http://dbpedia.org/resource/> .
```

```
dbr:Karlsruhe_Palace georss:point "49.014 8.404" ;  
dbo:wikiPageExternalLink <http://www.landmuseum.de/website/> ;  
rdf:type yago:Location100027167 ,  
yago:WikicatMuseumsOfAncientRome ,  
yago:Facility103315023 ,  
yago:Whole100003553 .
```

Addressing HTTP-Range 14 using Slash URIs and HTTP Content Negotiation

Let's try an example:

- I want to have **information** about the Karlsruhe Palace from DBpedia

1 **HTTP GET request**
Accept Header: text/html

`http://dbpedia.org/resource/Karlsruhe_Palace`

URI represents *"the name of the thing"*

2 **HTTP/2 303 See Other**

`http://dbpedia.org/page/Karlsruhe_Palace`

3 **HTTP GET request**
Accept Header: text/html

URI represents *"the description of the thing"*

4 **HTML Document**

Image by NordNordWest - Own work, CC BY-SA 3.0 de, <https://commons.wikimedia.org/w/index.php?curid=16921753>

Addressing HTTP-Range 14 using Slash URIs and HTTP Content Negotiation

Let's try an example:

- I want to have **machine-readable information** about the Karlsruhe Palace from DBpedia

① **HTTP GET request**
Accept Header: text/turtle

`http://dbpedia.org/resource/Karlsruhe_Palace`

URI represents *"the name of the thing"*

② **HTTP/2 303 See Other**

`http://dbpedia.org/data/Karlsruhe_Palace.ttl`

③ **HTTP GET request**
Accept Header: text/turtle

URI represents *"the description of the thing"*

④ **RDF (Turtle) Document**

Image by NordNordWest - Own work, CC BY-SA 3.0 de, <https://commons.wikimedia.org/w/index.php?curid=16921753>

Addressing HTTP-Range 14 using Slash URIs and HTTP Content Negotiation

Let's try it ourselves:

- Retrieve **information** about the Karlsruhe Palace from DBpedia

```
curl -L -H "Accept: text/html" http://dbpedia.org/resource/Karlsruhe_Palace
```

- Retrieve **machine readable information** about the Karlsruhe Palace from DBpedia

```
curl -L -H "Accept: text/turtle" http://dbpedia.org/resource/Karlsruhe_Palace
```

Image by NordNordWest - Own work, CC BY-SA 3.0 de, <https://commons.wikimedia.org/w/index.php?curid=16921753>

Principle 3: Provide Useful Information

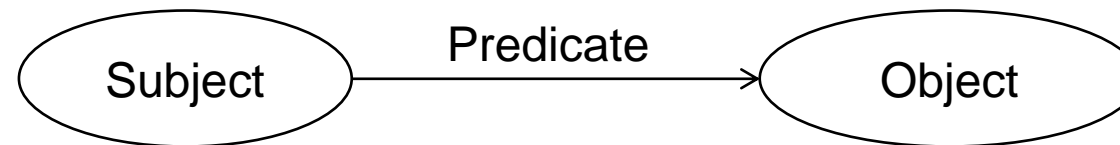
- When somebody looks up a URI, provide useful information using the standards
- **RDF is the data model** for both Semantic Web and Linked Data, providing content meaningful to computational users
- You can eg. write RDF in files, store and query RDF in so-called Triple Stores (databases for RDF), or embed RDF in other formats (eg. HTML)

Resource Description Framework (RDF)



1

- RDF is the foundational data model for both Semantic Web and Linked Data
- RDF comes with a formal underpinning → we can mathematically define and proof things
- An RDF *triple* is the basic RDF concept describing information as a subject-property-object structure
- Property (or predicate) specifies relation between subject and object
- Triples can be visualised:



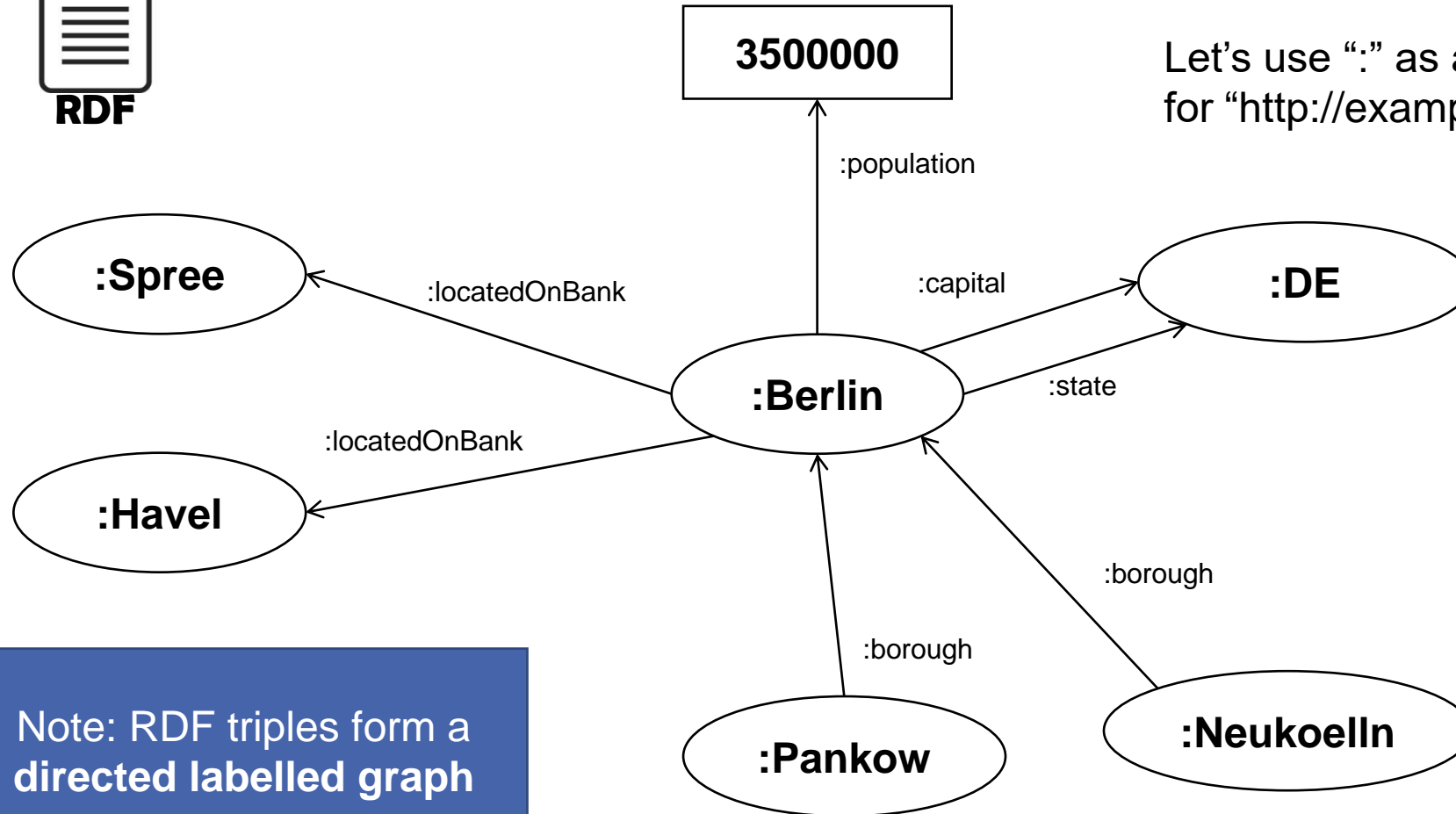
- Multiple triples form an RDF *graph*
- RDF graphs can be visualised as directed labelled graph

¹ <http://www.w3.org/RDF/icons/>

Facts in „Triples“

- Berlin is the capital of Germany.
- Berlin is a state of Germany.
- Berlin has a population of 3.5 Million.
- Berlin is located on the bank of the Spree.
- Berlin is located on the bank of the Havel.
- Pankow is a borough of Berlin.
- Neukölln is a borough of Berlin.

Example RDF Graph within an RDF Document



Let's use ":" as abbreviation for "http://example.org/doc.ttl#"

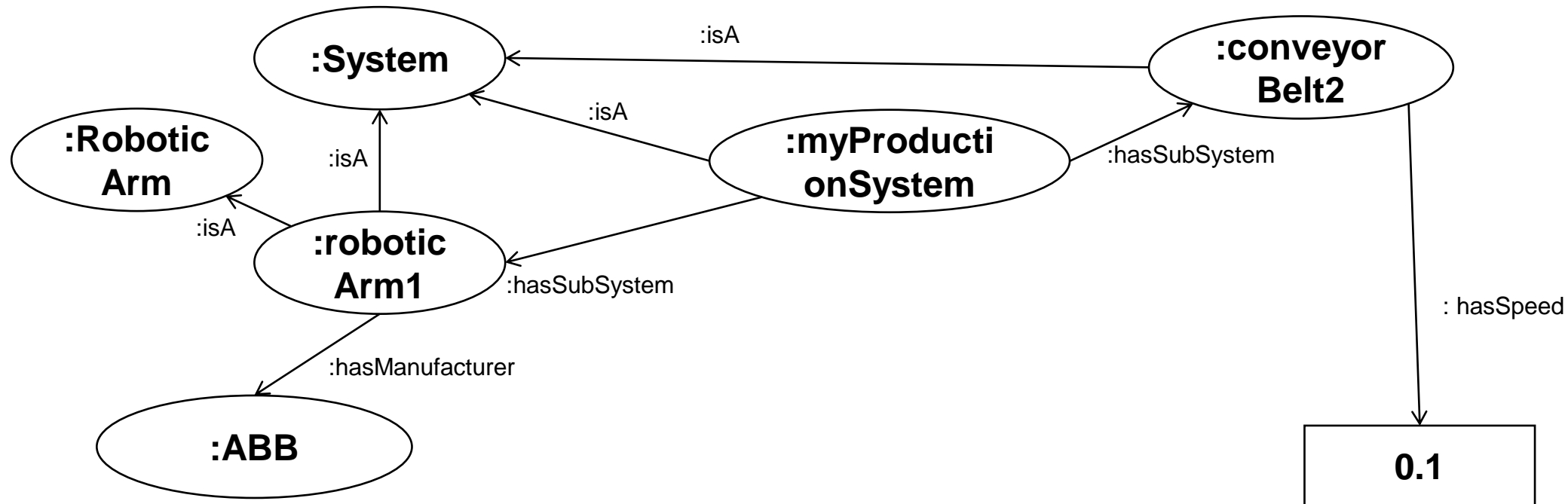
Note: RDF triples form a directed labelled graph

Exercise: Draw an (RDF) Graph

- Have a piece of paper ready
 - Use the facts on the right →
 - Identify connections, things, and values
 - Depict things in circles
 - Depict values in rectangles
 - Depict connections using arrows
 - Draw the graph on a piece of paper
- myProductionSystem is a System
 - myProductionSystem has subsystem roboticArm1
 - myProductionSystem has subsystem conveyorBelt2
 - roboticArm1 is a System
 - roboticArm1 is a RoboticArm
 - roboticArm1 has manufacturer ABB
 - conveyorBelt2 is a System
 - conveyorBelt2 has speed 0.1

Sample Solution

Let's use ":" as abbreviation for "http://example.org/doc.ttl#"



Principle 4: Include Links to Other URIs

- Associating things from one source to things from another source creates the mesh we will later use to perform algorithms on
- Links are required to be able to connect the separate data graphs together
- The graph-structured data model and the re-use of URIs across graphs allows for an easy merging of multiple graphs
- Central points on the web provide URIs for frequently used resources (e.g., DBpedia). Using these allows for a common understanding of descriptions and fast merging of multiple graphs

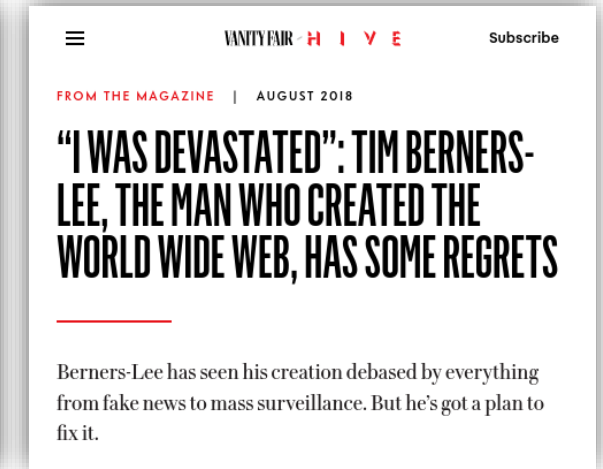
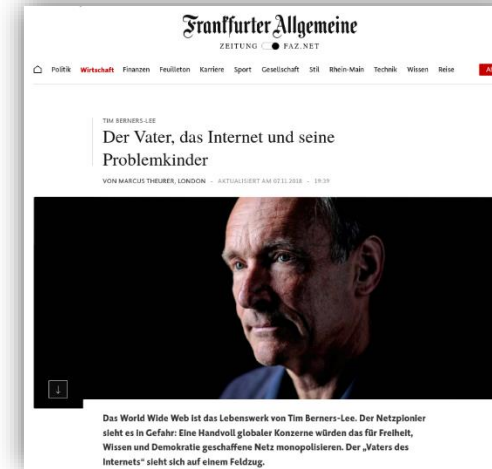


Distributed Knowledge Graphs

- Linked Data builds on HTTP
- Everybody can run a web server

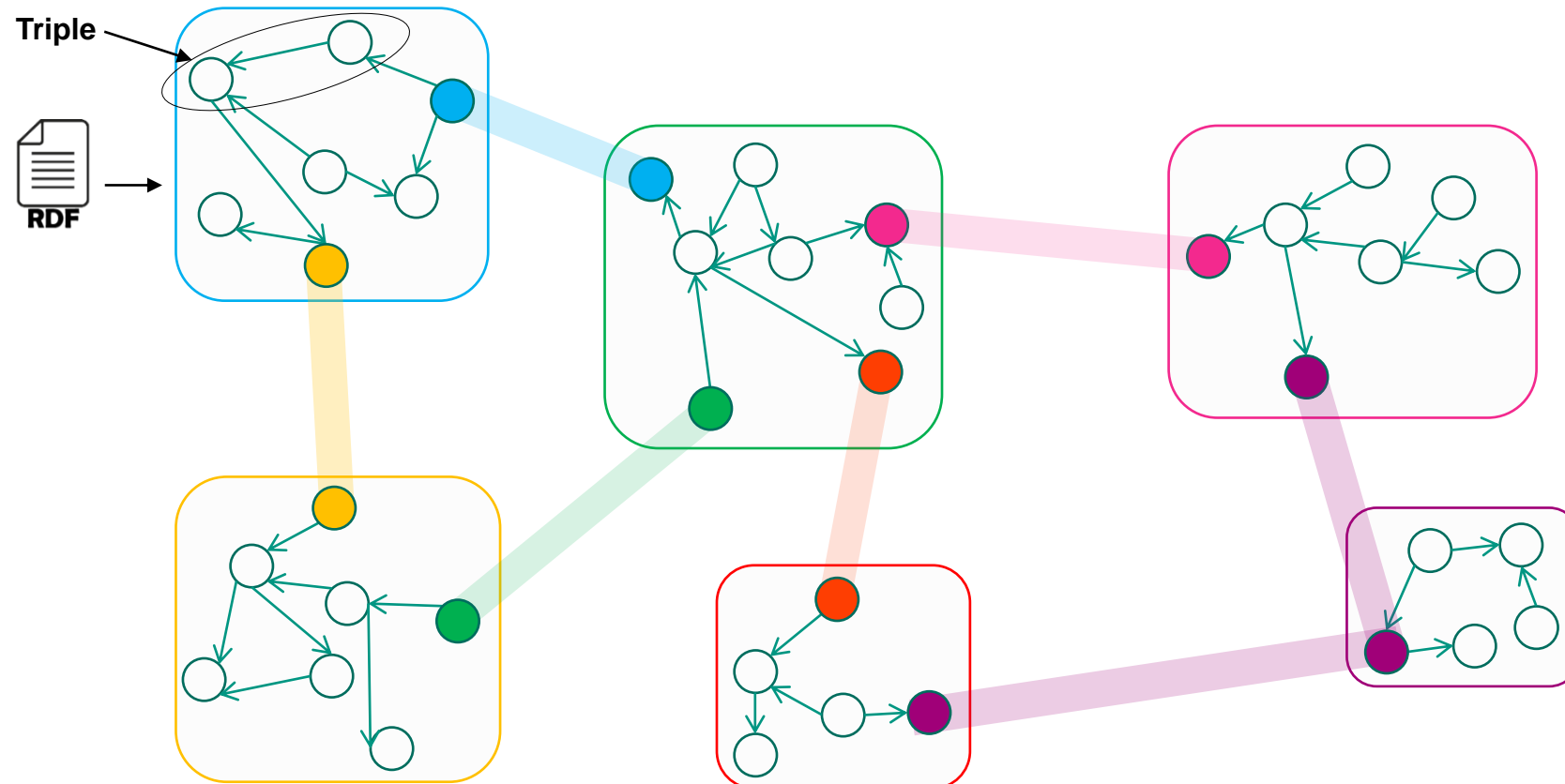
```
python3 -m http.server
```

 - Serves the contents of the working directory (which may contain RDF documents)
- vs. centralised systems of today (Facebook & Co.)
- Decentralised publishing → a distributed system
- Research challenge: Systems/algorithms/... that deal with large amounts of small interlinked RDF documents on the web



¹ K. Brooker, *Vanit. Fair*, 2018, **60**(696), 62-67.

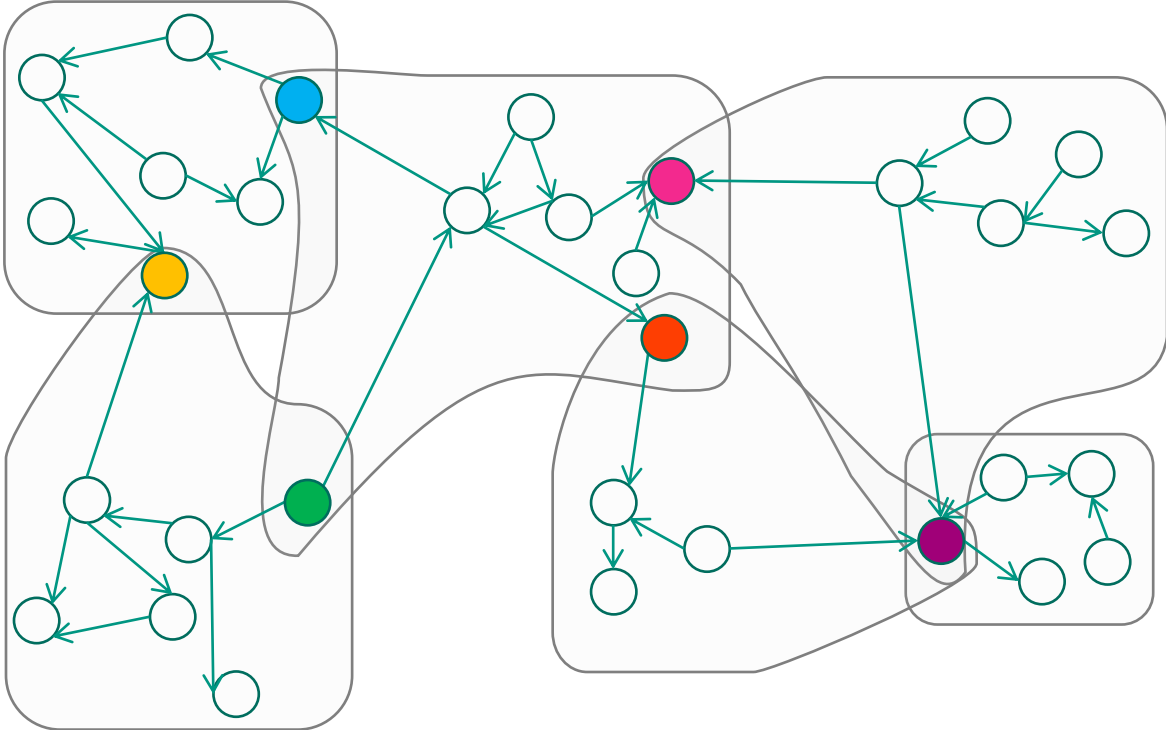
...Using URIs...



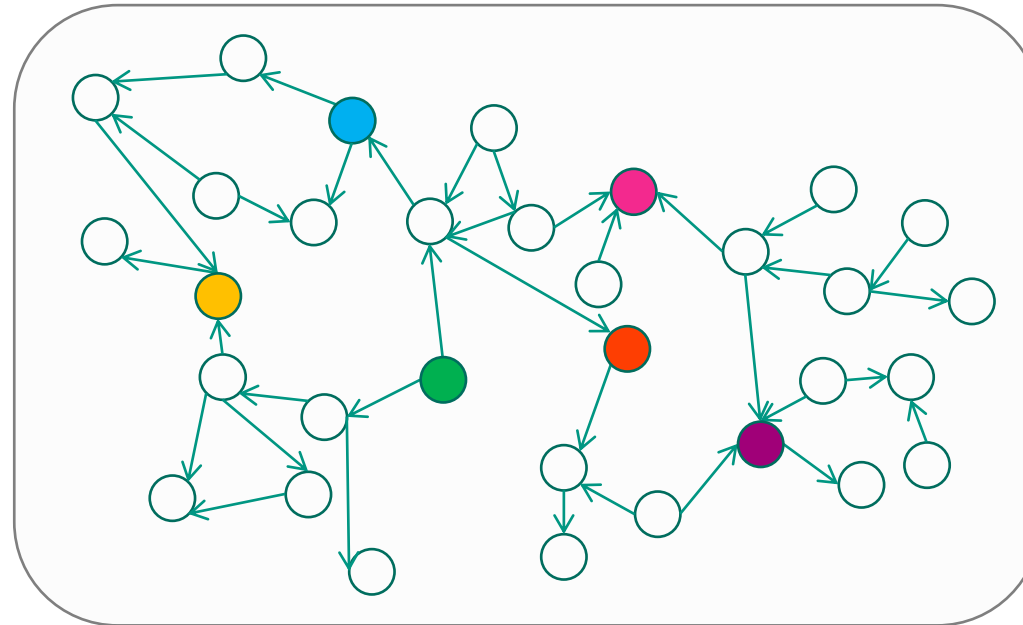
Circles with same color indicates identical resource

Documents and circles in the same color indicate correspondence between resource and information resource

...Can Actually Form...



...a Web of Data



- Each node is one resource, meaningfully linked to other resources, but acquired from different sources → Distributed Knowledge Graph

Creative Commons Licensing

- The slides have been prepared by Tobias Käfer, Andreas Harth, and Lars Heling
- This content is licensed under a Creative Commons Attribution 4.0 International license (CC BY 4.0):
<http://creativecommons.org/licenses/by/4.0/>



Desiderata for a Standardised Data Model for Data on the Web

- Low level of surprise (=low entropy) for machines¹ and those who program them
 - The higher the entropy, the more energy needed to process information (computing power, lines of code, memory, ...) ¹
- “Energy” could be put into:
 - Integrating data from different sources (merge operation, term disambiguation)
 - Writing processors
 - Validating processors
 - Running processors
- Contrast the “energy” required to process files with MIME types:
 - **text/uri-list**
 - **text/plain**
 - **text/html**
 - **application/xml**
 - **application/json**

¹ Cf. Mike Amundsen: “Autonomous Agents on the Web”, Keynote at the Workshop for Services and Applications over Linked Data, 2013
<https://www.slideshare.net/rnewton/autonomous-agents-on-the-web-22078931>

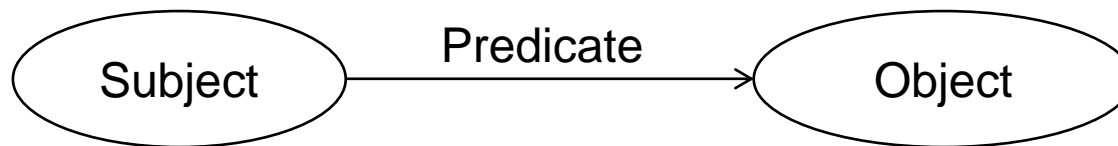
ENTER: THE RESOURCE DESCRIPTION FORMAT

Resource Description Framework (RDF)



1

- RDF is the foundational data format for both Semantic Web and Linked Data
- An RDF triple is the basic RDF concept describing information as a subject-property-object structure
- Property (or predicate) specifies relation between subject and object
- Triples can be viewed graphically:

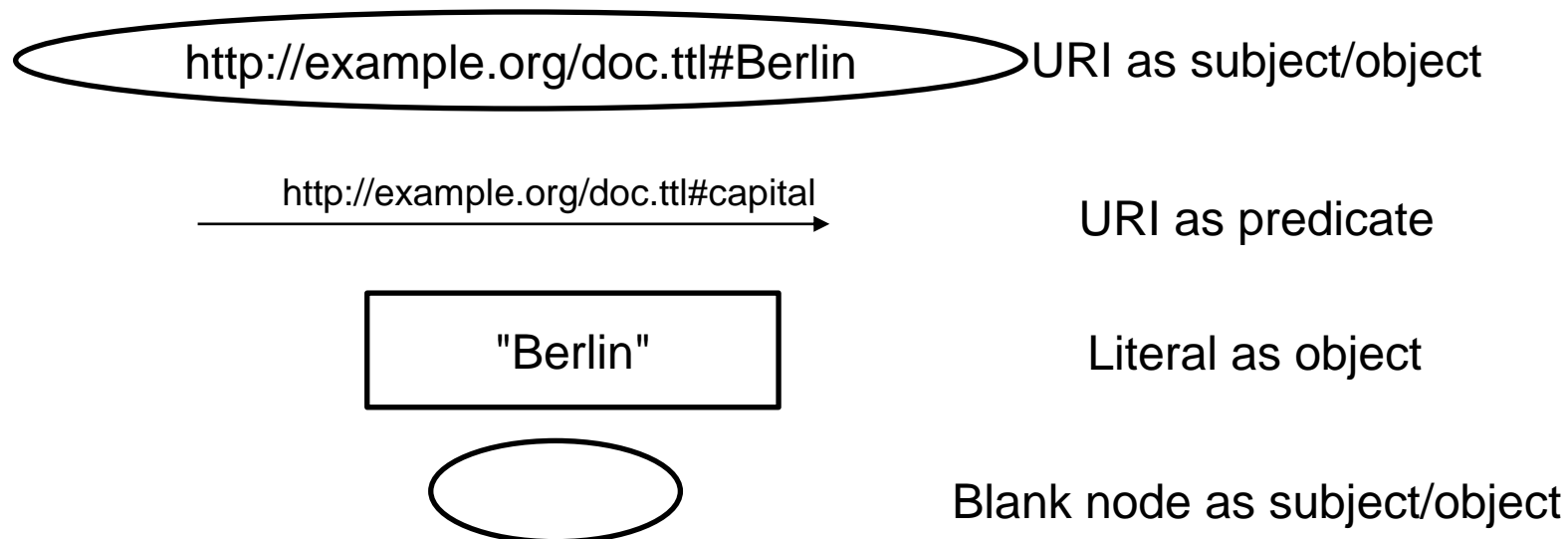


- RDF graphs can be presented as directed labelled graph

¹ <http://www.w3.org/RDF/icons/>

RDF Term Overview: URIs - Blank Nodes - Literals

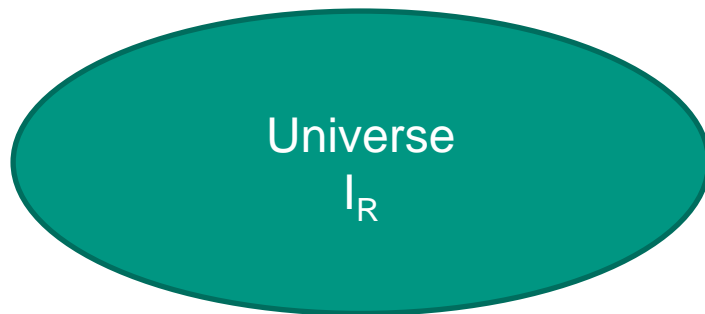
- **URIs** are used to globally identify resources
- **Blank nodes** refer to resources, too, but these resources can only be identified within a file and are not globally addressable (later more)
- **Literals** refer to concrete data values such as strings, integers, floats or dates. In RDF, we can use the datatypes defined as part of the XML Schema recommendation



RESOURCE DENOTATION USING RDF TERMS

Domain of Discourse

- Characterisation (Resource, Property, Universe): *A resource is a notion for things of discourse, be they abstract or concrete, physical or virtual. We write I_R for the set of resources, also called the universe or domain. We write I_P for the set of property resources.*



The set of all things we want to talk about



The set of resources for the URIs in predicate position

URIs – Uniform Resource Identifiers (Recap and Terminology)

- We use URIs as identifiers
- Full URIs
 - ...start with a scheme
 - Example: `http://example.org/doc.ttl#Berlin`
- CURIEs
 - Allow for abbreviating URIs
 - Example: with `doc:` being short for `http://example.org/doc.ttl#`, we can write `doc:Berlin` for `http://example.org/doc.ttl#Berlin`
- IRIs
 - Standard to allow for using characters outside of US-ASCII in URIs
 - We typically use “URI” and “IRI” interchangeably

`http://example.org/doc.ttl#Berlin` URI as subject/object

`http://example.org/doc.ttl#capital`

URI as predicate

<https://tools.ietf.org/rfc/rfc3986.txt>

URI Recap, Some Terminology and Practices

- Hierarchical URIs:
 - HTTP URIs are hierarchical in the path part of the URI
 - Example: `http://example.org/path/to/resource`
- Relative URIs
 - With hierarchical URIs you can have relative URIs that traverse the path
 - Example: `../../relative/../path/to/resource`
- Reference Resolution
 - Relative URIs can be converted to absolute URIs by resolving them
 - Example: resolving `../../relative/../path/to/resource` against `http://example.org/path/to/resource` yields the same URI
- Hash URIs
 - In Linked Data, we make the difference between a thing and the document about the thing. One way of expressing the difference is to use hash URIs
 - Example: `http://example.org/doc.ttl#Berlin`
- Slash URIs
 - Another way of making the difference is to use slash URIs for the thing and then use HTTP redirection to the document
 - Example: `http://dbpedia.org/resource/Berlin` redirects (303) to `http://dbpedia.org/data/Berlin.ttl` which in turn serves RDF

RDF Literals

■ Kinds of Literals:

- Simple Literals
- Language-tagged Literals
 - BCP47 language tags
- Typed literals

"Berlin"

Simple Literal

Literal as object

"Berlin"@de

Language-tagged Literal

- All literals have an (implicit) datatype → literals are pairs $\langle lex, dt \rangle$
 - For simple literals: `xsd:string`
 - For language-tagged literals: `rdf:langString` → triples $\langle lex, dt, lang \rangle$
- Eg. XML Schema datatypes

■ Lexical forms and value space

"1"^^xsd:integer

"01"^^xsd:integer

Two typed literals with different lexical forms denoting the same value

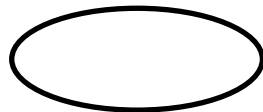
■ *Term Equality* of two Literals:

- Need to be equal, character by character:
 - Lexical forms
 - Datatype IRIs
 - Language tags (if any)

→ Two literals can have the same value without being the same RDF term.

Blank Nodes

- Blank nodes say that something [...] exists, without explicitly naming it
- Blank nodes *denote* resources
- Blank nodes do not *identify* resources
- Blank nodes do not have identifiers in the RDF abstractly speaking (see depiction below)
- In *implementations and serialisations*, blank nodes have identifiers (which are only locally scoped)



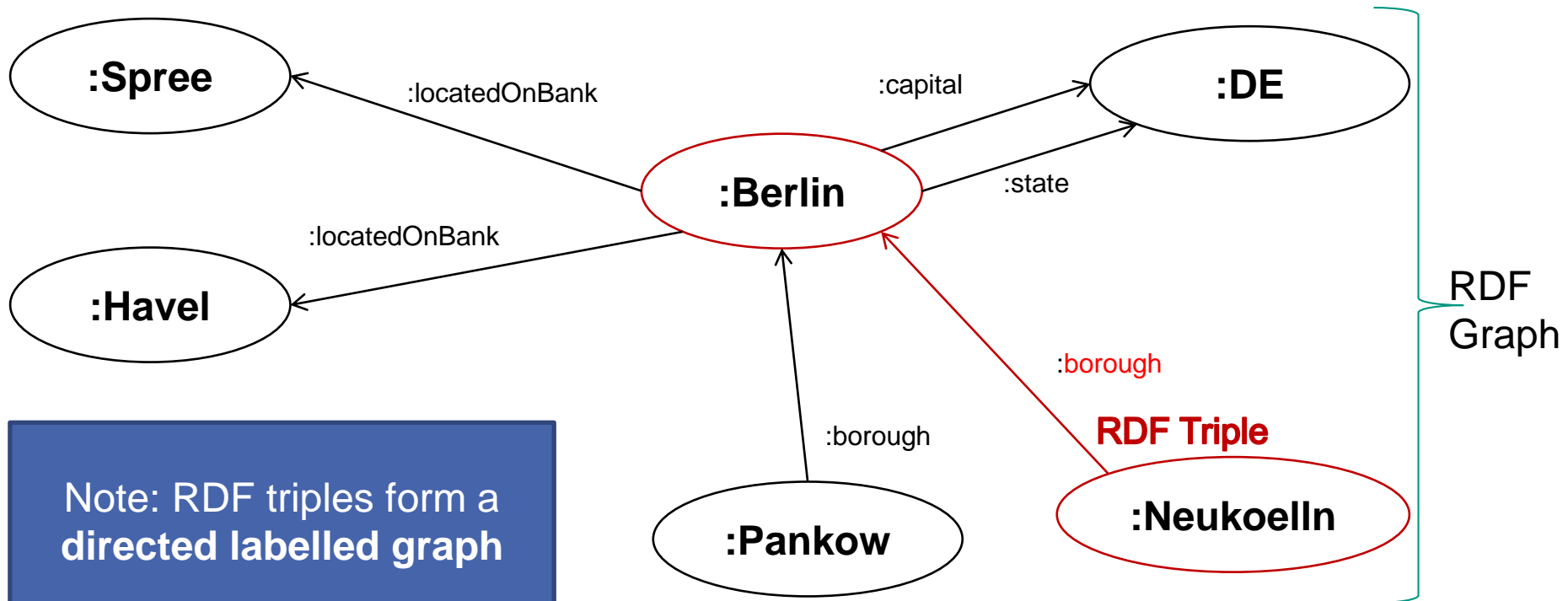
Blank node as subject/object

RESOURCE DESCRIPTIONS IN RDF GRAPHS

RDF – A Graph-based Data Model

- We arrange RDF Terms in RDF Triples → the edges in RDF Graphs

Definition (RDF Triple, RDF Graph). Let \mathcal{U} be the set of URIs, \mathcal{B} the set of blank nodes, and \mathcal{L} the set of RDF literals. A tuple $\langle s, p, o \rangle \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called an RDF triple, where s is the subject, p is the predicate and o is the object. A set of RDF triples is called RDF graph.

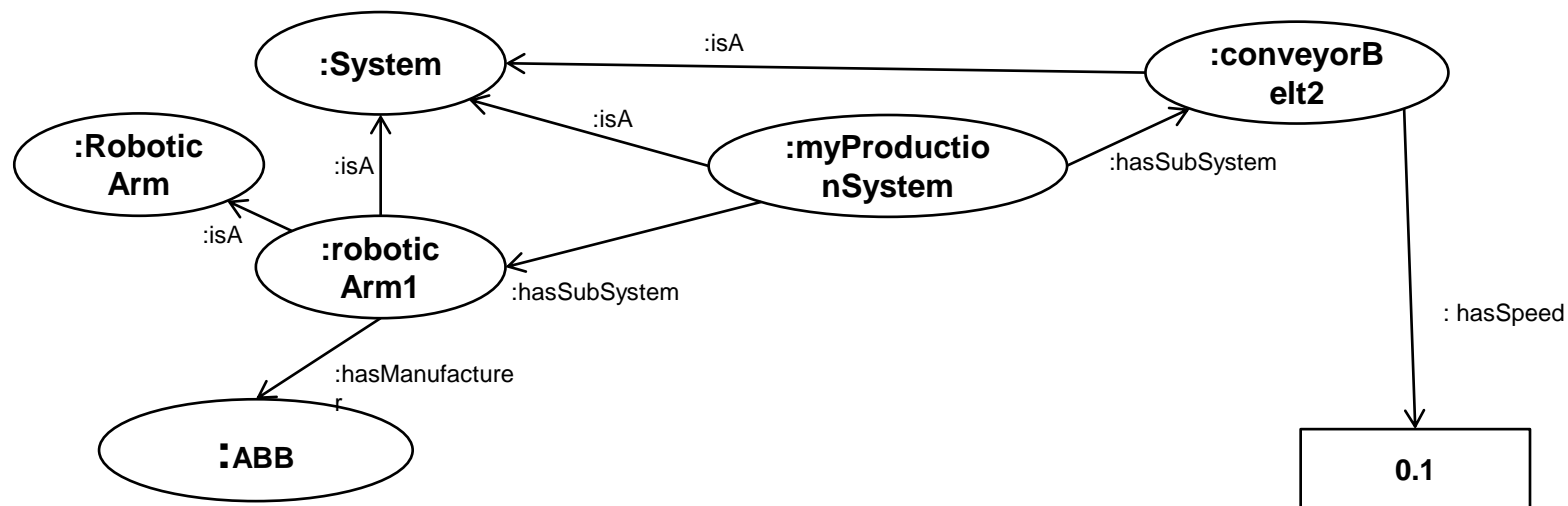


Note: RDF triples form a directed labelled graph

Instead of `http://example.org/doc.ttl#` we write just write ":"

Remember...

Let's use ":" as abbreviation for "http://example.org/doc.ttl#"

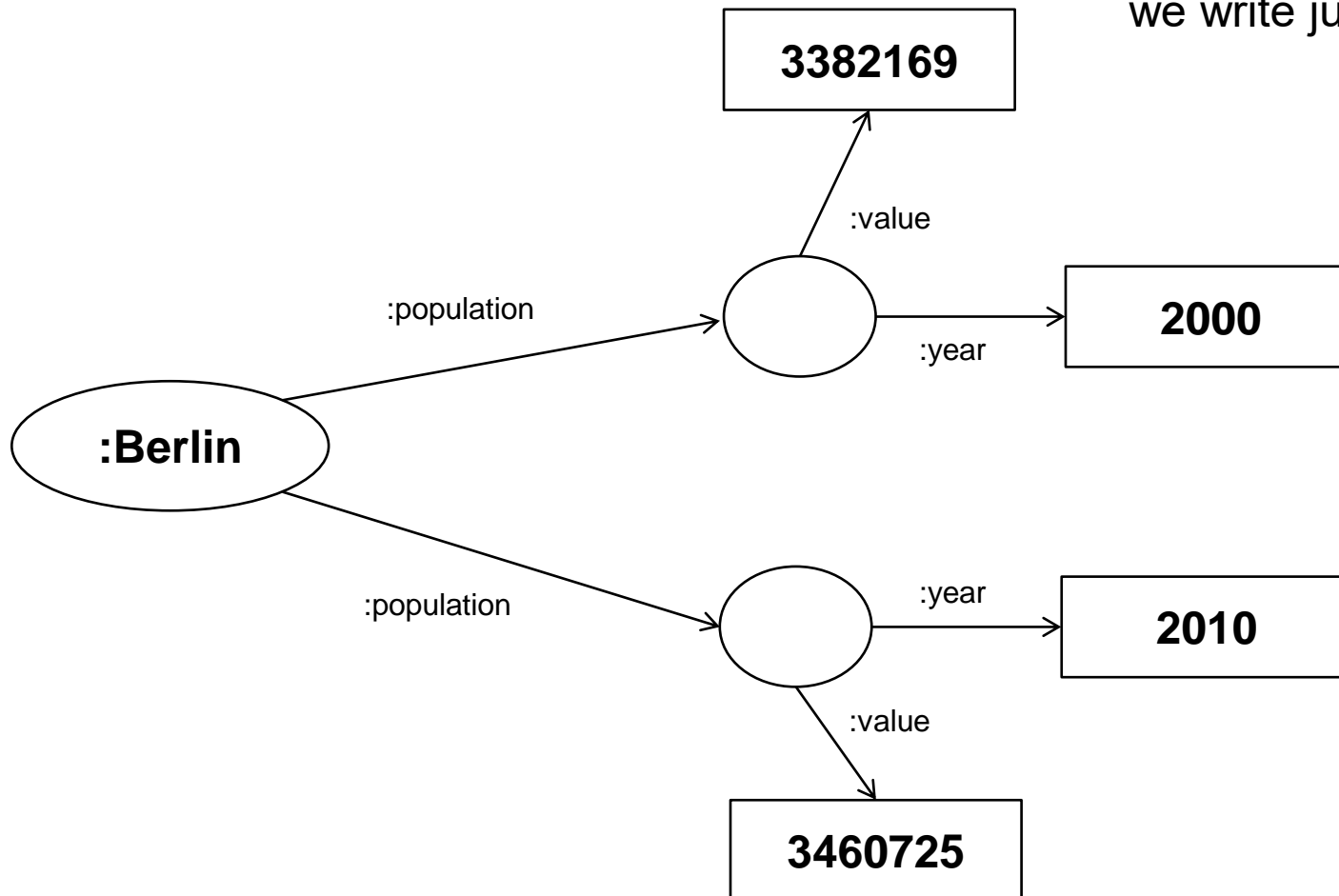


N-ary Relations

- An RDF property represents a binary relation between resources
- But there are cases where we want to model relations between more than two resources
- So-called n-ary relations can be modelled as binary relations, if we think of the relation itself as a resource
- Often, we use blank nodes to identify the relation (as resource)

Example: N-ary Relation in RDF

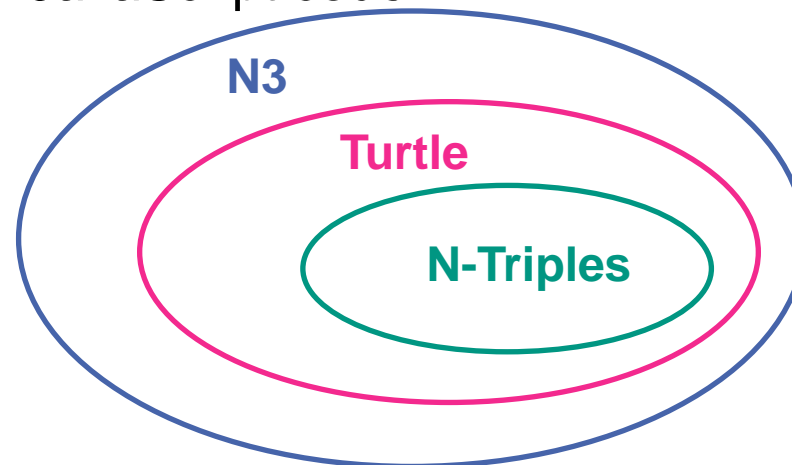
Instead of
<http://example.org/doc.ttl#>
we write just write “:”



RDF SERIALISATION FORMATS

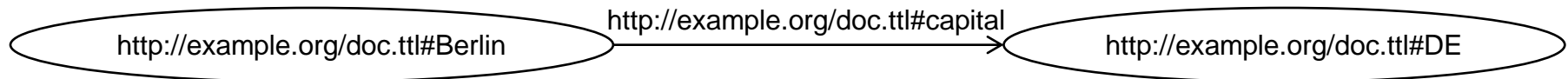
RDF Syntaxes

- Based on Notation3 (an expressive textual syntax for a superset of the RDF data model)
 - **N-Triples** is a very simple syntax, in which one triple is written in one line
 - **Turtle** adds syntactical features to N-Triples that increase human readability and writeability (eg. CURIEs, abbreviations)
- **RDF/XML** is a XML-based syntax with high historical relevance and practical prevalence
- **JSON-LD** is a JSON-based syntax, which allows for eased interoperability of RDF with JavaScript code



N-Triples – A Simple Syntax for RDF Triples

- N-Triples provides a very straight-forward way to write down RDF triples



- The basic structure consists of subject property object triples, followed by a dot and a newline
- URIs are enclosed in angle brackets (“<>”). No relative URIs!
- Blank nodes are prefixed with an underscore and a colon (“_:”)
- Literals are enclosed in quotation marks (“”)
- Comments are marked with a hash character (“#”)

```
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#capital> <http://example.org/doc.ttl#DE> .
```

```
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#label> "Berlin" .
```

```
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#population> _:bn . # part of n-ary rel.
```

- The s p o . representation is also called simple triple form

<https://www.w3.org/TR/n-triples/>

Simple, Datatyped, and Language-tagged Literals in N-Triples

- Simple, Datatyped and Language-tagged Literals enclose the lexical form in quotation marks (“”””)
- Datatyped literals use two caret characters (“^^”) to specify the datatype URI
- For example, a literal denoting the integer value 3460725 is written as

```
"3460725"^^<http://www.w3.org/2001/XMLSchema#integer>
```

- Language-tagged literals use the at character (“@”) to specify the language
- For example, the literal denoting *Berlin* in German is written as

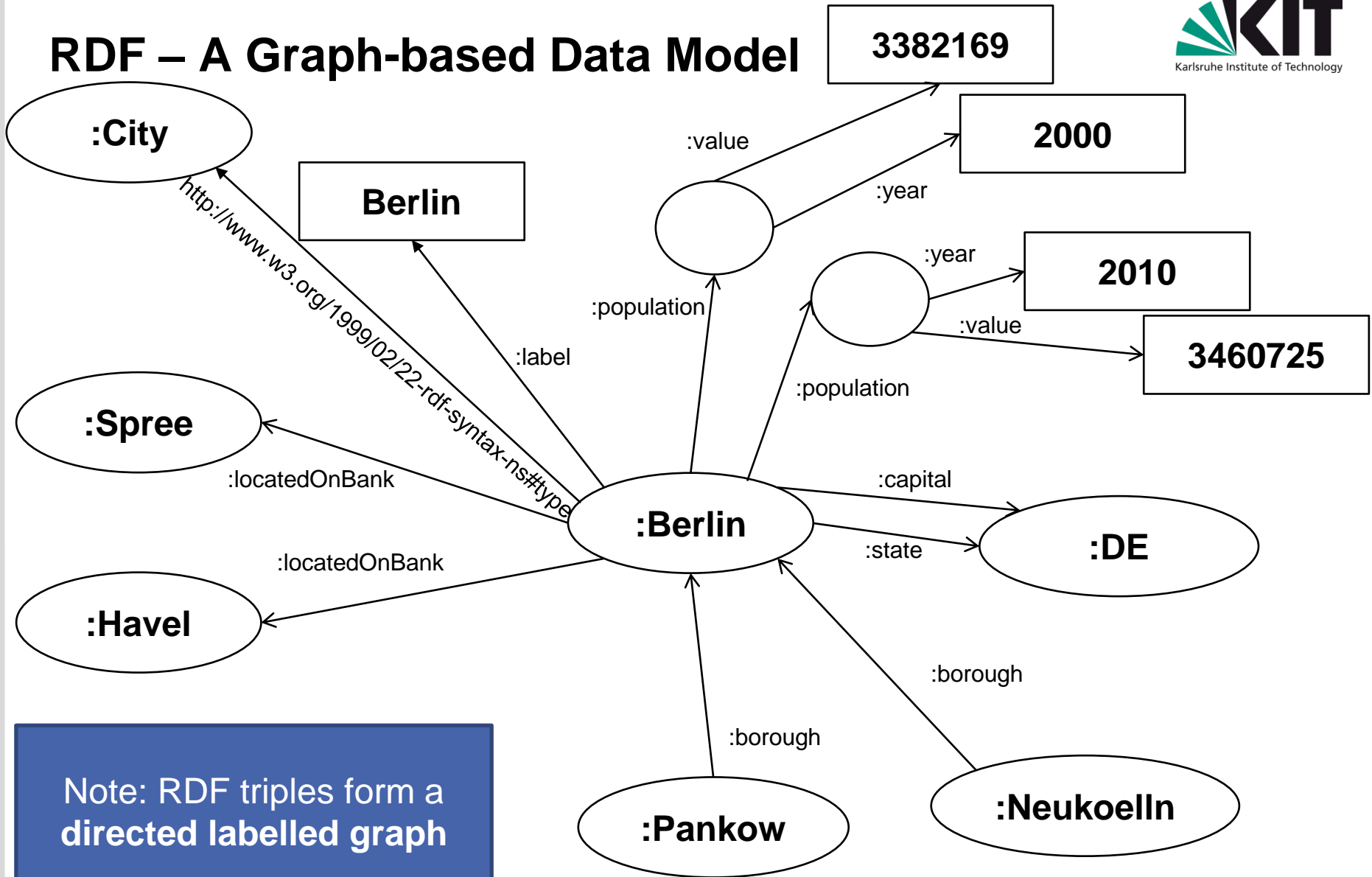
```
"Berlin"@de
```

Turtle

- Turtle – the “Terse RDF Triple Language”
- Easier for a human to read and write than N-Triples
- Will be used throughout the lecture
- Supports:
 1. CURIEs
 2. Relative URIs
 3. Abbreviation for `rdf:type` (“a”)
 4. Abbreviations for literals with some XML Schema datatypes
 5. Abbreviations for repetition of subject (“;”) and subject+predicate (“,”)
 6. Abbreviations for RDF lists (later)

- We now derive Turtle starting with N-Triples considering above points except 2 and 6

RDF – A Graph-based Data Model



Note: RDF triples form a directed labelled graph

Instead of `http://example.org/doc.ttl#` we write just write ":"

N-Triples

```

<http://example.org/doc.ttl#Berlin> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://example.org/doc.ttl#City> .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#capital> <http://example.org/doc.ttl#DE> .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#state> <http://example.org/doc.ttl#DE> .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#locatedOnBank> <http://example.org/doc.ttl#Spree> .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#locatedOnBank> <http://example.org/doc.ttl#Havel> .
<http://example.org/doc.ttl#Pankow> <http://example.org/doc.ttl#borough> <http://example.org/doc.ttl#Berlin> .
<http://example.org/doc.ttl#Neukoelln> <http://example.org/doc.ttl#borough> <http://example.org/doc.ttl#Berlin> .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#label> "Berlin"@de .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#population> _:genid1 .
<http://example.org/doc.ttl#Berlin> <http://example.org/doc.ttl#population> _:genid2 .
_:genid1 <http://example.org/doc.ttl#value> "3382169"^^<http://www.w3.org/2001/XMLSchema#integer> .
_:genid1 <http://example.org/doc.ttl#year> "2000"^^<http://www.w3.org/2001/XMLSchema#integer> .
_:genid2 <http://example.org/doc.ttl#value> "3460725"^^<http://www.w3.org/2001/XMLSchema#integer> .
_:genid2 <http://example.org/doc.ttl#year> "2010"^^<http://www.w3.org/2001/XMLSchema#integer> .
  
```

+ CURIEs

- You can allow for CURIEs by issuing @prefix directives of the form:
 - @prefix *prefix-label*: <associated URI> .

```
@prefix : <http://example.org/doc.ttl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
:Berlin rdf:type :City .
:Berlin :capital :DE .
:Berlin :state :DE .
:Berlin :locatedOnBank :Spree .
:Berlin :locatedOnBank :Havel .
:Pankow :borough :Berlin .
:Neukoelln :borough :Berlin .
:Berlin :label "Berlin"@de .
:Berlin :population _:genid1 .
:Berlin :population _:genid2 .
_:genid1 :value "3382169"^^xsd:integer .
_:genid1 :year "2000"^^xsd:integer .
_:genid2 :value "3460725"^^xsd:integer .
_:genid2 :year "2010"^^xsd:integer .
```

+ Abbreviation for rdf:type

- You can abbreviate `rdf:type` with “a”

```
@prefix : <http://example.org/doc.ttl#> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .  
  
:Berlin a :City .  
:Berlin :capital :DE .  
:Berlin :state :DE .  
:Berlin :locatedOnBank :Spree .  
:Berlin :locatedOnBank :Havel .  
:Pankow :borough :Berlin .  
:Neukoelln :borough :Berlin .  
:Berlin :label "Berlin"@de .  
:Berlin :population _:genid1 .  
:Berlin :population _:genid2 .  
_:genid1 :value "3382169"^^xsd:integer .  
_:genid1 :year "2000"^^xsd:integer .  
_:genid2 :value "3460725"^^xsd:integer .  
_:genid2 :year "2010"^^xsd:integer .
```

+ Abbreviations for Some XML Schema Datatypes

- You can use shorthands for numbers typed with `xsd:integer` ,
`xsd:decimal` (with “.”), and `xsd:float` (written in scientific notation)

@prefix : <http://example.org/doc.ttl#> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .

```
:Berlin a :City .
:Berlin :capital :DE .
:Berlin :state :DE .
:Berlin :locatedOnBank :Spree .
:Berlin :locatedOnBank :Havel .
:Pankow :borough :Berlin .
:Neukoelln :borough :Berlin .
:Berlin :label "Berlin"@de .
:Berlin :population _:genid1 .
:Berlin :population _:genid2 .
_:genid1 :value 3382169 .
_:genid1 :year 2000 .
_:genid2 :value 3460725 .
_:genid2 :year 2010 .
```

+ Abbreviations for Repetitions of Subject+Predicate

- Use the colon “,” in consecutive triples to repeat subject and predicate
- Order the triples wisely to benefit; indentation helps for the overview

```
@prefix : <http://example.org/doc.ttl#> .
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .
```

```
:Berlin a :City .  
:Berlin :capital :DE .  
:Berlin :state :DE .  
:Berlin :locatedOnBank :Spree , :Havel .  
:Pankow :borough :Berlin .  
:Neukoelln :borough :Berlin .  
:Berlin :label "Berlin"@de .  
:Berlin :population _:genid1 , _:genid2 .  
_:genid1 :value 3382169 .  
_:genid1 :year 2000 .  
_:genid2 :value 3460725 .  
_:genid2 :year 2010 .
```

+ Abbreviations for Repetitions of Subject

- Use the semicolon “;” in consecutive triples to repeat the subject
- Order the triples wisely to benefit; indentation helps for the overview

```
@prefix : <http://example.org/doc.ttl#> .
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .
```

```
:Berlin a :City ;  
  :capital :DE ;  
  :state :DE ;  
  :locatedOnBank :Spree , :Havel .  
:Pankow :borough :Berlin .  
:Neukoelln :borough :Berlin .  
:Berlin :label "Berlin"@de ;  
  :population _:genid1 , _:genid2 .  
_:genid1 :value 3382169 ;  
  :year 2000 .  
_:genid2 :value 3460725 ;  
  :year 2010 .
```

Are the triples in
a smart order?

+ Abbreviations for Blank Nodes

- Use brackets “[]” to abbreviate blank nodes
- Note that you need to group *all* mentions of the former blank node ID

```
@prefix : <http://example.org/doc.ttl#> .
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .
```

```
:Berlin a :City ;  
  :capital :DE ;  
  :state :DE ;  
  :locatedOnBank :Spree , :Havel .  
:Pankow :borough :Berlin .  
:Neukoelln :borough :Berlin .  
:Berlin :label "Berlin"@de ;  
  :population [ :value 3382169 ; :year 2000 ] ,  
             [ :value 3460725 ; :year 2010 ] .
```

Now You Know Turtle

- Language features not covered in the previous slides:
 - Relative URIs (see slides on URIs)
 - RDF list syntax (see later)

```
@prefix : <http://example.org/doc.ttl#> .
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#integer> .
```

```
:Berlin a :City ;  
  :capital :DE ;  
  :state :DE ;  
  :locatedOnBank :Spree , :Havel .  
:Pankow :borough :Berlin .  
:Neukoelln :borough :Berlin .  
:Berlin :label "Berlin"@de ;  
  :population [ :value 3382169 ; :year 2000 ] ,  
             [ :value 3460725 ; :year 2010 ] .
```


Turtle Exercise

- Describe the facts from →
as RDF Triples in Turtle Syntax
- Invent URIs as necessary
- Type your questions into the chat
and raise your hand once you're
done
- Steps:
 1. Open a text editor, eg. Nano
`nano production.nt`
 2. Don't use abbreviations, ie. write N-
Triples
 3. Double-check your solution using
`raper` or `RDFShape` [1]
`raper -i ntriples production.nt`
 4. Copy the file to a new file
`cp production.nt production.ttl`
 5. Edit the new file and apply as many
Turtle abbreviations as possible
 6. Double-check your solution using
`raper` or `RDFShape` [1]
`raper -i turtle production.ttl`
- `myProductionSystem isA System`
- `myProductionSystem hasSubSystem
roboticArm1`
- `myProductionSystem hasSubSystem
conveyorBelt2`
- `roboticArm1 isA System`
- `roboticArm1 isA RoboticArm`
- `roboticArm1 hasManufacturer ABB`
- `conveyorBelt2 isA System`
- `conveyorBelt2 hasSpeed 0.1`

[1] <http://rdfshape.herokuapp.com/dataConversions>

Sample Solution – Step 2

```
<http://example.org/#myProductionSystem> <http://example.org/#isA> <http://example.org/#System> .  
<http://example.org/#myProductionSystem> <http://example.org/#hasSubSystem> <http://example.org/#roboticArm1> .  
<http://example.org/#myProductionSystem> <http://example.org/#hasSubSystem> <http://example.org/#conveyorBelt2> .  
<http://example.org/#roboticArm1> <http://example.org/#isA> <http://example.org/#System> .  
<http://example.org/#roboticArm1> <http://example.org/#isA> <http://example.org/#RoboticArm> .  
<http://example.org/#roboticArm1> <http://example.org/#hasManufacturer> <http://example.org/#ABB> .  
<http://example.org/#conveyorBelt2> <http://example.org/#isA> <http://example.org/#System> .  
<http://example.org/#conveyorBelt2> <http://example.org/#hasSpeed> "0.1" .
```

Sample Solution – Step 5

```
@prefix : <http://example.org/#> .
:myProductionSystem a :System ; # <- let's pretend on the previous slide, we had rdf:type
  :hasSubSystem :roboticArm1 , :conveyorBelt2 .
:roboticArm1 a :System , :RoboticArm ; # <- let's pretend on the previous slide, we had rdf:type
  :hasManufacturer :ABB .
:conveyorBelt2 a :System ;
  :hasSpeed "0.1" .
# :hasSpeed 0.1 . <- In the previous slide, we had a string so we can't write this line or the two following:
# @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
# :hasSpeed "0.1"^^xsd:decimal .
```

RDF/XML Example

```

<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns="http://example.org/doc.ttl#">
  <rdf:Description rdf:about="http://example.org/doc.ttl#Berlin">
    <rdf:type rdf:resource="http://example.org/doc.ttl#City"/>
    <capital rdf:resource="http://example.org/doc.ttl#DE"/>
    <state rdf:resource="http://example.org/doc.ttl#DE"/>
    <locatedOnBank rdf:resource="http://example.org/doc.ttl#Spree"/>
    <locatedOnBank rdf:resource="http://example.org/doc.ttl#Havel"/>
    <borough rdf:resource="http://example.org/doc.ttl#Berlin"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/doc.ttl#Pankow">
    <borough rdf:resource="http://example.org/doc.ttl#Berlin"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/doc.ttl#Neukoelln">
    <borough rdf:resource="http://example.org/doc.ttl#Berlin"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/doc.ttl#Berlin">
    <label xml:lang="de">Berlin</label>
    <population rdf:nodeID="genid1"/>
    <population rdf:nodeID="genid2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="genid1">
    <value rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">3382169</value>
    <year rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">2000</year>
  </rdf:Description>
  <rdf:Description rdf:nodeID="genid2">
    <value rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">3460725</value>
    <year rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">2010</year>
  </rdf:Description>
</rdf:RDF>

```

JSON-LD Example

```

{
  "@context": {
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "doc": "http://example.org/doc.ttl#",
    "people": "doc:population",
    "country": { "@id": "doc:state", "@type": "@id" },
    "borough": { "@reverse": "doc:borough", "@type": "@id" },
    "locatedOnBank": { "@id": "doc:locatedOnBank", "@type": "@id" }
  },
  "@graph": [
    {
      "@id": "doc:Berlin",
      "http://example.org/doc.ttl#label": { "@value": "Berlin", "@language": "de" },
      "http://example.org/doc.ttl#capital": { "@id": "doc:DE" },
      "rdf:type": { "@id": "doc:City" },
      "locatedOnBank": [ "doc:Havel", "doc:Spree" ],
      "people": [
        { "doc:value": 3382169, "doc:year": 2000 },
        { "doc:value": 3460725, "doc:year": 2010 }
      ],
      "country": "doc:DE",
      "borough": [ "doc:Neukoelln", "doc:Pankow" ]
    }
  ]
}

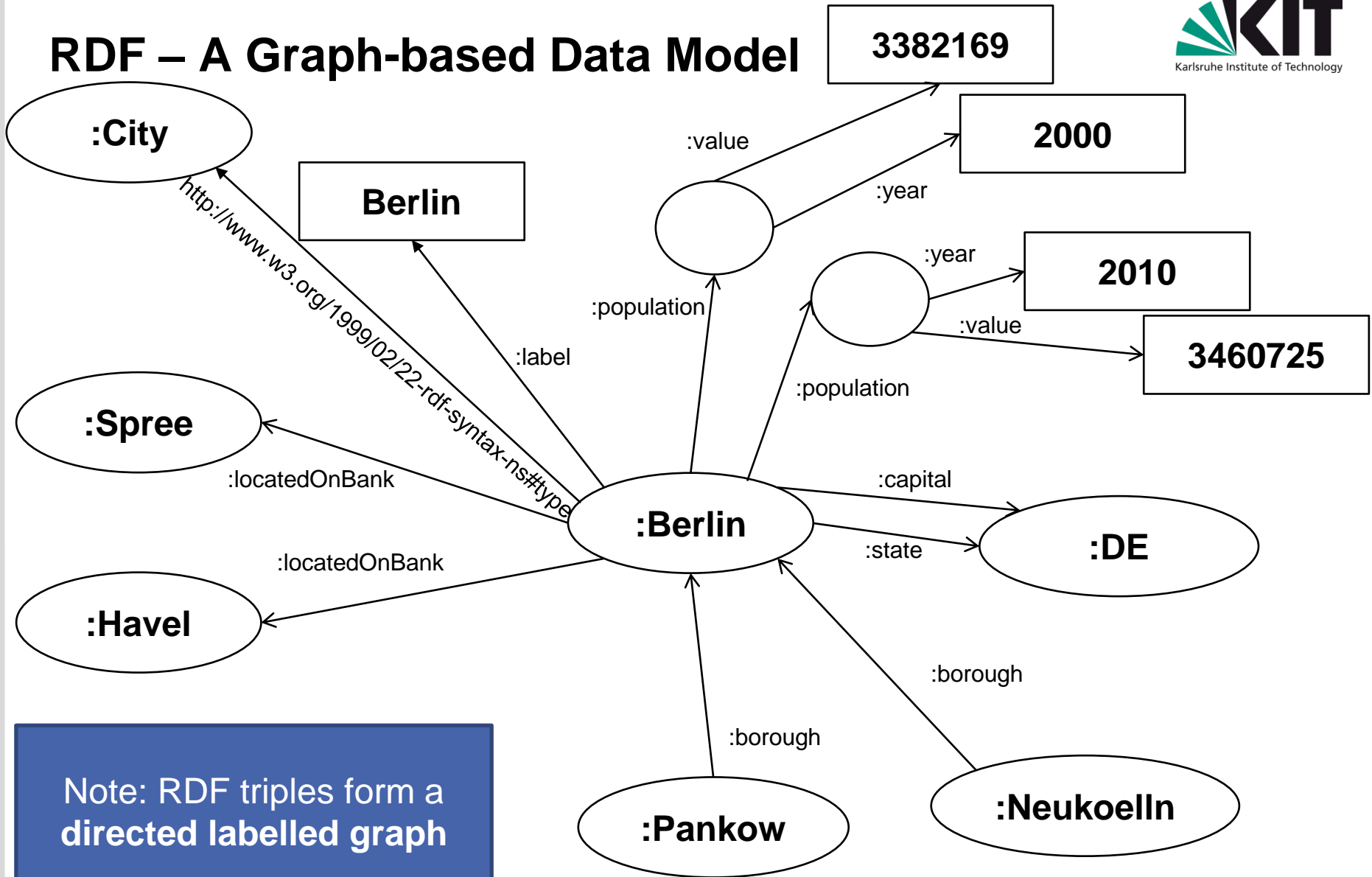
```

Exercise

- Open a text editor and turn the graph that you have drawn into Turtle.
 - Double-check your syntax using <http://rdfshape.herokuapp.com/dataConversions>
 - (make sure if you copy-and-paste that URIs like file://app get turned back into what they were before)
- Put your Turtle file on the web
 - Create an account at a SoLiD POD provider, e.g. <http://solidcommunity.net/>
 - Go to your public folder
 - Create a new file and paste your Turtle
 - Verify the results, by viewing the URI in your browser or using curl

WORKING WITH MULTIPLE RDF GRAPHS

RDF – A Graph-based Data Model

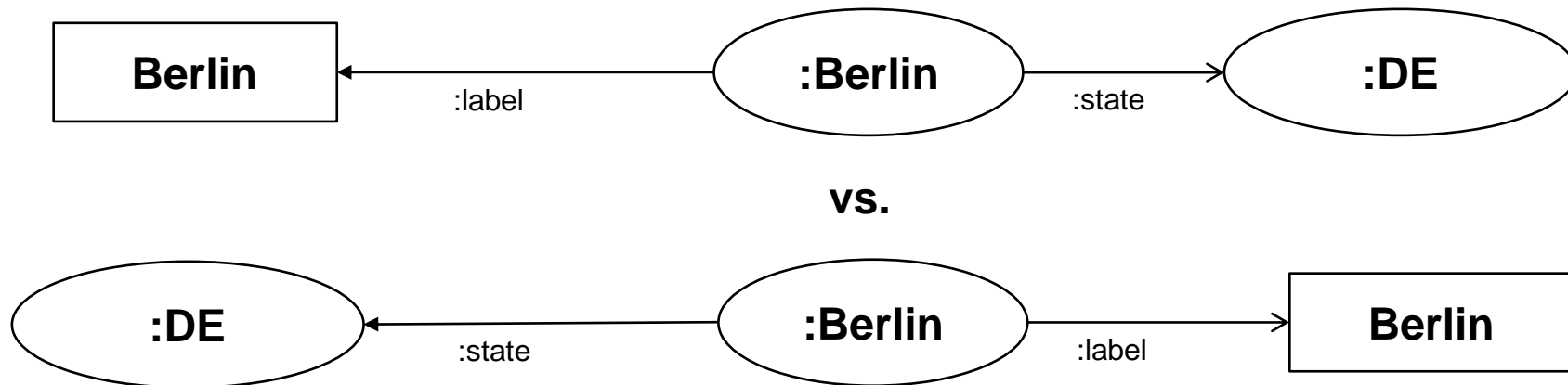


Note: RDF triples form a directed labelled graph

Instead of `http://example.org/doc.ttl#` we write just write ":"

Isomorphism As Equivalence Relation

- We employ isomorphism to check whether two RDF mean the same
- Are those two graphs the same?



```

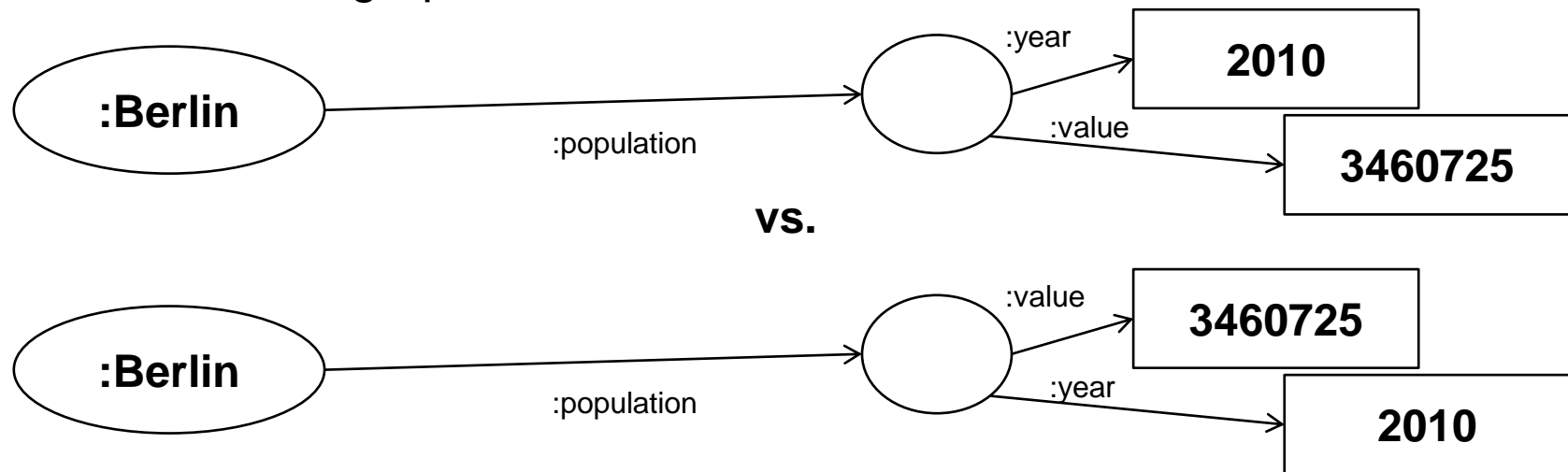
@prefix : <http://example.org/doc.ttl#> .
:Berlin :state :DE .
:Berlin :label "Berlin"@de .
  
```

```

@prefix : <http://example.org/doc.ttl#> .
:Berlin :label "Berlin"@de .
:Berlin :state :DE .
  
```

Isomorphism As Equivalence Relation

- We employ isomorphism to check whether two RDF mean the same
- Are those two graphs the same?



```

@prefix : <http://example.org/doc.ttl#> .
:Berlin :population _:bn1 .
_:bn1 :year 2010 .
_:bn1 :value 3460725 .
  
```

```

@prefix : <http://example.org/doc.ttl#> .
_:genid1 :value 3460725 .
:Berlin :population _:genid1 .
_:genid1 :year 2010 .
  
```

Isomorphism As Equivalence Relation

- Two RDF graphs are isomorphic if there is a bijection M between the two sets of nodes in the graphs G and G' such that:
 - M maps blank nodes to blank nodes.
 - $M(lit) = lit$ for all RDF literals lit which are nodes of G .
 - $M(iri) = iri$ for all IRIs iri which are nodes of G .
 - The triple (s, p, o) is in G if and only if the triple $(M(s), p, M(o))$ is in G'

```
@prefix : <http://example.org/doc.ttl#> .
:Berlin :population _:bn1 .
_:bn1 :year 2010 .
_:bn1 :value 3460725 .
```

VS.

```
@prefix : <http://example.org/doc.ttl#> .
_:genid1 :value 3460725 .
:Berlin :population _:genid1 .
_:genid1 :year 2010 .
```

Nodes:

URIs:

:Berlin

Literals:

"2010"^^xsd:integer

"3460725"^^xsd:integer

Blank Nodes:

_:bn1



Nodes:

URIs:

:Berlin

Literals:

"3460725"^^xsd:integer

"2010"^^xsd:integer

Blank Nodes:


_:genid1

<https://www.w3.org/TR/rdf11-concepts/#graph-isomorphism>

RDF Dataset

Considering the following prefix declarations:

```
@prefix dbr: <http://dbpedia.org/resource/> .  
@prefix dbp: <http://dbpedia.org/property/> .  
@prefix d: <http://example.org/doc.ttl#> .
```

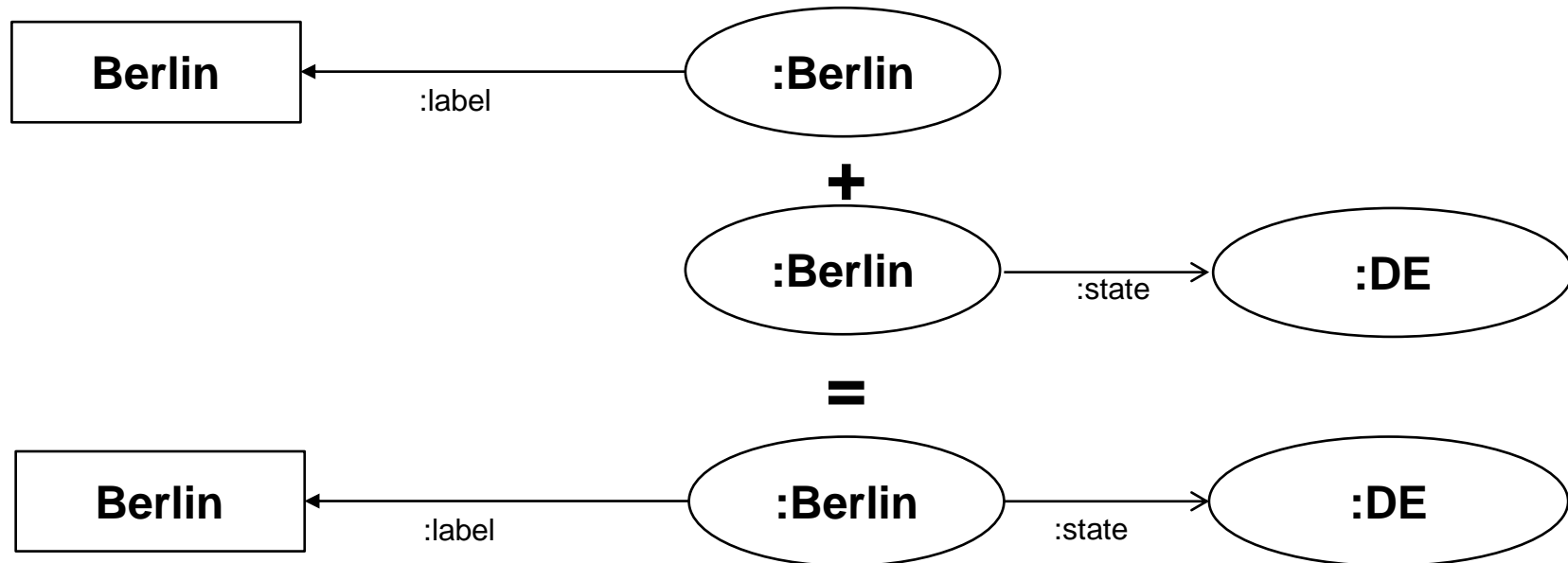


- To talk about a collection of RDF graphs, we use the RDF Dataset.
- In an RDF dataset, graphs can be identified using a name
 - The name can be a URI or a blank node
 - There can be one graph without a name, the default graph
 - The name does not need to have any meaning for the graph
- Definition (Named Graph, RDF Dataset): *Let \mathcal{G} be the set of RDF graphs and \mathcal{U} be the set of URIs. A pair $\langle g, u \rangle \in \mathcal{G} \times \mathcal{U}$ is called a named graph. An RDF dataset consists of a (possibly empty) set of named graphs (with distinct names) and a default graph $g \in \mathcal{G}$ without a name.*
- Example:

Name	Graph
<code><http://example.org/doc.ttl></code>	<code>d:Berlin d:state d:DE . d:Berlin d:label "Berlin"@de .</code>
<code><http://dbpedia.org/data/Berlin.ttl></code>	<code>dbr:Berlin dbo:areaCode 030 . dbr:Berlin dbo:kfz "B" .</code>
	<code>d:Berlin owl:sameAs dbr:Berlin .</code>

Combining 2 RDF Graphs: Union

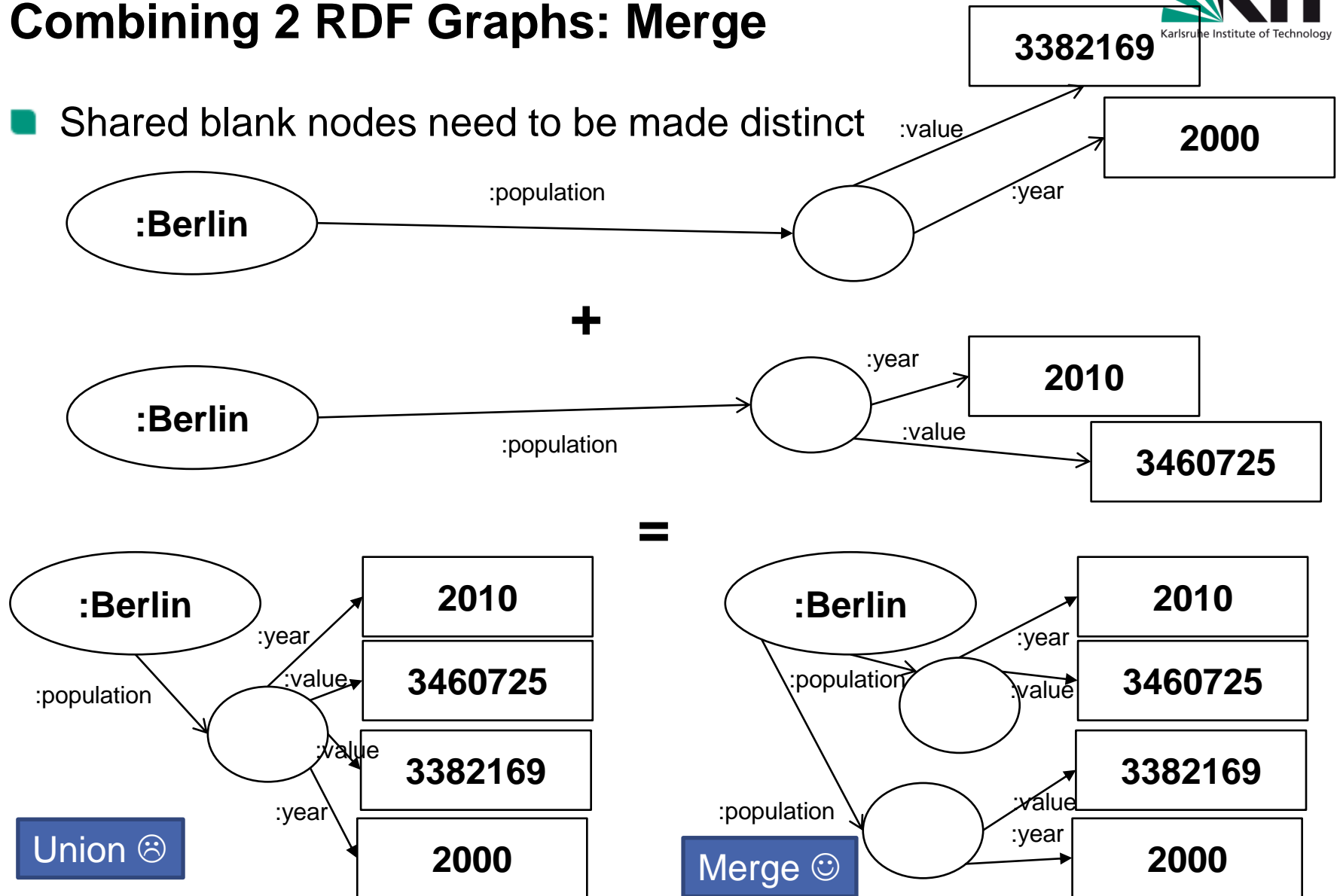
- No blank nodes in the RDF graphs → RDF graph combination trivial



- Simply take the union of the RDF triples in the RDF graphs

Combining 2 RDF Graphs: Merge

- Shared blank nodes need to be made distinct



RDF Merge Example in Triples

- Consider the following RDF graphs:

- G:

```
@prefix : <http://example.org/doc.ttl#>.
:Berlin :population _:pop .
_:pop :value 3382169 ; :year 2000 .
```

- E:

```
@prefix : <http://example.org/doc.ttl#>.
:Berlin :population _:pop .
_:pop :value 3460725 ; :year 2010 .
```

- Incorrect merge of G and E:

- G1:

```
@prefix : <http://example.org/doc.ttl#>.
:Berlin :population _:pop .
_:pop :value 3382169 ; :year 2000 .
_:pop :value 3460725 ; :year 2010 .
```

- Correct merges of G and E:

- G2:

```
@prefix : <http://example.org/doc.ttl#>.
:Berlin :population _:pop1, _:pop2 .
_:pop1 :value 3382169 ; :year 2000 .
_:pop2 :value 3460725 ; :year 2010 .
```

- G3:

```
@prefix : <http://example.org/doc.ttl#>.
:Berlin :population _:pop1, _:pop2 .
_:pop2 :value 3382169 ; :year 2000 .
_:pop1 :value 3460725 ; :year 2010 .
```

CC - Creative Commons Licensing

- The slides were prepared by Tobias Käfer, Andreas Harth, and Lars Helling
- **This content is licensed under a Creative Commons Attribution 4.0 International license (CC BY 4.0):**
<http://creativecommons.org/licenses/by/4.0/>



CC - Creative Commons Licensing

- The slides were compiled by Tobias Käfer, prepared by Lars Heling based on Andreas Harth's input, with major modifications by Maribel Acosta

- **This content is licensed under a Creative Commons Attribution 4.0 International license (CC BY 4.0):**
<http://creativecommons.org/licenses/by/4.0/>



Agenda

- 1. Introduction**
2. Structure of SPARQL Queries
3. Basic Graph Patterns
4. Querying Multiple (Named) RDF Graphs

Example Question

“What are the boroughs of Berlin?”

- How can we answer this question over RDF data?

Retrieving Data from a Dataset

- How to retrieve data from a dataset?
 - Queries are used in order to retrieve *relevant* data from a dataset
- Relational databases:
 - A set of tuples is stored in a table (Relation)
 - **Structured Query Language (SQL)**

Relation: Cities

Name	Population	BoroughOf
Oststadt	21 091	Karlsruhe
Pankow	384 367	Berlin
...

```
SELECT Name
FROM Cities
WHERE BoroughOf = "Berlin" ;
```

- Graph databases:
 - What is a dataset in RDF?
 - How can we query data represented in RDF?

RDF Datasets

- A collection of graphs is called an RDF dataset.
- An RDF dataset has one default graph without a name,
and
- zero or more graphs with a name (a URI)

SPARQL

- Acronym:
 - **SPARQL Protocol And RDF Query Language**
- Specified by W3C
 - Current version: SPARQL 1.1¹ (March 2013)
- There are eleven SPARQL Recommendations, covering:
 - Syntax and semantics of queries over RDF
 - Protocol to pose queries against a SPARQL endpoint and to retrieve results
 - Various serialisations of query results
 - Entailment regimes
 - Update language
 - Federated query
 - ...

¹ <http://www.w3.org/TR/sparql11-overview/>

Agenda

1. Introduction
- 2. Structure of SPARQL Queries**
3. Basic Graph Patterns
4. Querying Multiple (Named) RDF Graphs

Back to Our Question

“What are the boroughs of Berlin?”



```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough  
FROM <http://example.org/cities.ttl>  
WHERE {  
    (Some conditions)  
}
```

Components of SPARQL Queries (1)

PREFIX ex: `<http://example.org/cities.ttl#>`

```
SELECT ?borough
FROM <http://example.org/cities.ttl>
WHERE {
    (Some conditions)
}
```

Prefix definitions:

- PREFIX keyword to introduce CURIEs
- Subtly different from Turtle syntax
 - The final period is not used
 - No “@” at the beginning

Components of SPARQL Queries (2)

```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough  
FROM <http://example.org/cities.ttl>  
WHERE {  
    (Some conditions)  
}
```

Query form:

- ASK, SELECT, DESCRIBE, or CONSTRUCT
- Details in a bit...

Components of SPARQL Queries (3)

```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough  
FROM <http://example.org/cities.ttl>  
WHERE {  
    (Some conditions)  
}
```

Variable projection:

- Variables are “placeholders” for RDF terms
- Variables are prefixed using “?” or “\$”
- To select all variables contained in a query: “SELECT * “

Components of SPARQL Queries (4)

```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough
```

```
FROM <http://example.org/cities.ttl>
```

```
WHERE {  
    (Some conditions)  
}
```

Dataset selection:

- FROM or FROM NAMED keyword to specify the RDF dataset
- Indicates the sources for the data against which to find matches

Components of SPARQL Queries (5)

```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough
```

```
FROM <http://example.org/cities.ttl>
```

```
WHERE {  
    (Some condition)  
}
```

Query pattern:

- Specifies *what* we want to query
- Contains graph patterns that are matched against RDF data

Components of SPARQL Queries (6)

```
PREFIX ex: <http://example.org/cities.ttl#>
```

```
SELECT ?borough  
FROM <http://example.org/cities.ttl>  
WHERE {  
    (Some condition)  
} ORDER BY ?borough
```

Sequence modifiers:

- Modify the result set (query answers)
- ORDER BY changes the order of the result set
- LIMIT, OFFSET selects chunks of the result set
- DISTINCT (after SELECT), removes duplicate answers

Query Forms

- There are four different query forms that SPARQL supports:
 - SELECT
 - Return all or a subset of the solution mappings
 - CONSTRUCT
 - Return a set of triples/a graph, where the mappings are filled into a specific graph pattern template
 - ASK
 - Return true or false, depending on whether there is a solution mapping or graph pattern
 - DESCRIBE
 - Return a set of triples / a graph that describes a certain resource (URI)

Agenda

1. Introduction
2. Structure of SPARQL Queries
- 3. Basic Graph Patterns**
4. Querying Multiple (Named) RDF Graphs

Triple Patterns

- Building block of SPARQL queries: **triple patterns**.
 - Similar to RDF triples but with variables (specified with ? or \$).

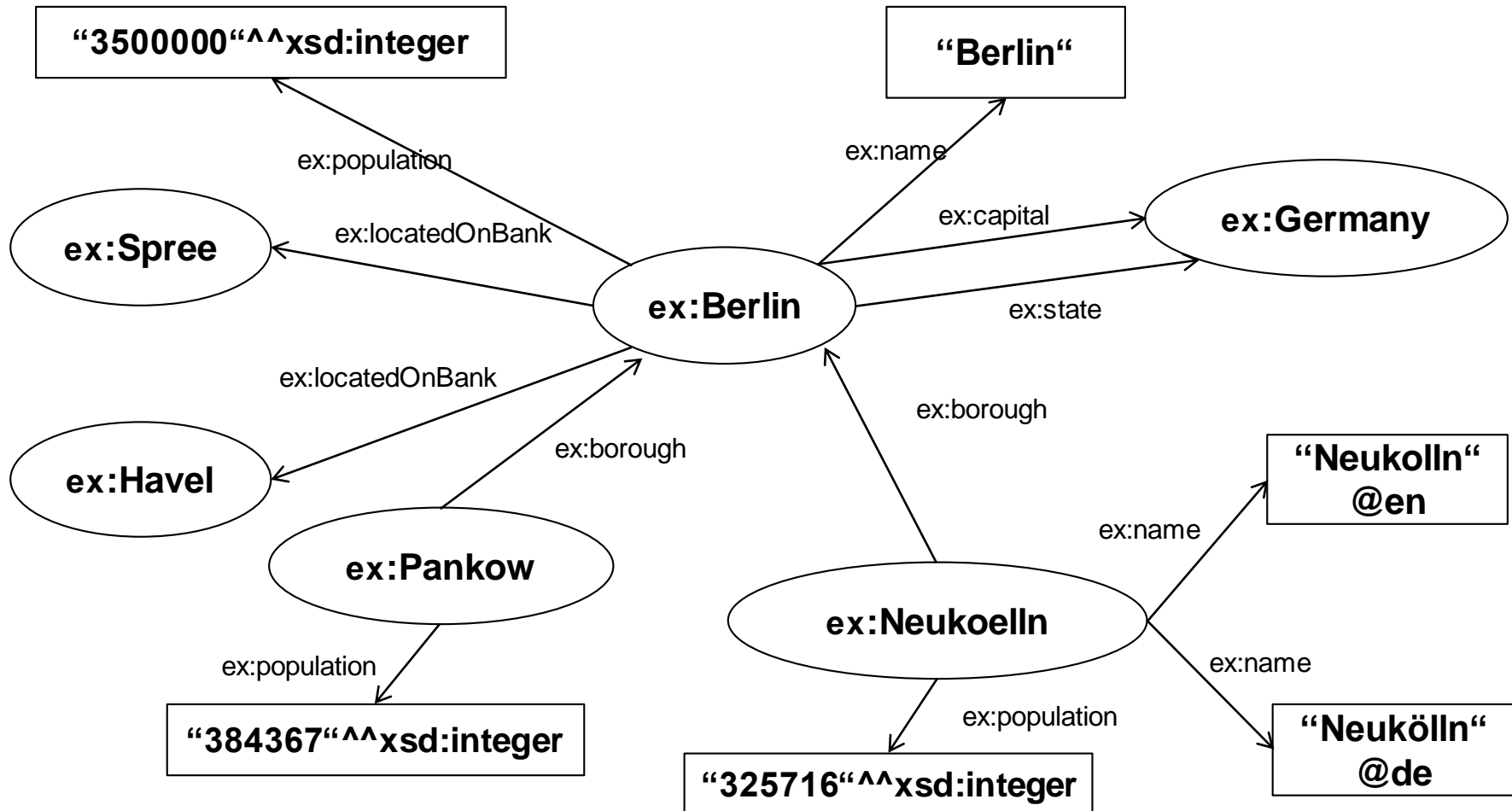
- **Example:** Berlin is the capital of _____.



Or:

`ex:Berlin ex:capital ?x .`

“What are the boroughs of Berlin?”



<http://example.org/cities.ttl>


“What are the boroughs of Berlin?”

```
{  
  ?berlin ex:name "Berlin" .  
  ?borough ex:borough ?berlin .  
}
```

Basic Graph Pattern (1)

- Basic Graph Pattern (BGP) contains several triple patterns.
- BGPs represent *conjunction* of triple patterns.
- **Example:** The following BGP obtains the boroughs of `ex:Berlin` **and** the population of the boroughs

```
{  
    ?borough ex:borough    ex:Berlin .  
    ?borough ex:population ?population .  
}
```

 A variable may be used on the subject, predicate or object position

Basic Graph Pattern (2)

- BGPs can be specified using Turtle syntax

- Example:

```
{ ?borough ex:borough      ?berlin ;  
      ex:population      ?population .  
  ?berlin ex:name         "Berlin" . }
```

- In BGPs blank nodes are treated similar to variables.

- Example:

```
{ _:bn1 ex:name      ?name .  
  _:bn1 ex:population ?population . }
```

- But: blank nodes may only appear on subject and object position of a triple pattern.

- In contrast to variables, one may not specify blank nodes in the query form (e.g., SELECT)

Exercise

Write a SPARQL query into a file that *retrieves all systems* from your file that you uploaded to the solid pod, my RDF file would contain:

```
@prefix : <http://example.org/#> .  
:myProductionSystem a :System ;  
    :hasSubSystem :roboticArm1 , :conveyorBelt2 .  
:roboticArm1 a :System , :RoboticArm ;  
    :hasManufacturer :ABB .  
:conveyorBelt2 a :System ;  
    :hasSpeed "0.1" .
```

- Write the query in a text editor and save it to file myquery.rq
- Use Linked Data-Fu to evaluate your query:

```
ldfu.sh -i http://where.is/your/turtle.ttl -q myquery.rq -
```

Solution Sketch

```
PREFIX : <http://example.org/#>
```

```
SELECT ?thing
```

```
WHERE {
```

```
    ?thing a :System .
```

```
}
```


Solution Sketch

```
PREFIX : <http://example.org/#>
```

```
SELECT *
```

```
WHERE {
```

```
  ?thing a :System ; :hasSubSystem ?sub .
```

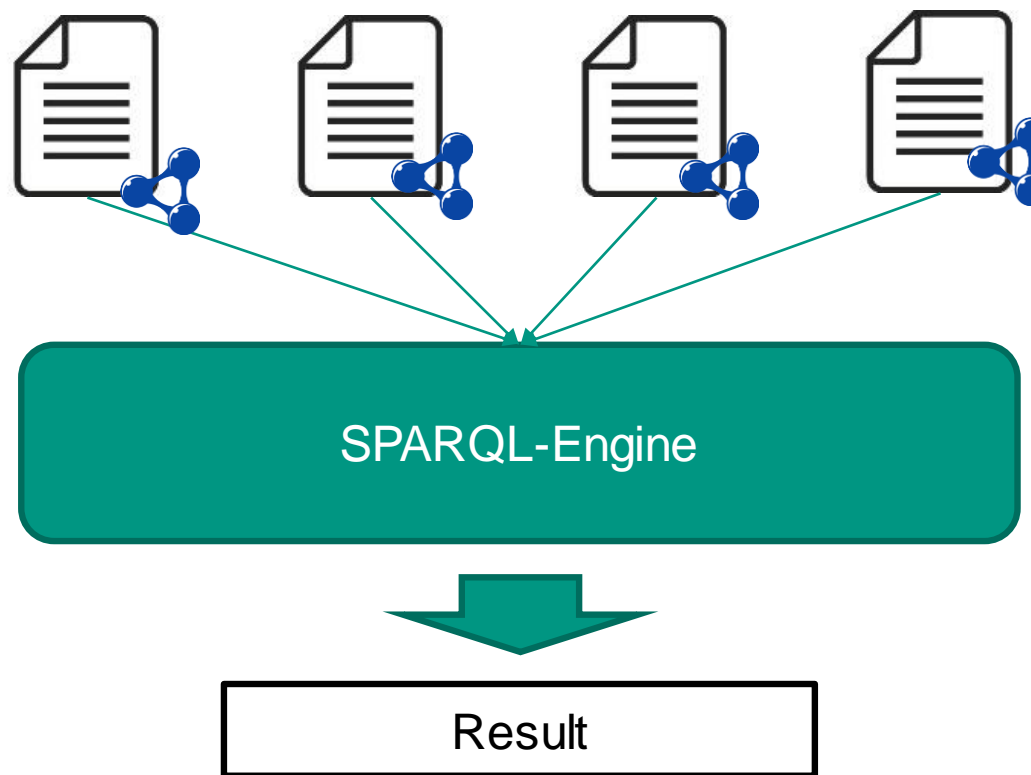
```
}
```

Agenda

1. Introduction
2. Structure of SPARQL Queries
3. Basic Graph Patterns
- 4. Querying Multiple (Named) RDF Graphs**

Multiple Graphs

- Information may be spread over several documents
- Therefore, several documents should be addressable in a query



Multiple Graphs

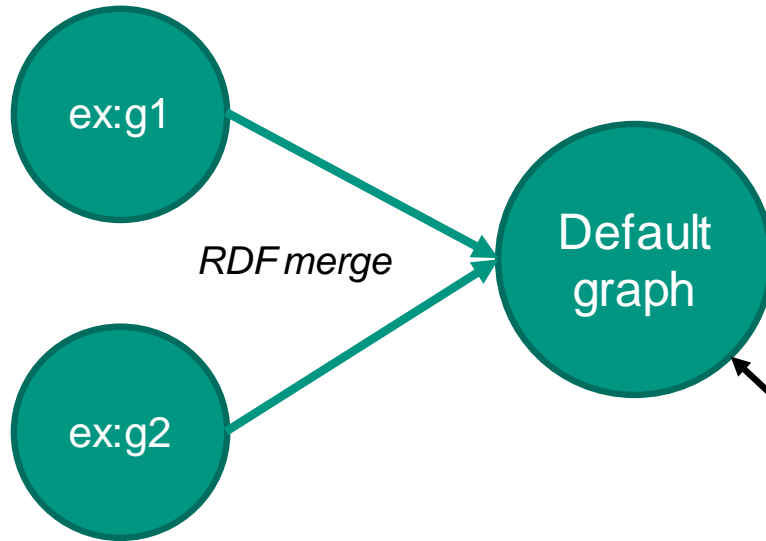
- SPARQL supports handling multiple graphs:
 - These graphs may be different data sources
 - Graphs can be added using the FROM keyword
 - All graphs specified in the FROM clause are combined to a default graph

- SPARQL supports handling of multiple **named** graphs:
 - Using the FROM NAMED keyword
 - These graphs can be accessed using the GRAPH keyword
 - Used to query data from specific graphs



To identify the triples belonging to a graph data we extend the triple model to quadruples, to be able to hold information on the context (name of the graph).

Multiple Graphs - Example



Named Graphs



```
PREFIX ex : <...>
```

```
SELECT *
```

```
FROM ex:g1
```

```
FROM ex:g2
```

```
FROM NAMED ex:g3
```

```
FROM NAMED ex:g4
```

```
WHERE
```

```
{
```

```
...
```

```
?s ?p ?o.
```

```
GRAPH ex:g3 { ... }
```

```
GRAPH ?graph { ... }
```

```
}
```

SPARQL Query Processors vs. SPARQL Endpoints

Query Processor

- Acts as user agent
- Graphs are retrieved via HTTP during query processing
- Default graph is empty, so queries require FROM/FROM NAMED clauses

Endpoint

- Acts as server
- Graphs are indexed and stored on disk during installation (like a database)
- Default graph is configured, so no FROM/FROM NAMED clauses needed

In Linked Data-Fu, `-i` is a surrogate for FROM, remember:

```
ldfu.sh -i http://where.is/your/turtle.ttl -q myquery.rq -
```

Overview of Core SPARQL Features

- Basic concepts: Triple patterns
- SPARQL Query structure:
 - Prefix declarations: **PREFIX**
 - Query forms: **ASK**, **SELECT**, **DESCRIBE**, **CONSTRUCT**
 - Variable projection: Subset of variables that we want to return
 - Dataset selection: **FROM**, **FROM NAMED**
 - Query patterns
 - **Basic Graph Patterns (BGP)**
 - Graph Patterns (**UNION**, **OPTIONAL**, **GRAPH**)
 - Functions (**FILTER**, **BIND AS**)
 - Sequence modifiers: **ORDER BY**, **LIMIT**, **OFFSET**, **DISTINCT**

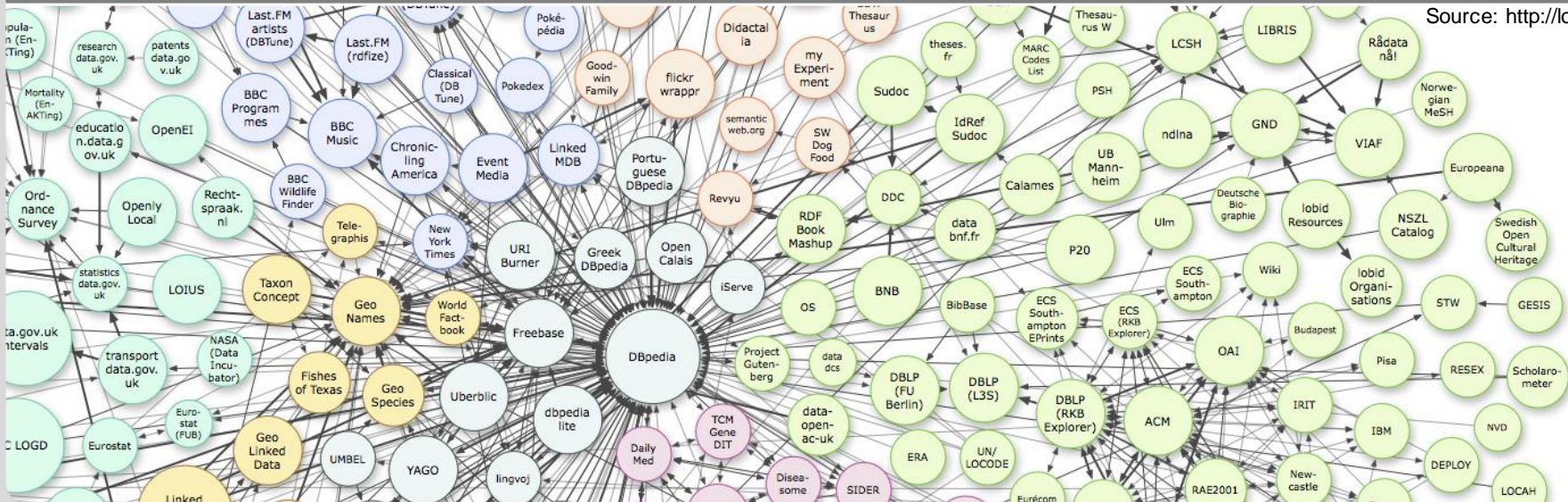
Knowledge Graphs IV

Data Integration, Link Following, and Programming in Rules

Dr. Tobias Käfer

AI4INDUSTRY SUMMER SCHOOL

Source: <http://lod-cloud.net>



Agenda

Rules for:

- **Data Integration**
- Link following
- Programming

Web Standards

- Data providers publish data on web servers
- Data consumers access data with user agents

- Resource Description Framework
 - Graph-structured data: nodes (URIs, literals, blank nodes) and edges (URIs)
 - Interlink information (relationships)

- How can groups of people use RDF to
 - encode a shared understanding of a domain,
 - organise knowledge in a machine-processable way and
 - give meaning to data that can be exploited?



Ontology in Informatics

“An Ontology is a

formal specification

of a shared

conceptualisation

of a domain of interest”

> interpretable by machines

> based on consensus

> describes terminology

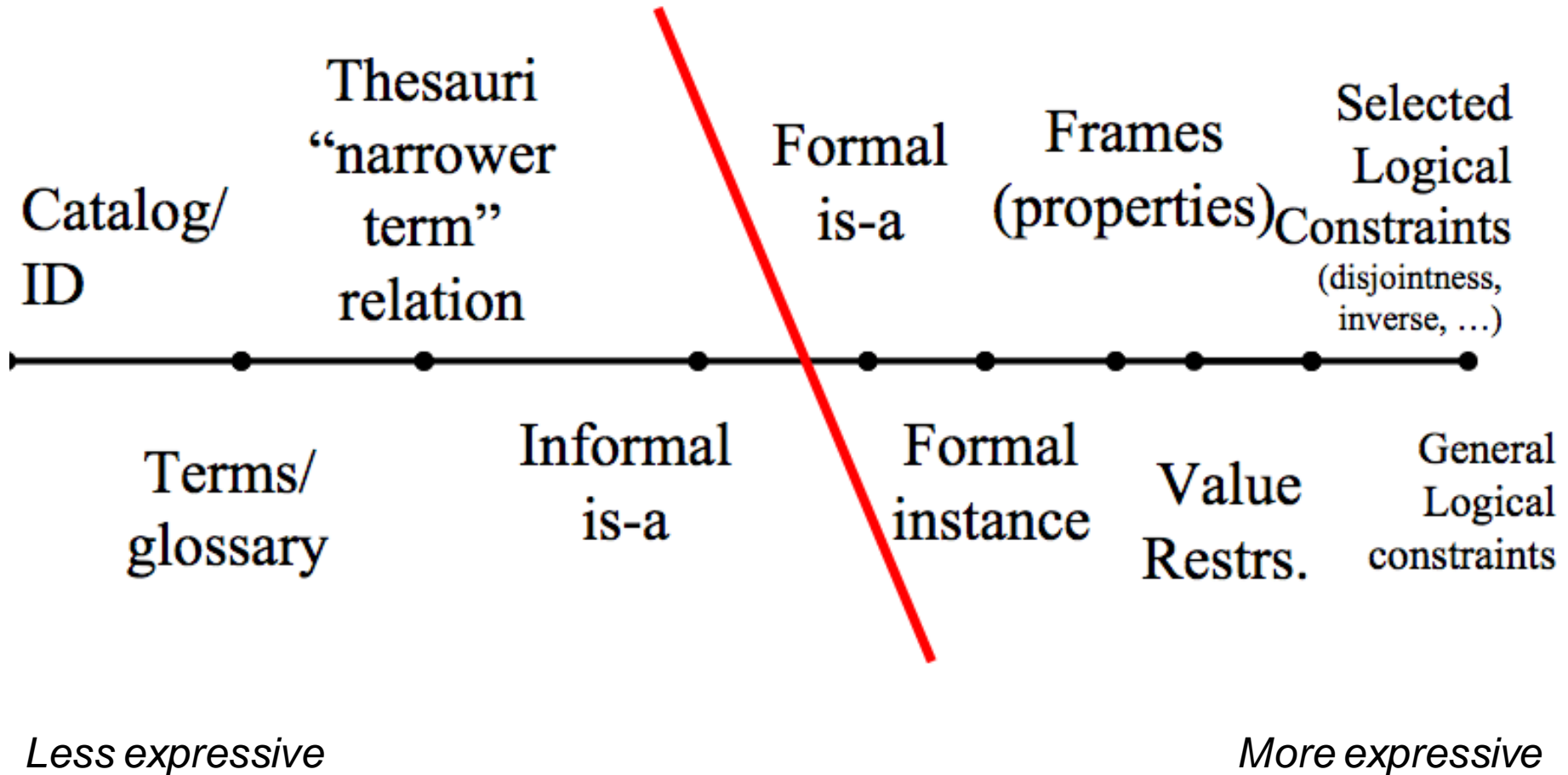
> models a specific topic

Studer, Benjamins and Fensel (1998) based on Gruber (1993) and Borst (1997)

- An ontology is an engineering artefact, consisting of:
 - A specific vocabulary (set of terms - URIs and literals) used to describe a certain reality, plus
 - A set of explicit assumptions regarding the intended meaning of the vocabulary

Ontology Spectrum

From 99 AAAI panel with Gruninger, Lehmann, McGuinness, Ushold, Welty, 2000 Dagstuhl talk by McGuinness



Vocabularies and Vocabulary Descriptions/Ontologies

- Vocabularies are sets of terms, eg.
 - **Individuals:**
Entities identified via a URI or blank node; a vocabulary description may include descriptions of identity (comes later)
 - **Classes:**
Sets of individuals identified via URIs or blank nodes; a vocabulary description may include the characteristics of classes
 - **Properties:**
Properties identified via URIs; a vocabulary description may include the characteristics of properties
- Ontologies (vocabulary descriptions) are collections of terms together with their (logically) defined meaning

Core Semantic Web Vocabularies

- To bootstrap meaning of vocabulary terms, we could use terms that are widely agreed; how about we use mathematics?
- The W3C standardised fundamental vocabularies (based on mathematics) that can be used to express other vocabularies.
- **RDF¹**: We consider the RDF vocabulary, i.e., the URIs defined as part of the RDF W3C Recommendation.
- **RDFS²**: We examine RDF Schema, a simple ontology language that offers means to describe characteristics of classes and properties.
- Throughout the slides, assume the following prefix declarations:
@prefix rdf: <<http://www.w3.org/1999/02/22-rdf-syntax-ns#>> .
@prefix rdfs: <<http://www.w3.org/2000/01/rdf-schema#>> .
@prefix : <#> .

¹ <https://www.w3.org/TR/rdf11-primer/>

² <https://www.w3.org/TR/rdf-schema/>

Why Formal Semantics?

- After introduction of RDF and RDFS, criticism of tool developers: different tools were incompatible (despite the existing specification)
- E.g.:
 - Same RDF document
 - Same entailment relation
 - **Different results**
- Thus, a model-theoretic semantics was defined for entailment:
 - provides a formal specification of when truth is preserved by transformations of RDF or operations which derive RDF triples from other RDF triples (logical consequence).

A Classical Example for Entailment

- Premise: All men are mortal
- Premise: Socrates is a man
- Conclusion: Socrates is mortal

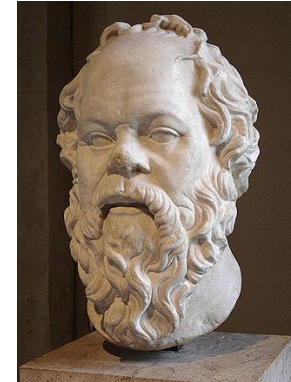


Photo from Wikipedia

- In RDF using RDFS vocabulary:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

```
@prefix : <#> .
```

```
:Man rdfs:subClassOf :Mortal .      # premise
```

```
:Socrates a :Man .                  # premise
```

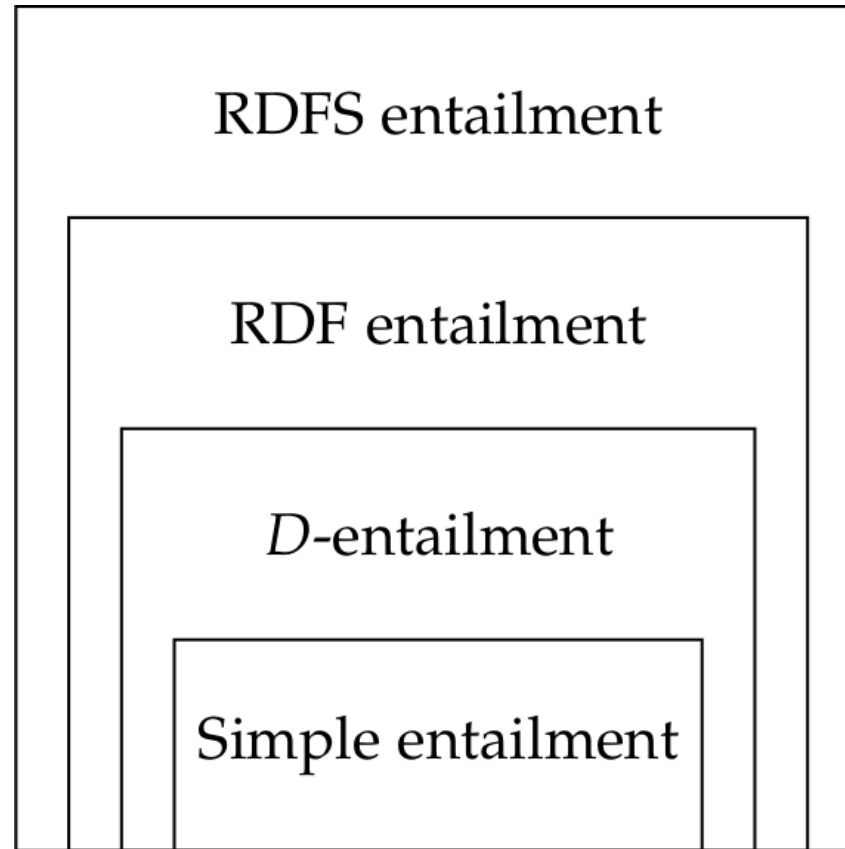
```
:Socrates a :Mortal .                # conclusion
```


Layered Entailment

expressivity



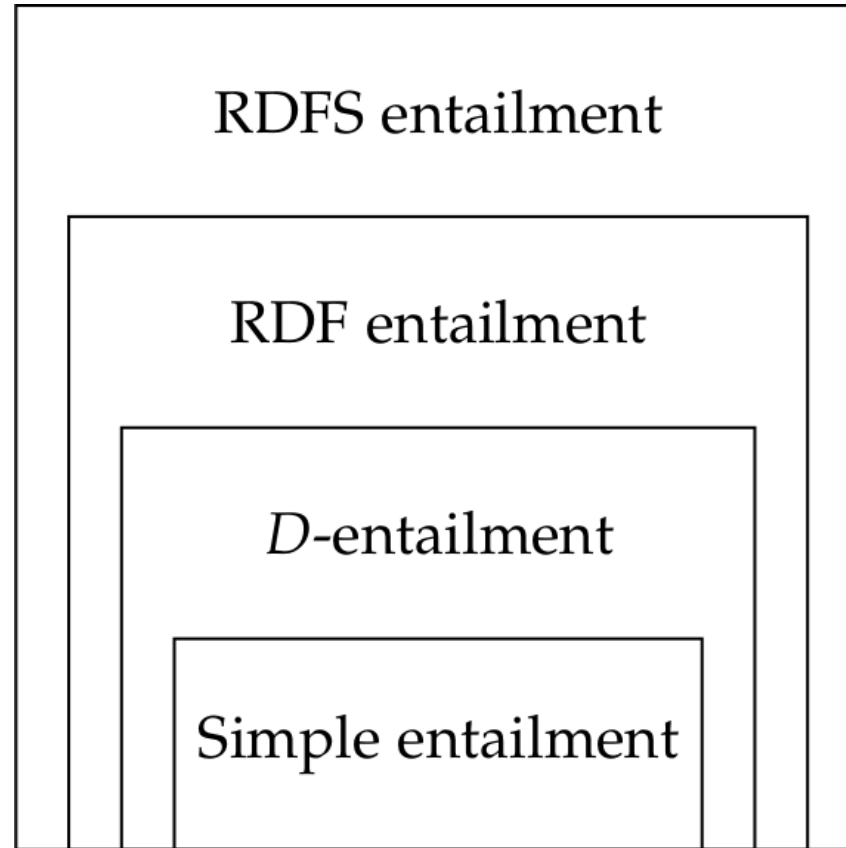
OWL



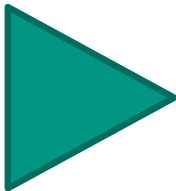
- Higher expressivity → More logical conclusions (entailments) and higher computational complexity.
- Defined mathematically via sets and functions using model theory
- Rules as way to implement the mentioned entailment regimes.

Layered Entailment

expressivity



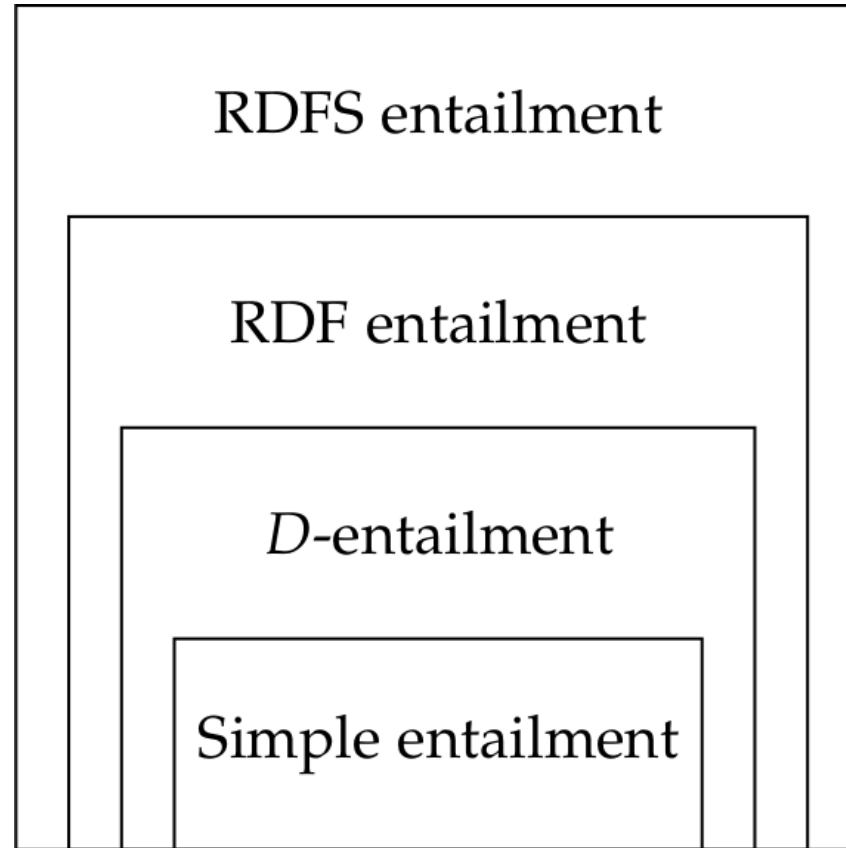
Interesting for bootstrapping the definitions via sets and functions



- Higher expressivity → More logical conclusions (entailments) and higher computational complexity.
- Defined mathematically via sets and functions using model theory
- Rules as way to implement the mentioned entailment regimes.

Layered Entailment

expressivity



Tells you how to formally define typed literals and when values are the same



- Higher expressivity → More logical conclusions (entailments) and higher computational complexity.
- Defined mathematically via sets and functions using model theory
- Rules as way to implement the mentioned entailment regimes.

RDF VOCABULARY AND ENTAILMENT

RDF Vocabulary

- The RDF vocabulary allows to make basic statements about resources and triples
- The following table lists all RDF terms, other than the container membership properties `rdf:_1`, `rdf:_2`, `rdf:_3` ...

Class URIs	Property URIs	Datatype URIs	Instance URIs
<code>rdf:Property</code>	<code>rdf:type</code>	<code>rdf:langString</code>	<code>rdf:nil</code>
<code>rdf:List</code>	<code>rdf:first</code>	<code>rdf:HTML</code>	
<code>rdf:Bag</code>	<code>rdf:rest</code>	<code>rdf:XMLLiteral</code>	
<code>rdf:Alt</code>	<code>rdf:value</code>	<code>rdf:PlainLiteral</code>	
<code>rdf:Seq</code>	<code>rdf:subject</code>		
<code>rdf:Statement</code>	<code>rdf:predicate</code>		
	<code>rdf:object</code>		

Formal Instances (rdf:type)

- The URI `rdf:type` allows to specify that a resource is an instance of something
- For example, the following describes `:Berlin` as being a `:City`, as follows:

```
:Berlin rdf:type :City .
```

What was the shortcut for `rdf:type` in the Turtle syntax?

rdf:Property

- The term `rdf:Property` denotes the resource that contains as members all resources occurring on predicate position in RDF triples
- Given an RDF graph

`:s :p :o .`

- we can conclude

`:p rdf:type rdf:Property .`

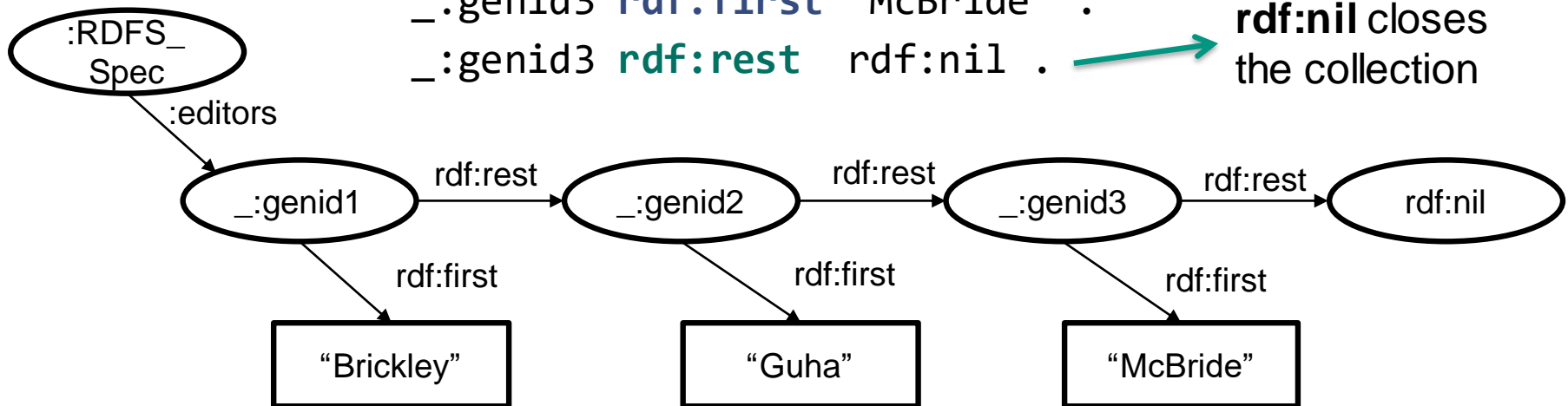
Collections aka `rdf:Lists`

- A collection is a **closed group** of elements
- Example: Editors of the RDFS spec “Brickley”, “Guha”, “McBride”

```

:RDFS_Spec :editors _:genid1 .
_:genid1 rdf:first "Brickley" .
_:genid1 rdf:rest _:genid2 .
_:genid2 rdf:first "Guha" .
_:genid2 rdf:rest _:genid3 .
_:genid3 rdf:first "McBride" .
_:genid3 rdf:rest rdf:nil .
  
```

`rdf:nil` closes the collection



RDF Lists

- Lists can only appear in subject or object position of a triple
- The class `rdf:List` contains the RDF lists
- Turtle provides a syntax abbreviation for specifying collections (“lists structures”) by enclosing the RDF terms with ()

```
#the object of this triple is the RDF collection blank node  
:RDFS_Spec :editors ( “Brickley” “Guha” “McBride” ) .
```

RDF Axiomatic Triples

- What is an axiom?
 - A self-evident or universally recognised truth ¹
 - An established rule, principle, or law ¹
- The following triples have to be true in any RDF interpretation, by definition:

```
rdf:type rdf:type rdf:Property .  
rdf:subject rdf:type rdf:Property .  
rdf:predicate rdf:type rdf:Property .  
rdf:object rdf:type rdf:Property .  
rdf:first rdf:type rdf:Property .  
rdf:rest rdf:type rdf:Property .  
rdf:value rdf:type rdf:Property .  
rdf:nil rdf:type rdf:List .  
rdf:_1 rdf:type rdf:Property .  
rdf:_2 rdf:type rdf:Property .  
...
```



Since the elements of a container may be infinite, the application of the axiomatic triples results in an infinite interpretation

¹ <http://www.thefreedictionary.com/axiom>

RDF Entailment Patterns

- The following entailment patterns can be used as an easy way to apply the RDF entailment rules to a graph
- Variables are denoted with a “?” (as in SPARQL)
- The patterns are applied by assigning values to the variables in the “If” statement and adding (inferring) the “Then” statement
- Patterns*:

	If ...	Then ...
rdfD1	<code>?x ?p "sss"^^ddd .</code>	<code>?x ?p _:n . _:n rdf:type ddd .</code>
rdfD2	<code>?x ?p ?y .</code>	<code>?p rdf:type rdf:Property .</code>

- Alternative pattern to rdfD1 (assuming generalised RDF)

	If ...	Then ...
GrdfD1	<code>?x ?p "sss"^^ddd .</code>	<code>"sss"^^ddd rdf:type ddd .</code>

- For the following examples we consider our graph: <http://example.org/cities.ttl>

* "sss" represents some Unicode string

RDFS VOCABULARY AND ENTAILMENT

RDFS Intuition and Vocabulary

- The RDFS vocabulary allows to make statements about classes of things and properties and to provide documentation to resources
- RDFS entailment is a lot about the semantics of those classes and properties
- RDFS terms are:

Properties:

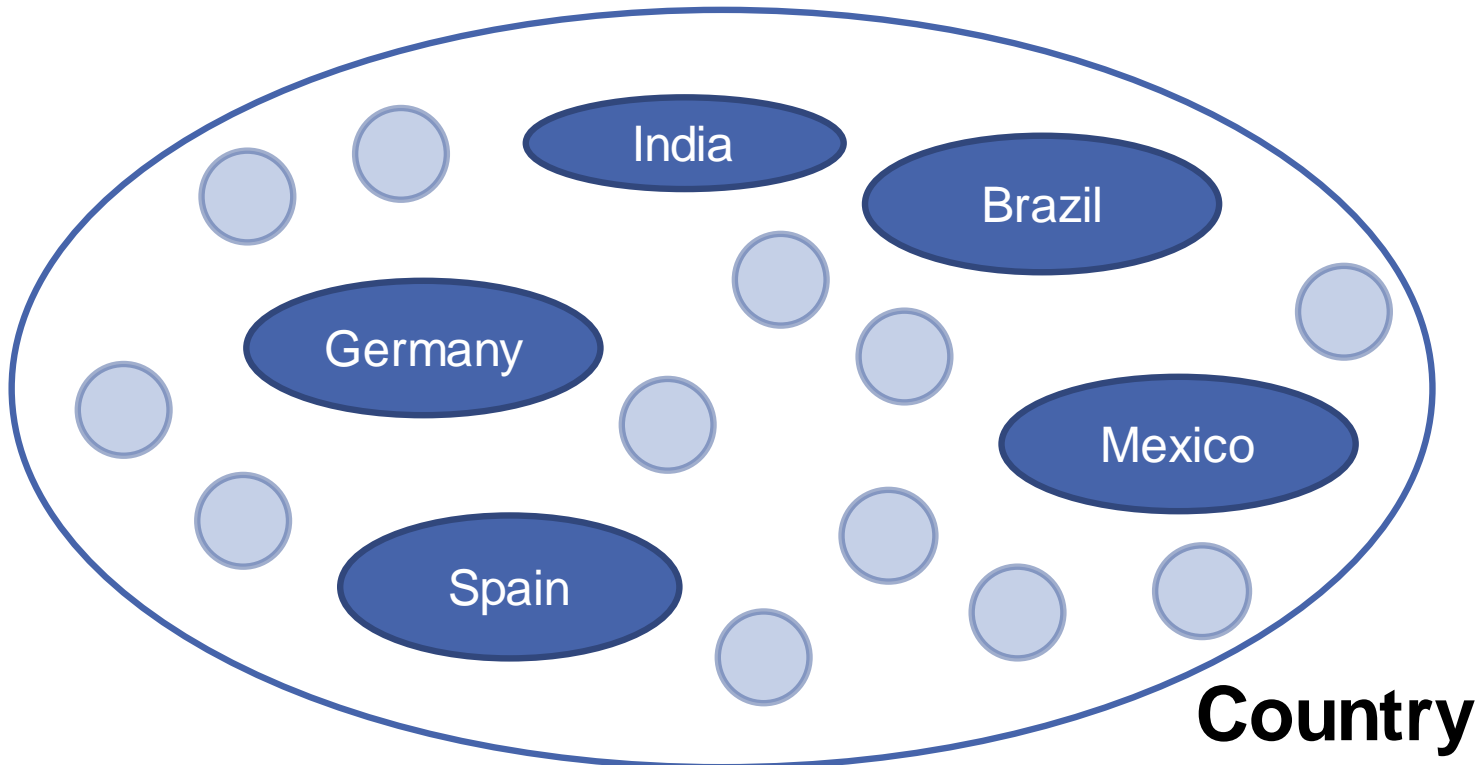
`rdfs:domain`
`rdfs:range`
`rdfs:subClassOf`
`rdfs:subPropertyOf`
`rdfs:member`
`rdfs:comment`
`rdfs:seeAlso`
`rdfs:isDefinedBy`
`rdfs:label`

Classes:

`rdfs:Resource`
`rdfs:Literal`
`rdfs:Datatype`
`rdfs:Class`
`rdfs:Container`
`rdfs:ContainerMembershipProperty`

¹ <http://www.w3.org/TR/rdf-schema/>

Classes – Analogy to Set Theory



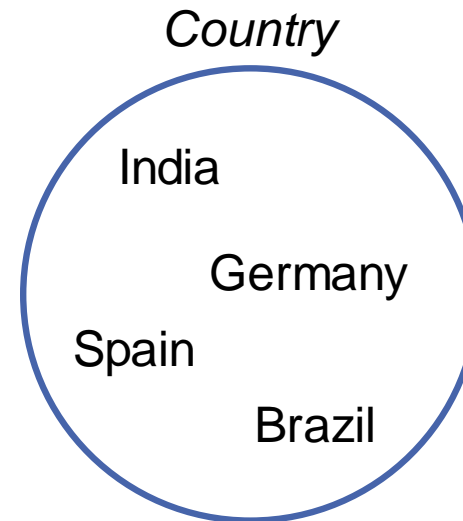
- Individuals represent elements of a set
- Classes represent a set that is identified via a URI or a blank node

Classes: Example (1)

- To define the class:
 - `rdf:type rdfs:Class`

- To relate instances to the class:
 - `rdf:type`

The class of *countries*



URI of the class ← `:Country` `rdf:type rdfs:Class .`

Instances of the class

```

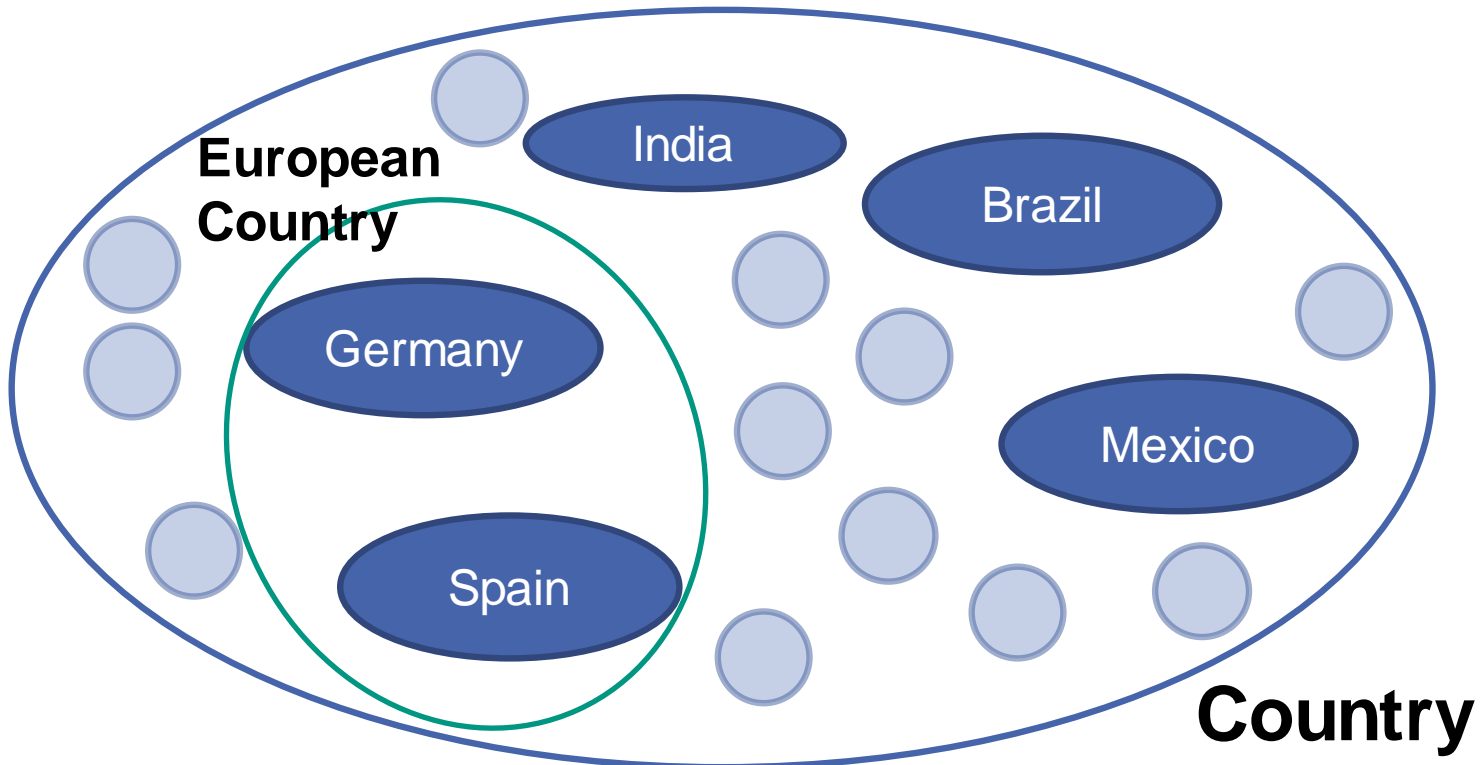
:India    rdf:type    :Country .
:Germany  rdf:type    :Country .
:Spain    rdf:type    :Country .
:Brazil   rdf:type    :Country .
  
```

Class Hierarchies

- Given several classes, we can specify a hierarchical relationship between them: the subclass relation
- In RDFS, a class may have several subclasses, and a class can be a subclass of several (super)classes
- Example:
 - We have two classes: `:Country` and `:EuropeanCountry`
 - We want to say that everything that is a European country is also a country
 - That is, `:EuropeanCountry` is a subclass of `:Country`
 - We use `rdfs:subClassOf` to specify the subclass relationship:

```
:EuropeanCountry rdfs:subClassOf :Country .
```


Class Hierarchies – Analogy to Set Theory



- `rdf:type` corresponds to \in
- `rdfs:subClassOf` corresponds to \subseteq

RDFS Axiomatic Triples

```
rd:type rdfs:domain rdfs:Resource ; rdfs:range rdfs:Class .
rdfs:domain rdfs:domain rdf:Property ; rdfs:range rdfs:Class .
rdfs:range rdfs:domain rdf:Property ; rdfs:range rdfs:Class .
rdfs:subPropertyOf rdfs:domain rdf:Property ; rdfs:range rdf:Property .
rdfs:subClassOf rdfs:domain rdfs:Class ; rdfs:range rdfs:Class .
rdf:subject rdfs:domain rdf:Statement ; rdfs:range rdfs:Resource .
rdf:predicate rdfs:domain rdf:Statement ; rdfs:range rdfs:Resource .
rdf:object rdfs:domain rdf:Statement ; rdfs:range rdfs:Resource .
rdfs:member rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .
rdf:first rdfs:domain rdf:List ; rdfs:range rdfs:Resource .
rdf:rest rdfs:domain rdf:List ; rdfs:range rdf:List .
rdfs:seeAlso rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .
rdfs:isDefinedBy rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .
rdfs:comment rdfs:domain rdfs:Resource ; rdfs:range rdfs:Literal .
rdfs:label rdfs:domain rdfs:Resource ; rdfs:range rdfs:Literal .
rdf:value rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .

rdf:Alt rdfs:subClassOf rdfs:Container .
rdf:Bag rdfs:subClassOf rdfs:Container .
rdf:Seq rdfs:subClassOf rdfs:Container .
rdfs:ContainerMembershipProperty rdfs:subClassOf rdf:Property .

rdfs:isDefinedBy rdfs:subPropertyOf rdfs:seeAlso .

rdfs:Datatype rdfs:subClassOf rdfs:Class .

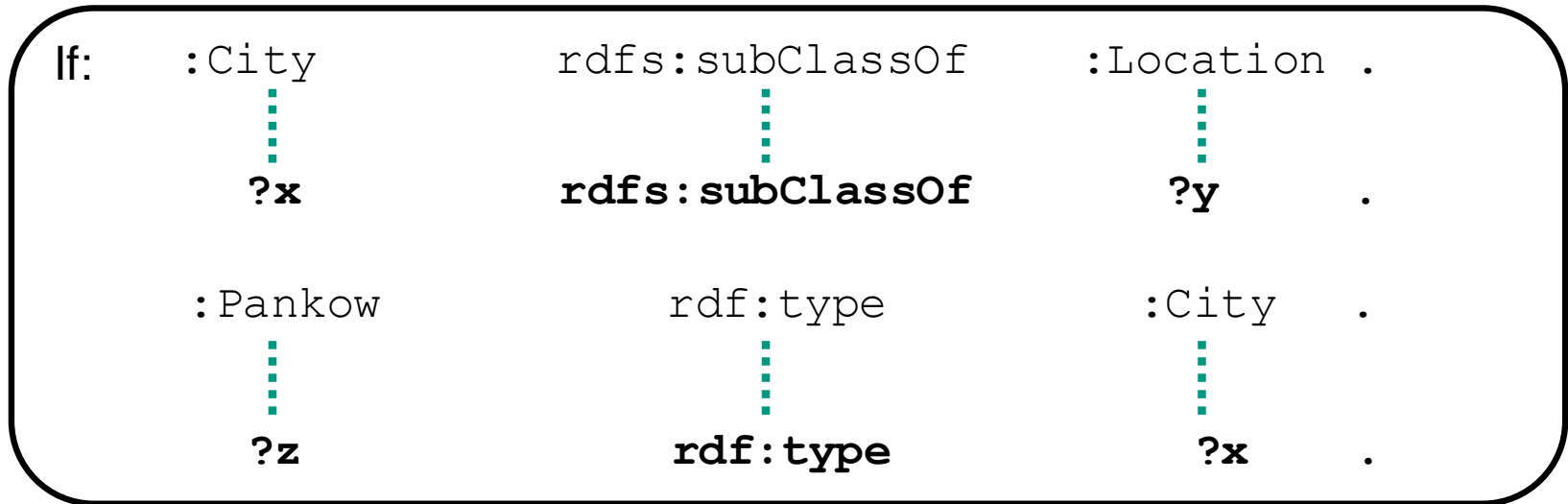
rdf:_1 a rdfs:ContainerMembershipProperty ; rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .
rdf:_2 a rdfs:ContainerMembershipProperty ; rdfs:domain rdfs:Resource ; rdfs:range rdfs:Resource .
...
```

RDFS Entailment Patterns

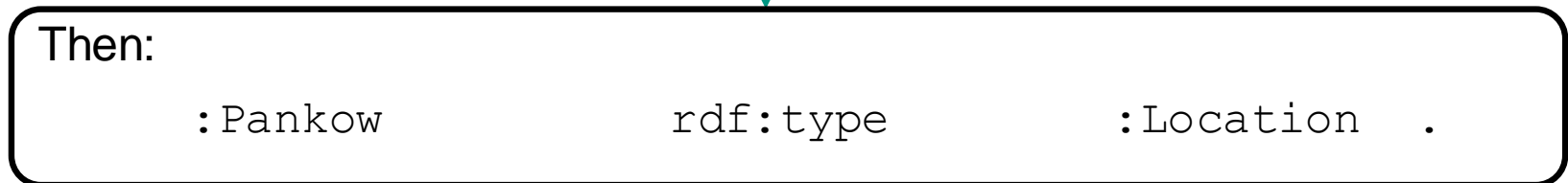
	If...	Then...
rdfs1	Any URI ddd in D	ddd rdf:type rdfs:Datatype .
rdfs2	?p rdfs:domain ?x . ?y ?p ?z .	?y rdf:type ?x .
rdfs3	?p rdfs:range ?x . ?y ?p ?z .	?z rdf:type ?x .
rdfs4a	?x ?p ?z .	?x rdf:type rdfs:Resource .
rdfs4b	?y ?p ?z .	?z rdf:type rdfs:Resource .
rdfs5	?x rdfs:subPropertyOf ?y . ?y rdfs:subPropertyOf ?z .	?x rdfs:subPropertyOf ?z .
rdfs6	?x rdf:type rdf:Property .	?x rdfs:subPropertyOf ?x .
rdfs7	?p2 rdfs:subPropertyOf ?p1 . ?x ?p2 ?y .	?x ?p1 ?y .
rdfs8	?x rdf:type rdfs:Class .	?x rdfs:subClassOf rdfs:Resource .
rdfs9	?x rdfs:subClassOf ?y . ?z rdf:type ?x .	?z rdf:type ?y .
rdfs10	?x rdf:type rdfs:Class .	?x rdfs:subClassOf ?x .
rdfs11	?x rdfs:subClassOf ?y . ?y rdfs:subClassOf ?z .	?x rdfs:subClassOf ?z .
rdfs12	?x rdf:type rdfs:ContainerMembershipProperty .	?x rdfs:subPropertyOf rdfs:member .
rdfs13	?x rdf:type rdfs:Datatype .	?x rdfs:subClassOf rdfs:Literal .

RDFS Entailment Patterns – rdfs9

■ Example:



?z rdf:type ?y .



MORE EXPRESSIVE ENTAILMENT REGIMES

Extending RDFS with other useful features

- OWL is a fairly expressive ontology language
- RDFS plus, RDFS 3.0, OWL LD „extend“ RDFS entailment with the semantics of some terms from OWL such as:
 - owl:sameAs
 - owl:equivalentProperty
 - owl:inverseOf
 - ...

IMPLEMENTING ENTAILMENT

Approaches for Evaluating Entailment Patterns

Examples for when users are interested in the derived knowledge

- Queries, eg. of downstream applications
- Conditions for actions outside the realm of the

Approaches:

- Materialization / forward chaining
- Query rewriting / backward chaining
- Hybrid approaches

Algorithm for Materialisation: Extend the Graph with Inferred Triples

```
Require: assertions ▷ Graph  
Require: rules ▷ Derivation rules  
var data, oldData: set<triple>  
var fixpointReached: boolean  
data.clear()  
data.add(assertions)  
repeat ▷ Loop for determining the fixpoint  
  fixpointReached ← true  
  for rule : rules do  
    if rule.matches(data) then  
      oldData = data.copy()  
      if rule.type==derivation then  
        data.add(rule.match(data).data)  
      end if  
      if ! data.copy().remove(oldData).isEmpty() then  
        fixpointReached ← false  
      end if  
    end if  
  end for  
until fixpointReached
```

Notation3 Rule Syntax

- We introduce Notation3 (N3), a superset of Turtle syntax
- N3 extends the RDF data model with
 - variables (prefixed with a ?) and
 - graph quoting (via { }) for subject and object of a triple
- Together with a URI for implication (`<http://www.w3.org/2000/10/swap/log#implies>`, shortcut: `=>`), we can encode rules in N3 syntax.

Notation3 Derivation Rules

- A N3 rule is of the form $\{ \text{body} \} \Rightarrow \{ \text{head} \} .$
- The **body** of a rule (the „if“ part) is also called antecedent
- The **head** of a rule (the „then“ part) is also called consequent
- The body is a set of triple patterns: a BGP
- The head is a graph template

Example: RDFS Entailment Patterns as Rules

	if S contains	then S RDFS-entails recognising D
<i>rdfs1</i>	any URI <i>ddd</i> in D	<i>ddd</i> <code>rdf:type</code> <code>rdfs:Datatype</code> .
<i>rdfs2</i>	<code>?p rdfs:domain ?x . ?y ?p ?z .</code>	<code>?y rdf:type ?x .</code>
<i>rdfs3</i>	<code>?p rdfs:range ?x . ?y ?p ?z .</code>	<code>?z rdf:type ?x .</code>
<i>rdfs4a</i>	<code>?x ?p ?y .</code>	<code>?x rdf:type rdfs:Resource .</code>
<i>rdfs4b</i>	<code>?x ?p ?y .</code>	<code>?y rdf:type rdfs:Resource .</code>
<i>rdfs5</i>	<code>?x rdfs:subPropertyOf ?y . ?y rdfs:subPropertyOf ?z .</code>	<code>?x rdfs:subPropertyOf ?z .</code>

Entailment pattern *rdfs5* as derivation rule:

@prefix `rdfs:` <<http://www.w3.org/2000/01/rdf-schema#>> .

{ `?x rdfs:subPropertyOf ?y .`
`?y rdfs:subPropertyOf ?z .` } => { `?x rdfs:subPropertyOf ?z .` } .

Exercise: Query Evaluation with Materialization

- Given the following RDF graph G available at <http://example.org/persons> and the SPARQL expression E. Assume all the prefix definitions.

```
:Magneto a dbo:Person ;  
          rdfs:label "Max Eisenhardt" .  
foaf:Person rdfs:subClassOf foaf:Agent .  
foaf:Person owl:equivalentClass dbo:Person .
```

- Query Q:
SELECT ?p WHERE { ?p a foaf:Agent }
- Entailment regime R with the following set of rules:
{ { ?x owl:equivalentClass ?y . } => { ?y owl:equivalentClass ?x . },
 { ?x owl:equivalentClass ?y . ?a rdf:type ?x . } => { ?a rdf:type ?y . },
 { ?x rdfs:subClassOf ?y . ?a rdf:type ?x . } => { ?a rdf:type ?y . } }
- Materialise R on the graph G and evaluate Q.

Agenda

Rules for:

- Reasoning
- **Link following**
- Programming

How to Combine Link-Following and Querying?

- The Linked Data principles point towards combining web architecture with knowledge representation
- But all the bits and pieces we have seen so far do not fit yet:
- We can dereference URIs of things via HTTP, view the resulting RDF and follow links (e.g., in the RDF browser)

OR

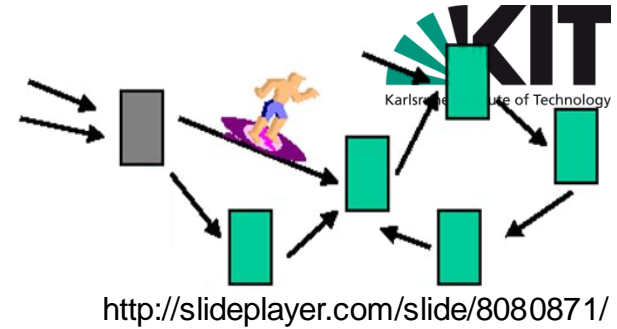
- We can query RDF documents with SPARQL given a fixed set of URIs to documents in FROM/FROM NAMED clauses

BUT

- How do we query Linked Data while following links?

General User Agent Model

- Characteristics of a generic user agent on the web (e.g., web browser):



1. The user agent starts its interaction based on a specific seed URI

2. The user agent performs HTTP requests on URIs and parses the response

3. Based on the response the user agent has one or multiple choices as to which interaction to perform next

4. The user agent decides which link to follow and initiates a new request

Reduction to What we Learnt: Crawl-Index-Serve

- Crawl-index-serve architecture for Linked Data:
 - Crawl Linked Data (on the level of documents, parse RDF into quads), specify the amount of hops for expansion
 - Load the resulting RDF Dataset (quads) into a SPARQL store
 - Serve query solutions from the SPARQL store
- Materialising the data (crawling, indexing) takes time
- Indexes of Linked Data get outdated [1]
- Indiscriminate expansion of links
- Requires many systems (crawler, SPARQL store), server capacity
- Possibly too much overhead if users are interested in the solution to a single query
- **How about more clever user agents? That run on people's computers?**
- **That access live data?**

[1] Käfer, Umbrich, Abdelgayed, O'Byrne, Hogan: Observing Linked Data Dynamics. Proc. 11th Extended Semantic Web Conference (2013).

Linked Data Principles¹



Tim Berners-Lee presenting Linked Data. TED CC-BY-ND

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)
4. Include links to other URIs, so that they can discover more things.

¹ <http://www.w3.org/DesignIssues/LinkedData.html>

Two Perspectives on the Linked Data Principles

Server (Publisher)

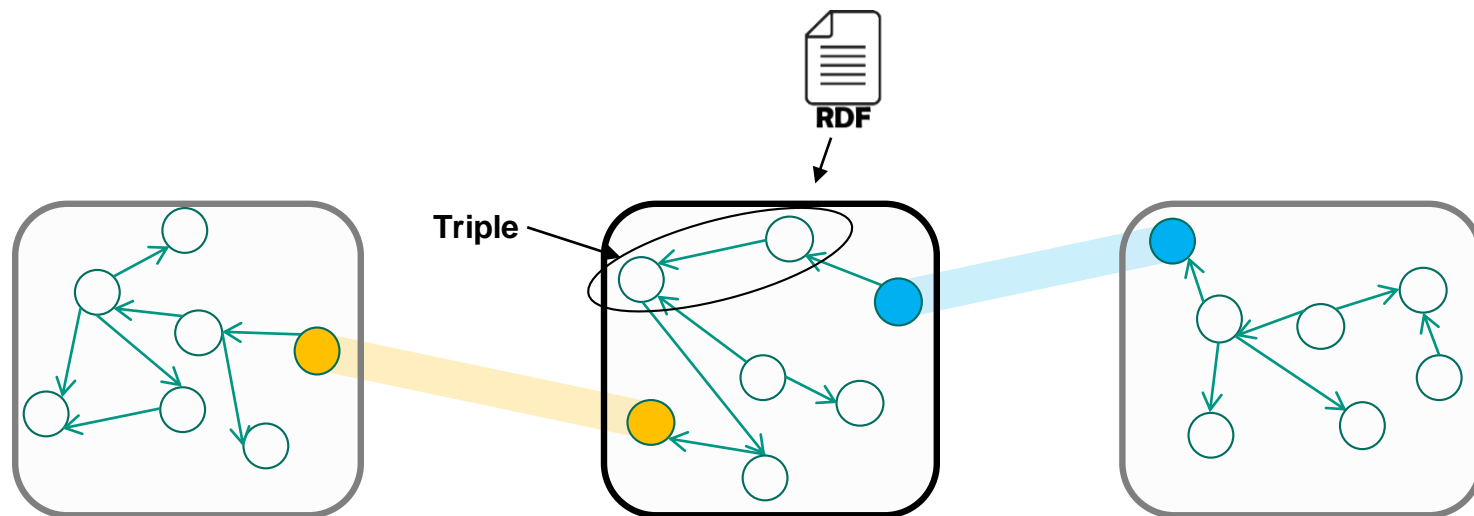
1. Coin URIs to name things. ✓
2. Use a HTTP server to provide access to documents. ✓
3. Upon receiving a request for a URI, the server returns useful information (about the URI in the request) in RDF and RDF Schema. ✓
4. The “useful information” the server returns in the RDF document includes links to other URIs (on other servers). ✓

User Agent (Consumer)

1. Assume URIs as names for things. ✓
2. User agents look up HTTP URIs. ✓
3. User agents process RDF/RDFS documents containing useful information and provide the ability to evaluate SPARQL queries. ✓
4. User agents can discover more things via accessing links to other URIs. ✗

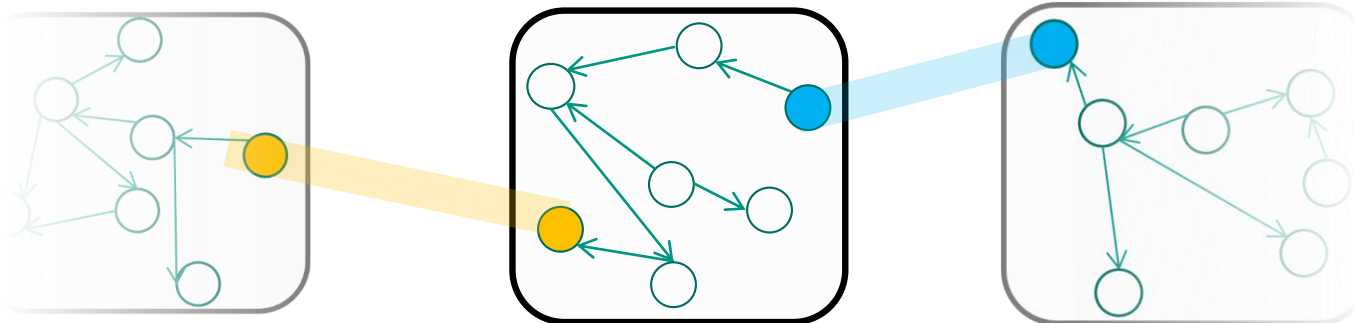
Operating on a Fixed RDF Dataset

- Until now, both in querying and with entailment, we have assumed that the data over which we operate is fixed at the beginning of the processing.
- That is, we have assumed a fixed RDF Dataset.



Operating on the Web as RDF Dataset

- We would like to use the entire Linked Data web, i.e., a huge RDF dataset *Web*, as basis for querying.
- But the web is too big; downloading the entire web is impractical.
- One of the core features of the web are hyperlinks.
- A user agent starts from an entry point and then follows links.
- Following links can lead to hitherto unknown servers, with unknown data of unknown schema.



- How can we specify a (finite) RDF dataset in a flexible way?

Dereferencing URIs¹

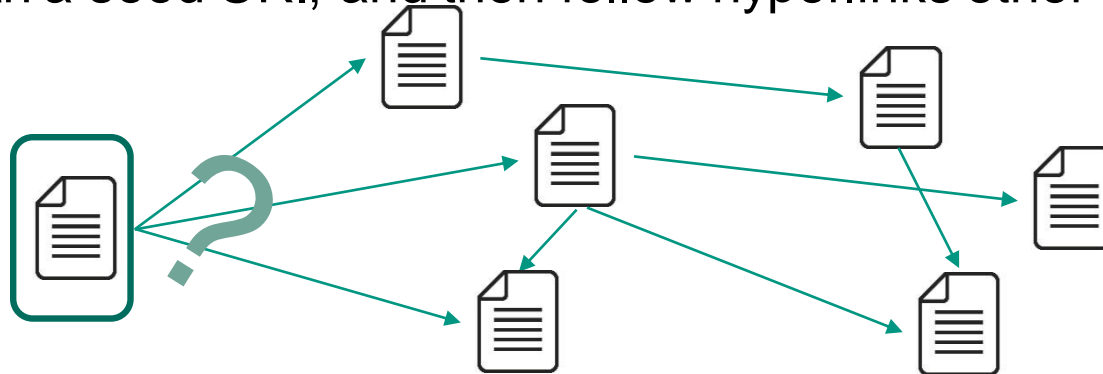
- We define ways for accessing RDF graphs published on the web as Linked Data
- Linked Data provides a combination of knowledge representation language (RDF, RDFS) and web architecture (HTTP)
- A key characteristic of Linked Data is the tight connection between an identifier and a source, i.e., the name for a thing² is associated with the document where one can find related information

¹ See also in Chapter 2


² “non-information” resource, not defined in any RFC, which only know “other resources” and “information resources”.

Motivation for Request Rules

- We want to specify an RDF dataset constructed during query evaluation
- Start with a seed URI, and then follow hyperlinks other data sources



- Given a set of links within a dataset we need to specify:
 - Which links to follow?
 - Order of following links?
 - How far to follow links?

 Request rules as a way to specify traversal

Representing HTTP Requests in RDF

- To model HTTP requests in RDF we require a vocabulary for HTTP requests (and headers)
- Namespace for the core terms of HTTP vocabulary¹ in RDF:

`http://www.w3.org/2011/http#`

- We also make use of a vocabulary for HTTP methods and HTTP headers
- Using the HTTP vocabulary, we are able to represent any kind of HTTP-interaction using RDF

¹<http://www.w3.org/TR/HTTP-in-RDF10/>

HTTP Vocabulary: Example

- Let us consider the simple request:

```
GET    /article/420 HTTP/1.1
Host:  example.org
Accept: text/turtle
```

- Represented using the HTTP vocabulary:

```
@prefix http: <http://www.w3.org/2011/http#> .
@prefix httpm: <http://www.w3.org/2011/http-methods#> .
@prefix httph: <http://www.w3.org/2011/http-headers#> .
```

```
[ ]      a      http:Request;
          http:requestURI      "/article/420";
          http:httpVersion      "1.1";
          http:mthd      httpm:GET;
          http:headers      ( [ http:hdrName      httph:host ;
                               http:fieldValue      "example.org" ]
                               [ http:hdrName      httph:accept ;
                               http:fieldValue      "text/turtle" ] ) .
```

Syntax of Request Rules in Notation3

- Request with both fixed and variable request targets can appear as the head of a request rule

Definition 29 (Request Rule) *Let q be a graph pattern and r a request represented in the HTTP vocabulary. A request rule is an N3 triple with the form $\{ q \} \Rightarrow \{ r \}$.*

- Form:

$$\overbrace{\{ \text{graph pattern} \}}^{\text{body}} \Rightarrow \overbrace{\{ \text{request template} \}}^{\text{head}} .$$

- Properties:
 - Existential: *head* contains blank nodes
 - Safe: all variable are part of both *head* and *body* of the rule

Request Rule – Example 1

- Request URIs of people that Andreas knows

```
@prefix http: <http://www.w3.org/2011/http#> .  
@prefix httpm: <http://www.w3.org/2011/http-methods#> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
{  
  <http://harth.org/andreas/foaf#ah> foaf:knows ?person .  
} => {  
  [] http:method hmthd:GET ;  
    http:requestURI ?person . } .
```



Request rules allow for fine-grained manner to determine which resources to retrieve and which links to follow

Request Rule – Example 2

- The following rule dereferences all class URIs that occur in the data:

```
@prefix http: <http://www.w3.org/2011/http#> .  
@prefix httpm: <http://www.w3.org/2011/http-methods#> .  
  
{  
  ?s a ?c .  
} => {  
  [] http:method httpm:GET ;  
    http:requestURI ?c .  
} .
```

Require: assertions ▷ Graph

Require: rules ▷ Derivation and GET request rules

```
var data, oldData: set<triple>
```

```
var fixpointReached: boolean
```

```
data.clear()
```

```
data.add(assertions)
```

repeat ▷ Loop for determining the fixpoint

```
  fixpointReached <- true
```

```
  for rule : rules do
```

```
    if rule.matches(data) then
```

```
      oldData = data.copy()
```

```
      if rule.type==derivation then
```

```
        data.add(rule.match(data).data)
```

```
      else ▷ So the rule must be an interaction rule
```

```
        if rule.match(data).request.type==GET then
```

```
          data.add(rule.match(data).request.execute())
```

```
        end if
```

```
      end if
```

```
      if ! data.copy().remove(oldData).isEmpty() then
```

```
        fixpointReached <- false
```

```
      end if
```

```
    end if
```

```
  end for
```

```
until fixpointReached
```

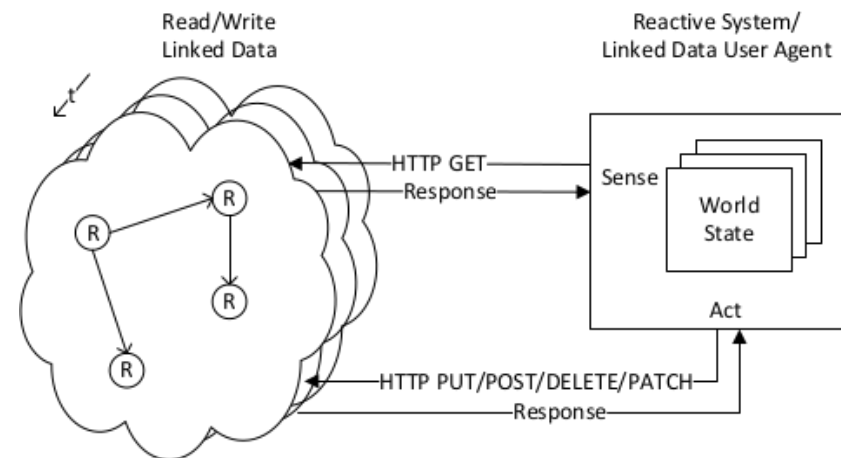
**Algorithm for
Constructing
an RDF
Dataset
Based on
Request
Rules
(Integrated
using
Derivation
Rules)**



Linked Data-Fu Overview

- Approach for accessing, integrating, querying and manipulating web data
- The language allows developers to specify interactions using rules
- The engine executes desired interactions in parallel

- **Derivation rules** support reasoning constructs, e.g., transitivity, reflexivity of properties
- **Request rules** specify how and when to interact with resources, i.e., retrieve the state of resources (sense) or manipulate the state of resources (act)



<http://linked-data-fu.github.io/>

Stadtmüller, Speiser, Harth, Studer: Data-Fu: a language and an interpreter for interaction with read/write linked data. WWW 2013

Linked Data-Fu



- A system to
 - execute programs with request rules to construct a RDF dataset
 - apply entailment patterns expressed in Notation3
 - process SPARQL queries, including entailment, over the RDF dataset created via link-following
- Linked Data-Fu programs run as user agents
 - Request rules can specify link-following based on HTTP GET requests
 - With allowing additional HTTP requests (PUT, POST, DELETE), the user agents can effect change in resource state

Agenda


Rules for:

- Reasoning
- Link following
- **Programming**

From Linked Data to Read-Write Linked Data

- With HTTP GET requests, one can implement systems that answer queries on data published on the web
- But HTTP has more request methods:
 - HTTP POST is used on the web to handle HTML forms and can be used to create resources
 - HTTP PUT can be used to overwrite resource state
 - HTTP DELETE can be used to delete resources
- With POST, GET, PUT and DELETE, one can implement applications that require CRUD (create-read-update-delete) operations on web architecture

Putting the *Web* back into the Semantic Web

- Linked Data Platform (W3C recommendation specified led by IBMers)
 - Read-Write interaction with Linked Data resources and collections of Linked Data resources
- Solid: Social Linked Data
 - Conventions and tools (mainly  JavaScript) for building decentralised social applications based on Read-Write Linked Data
 - Users store personal data in "pods" (personal online data stores) hosted wherever the user desires
- Web of Things



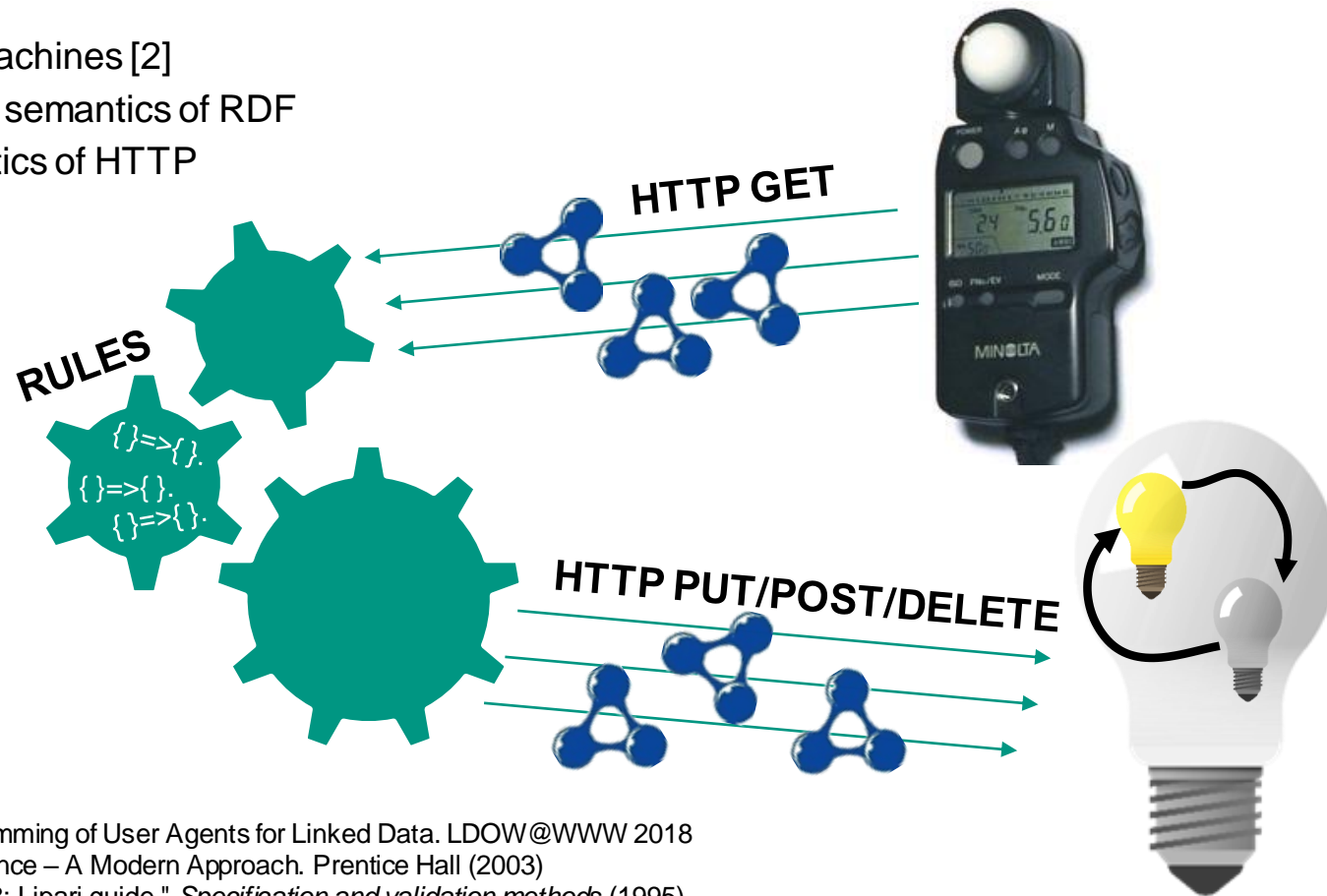
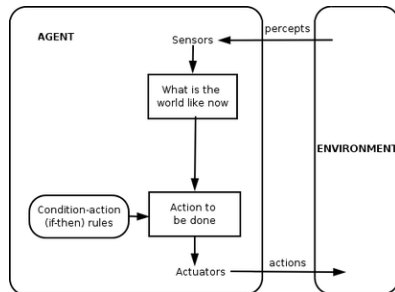
The article in the Scientific American is a lot about ontologies

Programming User Agents: ASM4LD [0]

- Aim: Execution of agent specifications on Read-Write Linked Data
- Inspired by Simple Reflex Agents [1]
- Based on:
 - Abstract State Machines [2]
 - Model-theoretics semantics of RDF
 - Message semantics of HTTP

- In a nutshell:


```
while(true):
    sense()
    think()
    act()
```



[0] Käfer & Harth: Rule-based Programming of User Agents for Linked Data. LDOW@WWW 2018
 [1] Russell & Norvig: Artificial Intelligence – A Modern Approach. Prentice Hall (2003)
 [2] Gurevich: "Evolving algebras 1993: Lipari guide." *Specification and validation methods* (1995)

Require: assertions ▷ Graph

Require: rules ▷ Derivation and request rules

var data, oldData: set<triple>

var fixpointReached: Boolean

var unsafeRequests: set<request>

while true do ▷ Loop of the ASM steps

 unsafeRequests.clear()

 data.clear()

 data.add(assertions)

repeat ▷ Loop for determining the fixpoint and the update set

 fixpointReached <- true

for rule : rules **do**

if rule.matches(data) **then**

 oldData = data.copy()

if rule.type==derivation **then**

 data.add(rule.match(data).data)

else ▷ So the rule must be an interaction rule

if rule.match(data).request.type==GET **then**

 data.add(rule.match(data).request.execute())

else

 unsafeRequests.add(rule.match(data).request)

end if

end if

if ! data.copy().remove(oldData).isEmpty() **then**

 fixpointReached <- false

end if

end if

end for

until fixpointReached

for request : unsafeRequests **do** ▷ Enacting the update set

 request.execute()

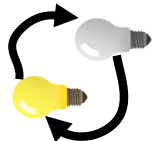
end for

end while

**Algorithm to
combine
materialisation,
link following,
and
programming**

Turn the Light On in Linked Data-Fu

■ Loop



$\{ \langle 0.5 \rangle = \langle \dots \rangle$

$\langle \dots \rangle$



$\}$

```
{ [] a http:Request ;
  http:hasMethod httpM:GET ;
  http:requestURI </ambient/light> . }
```

```
{ [] a http:Request ;
  http:hasMethod httpM:GET ;
  http:requestURI </relay/1> . }
```

```
{ </ambient/light> rdf:value ?val .
  ?val math:lessThan 0.5 .
  </relay/1#r> :isOn false . }
```

=>

```
{ [] a http:Request ;
  http:hasMethod httpM:PUT ;
  http:requestURI </relay/1> ;
  http:body
    { </relay/1#r> :isOn true . } . }
```

SENSE:
Retrieve the
world state

THINK:
Conditionally...

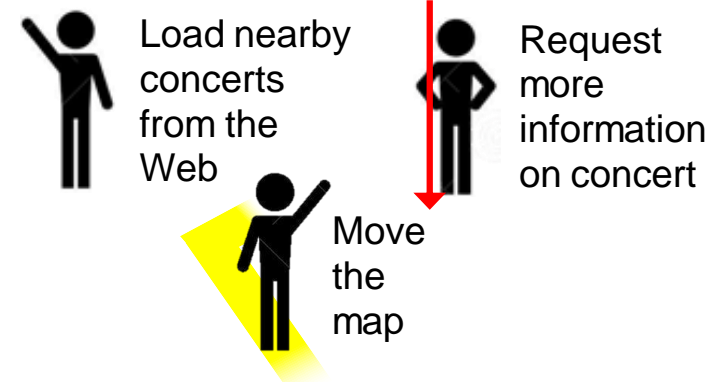
ACT:
...manipulate
the world state

Higher-level Ways of Programming Agents

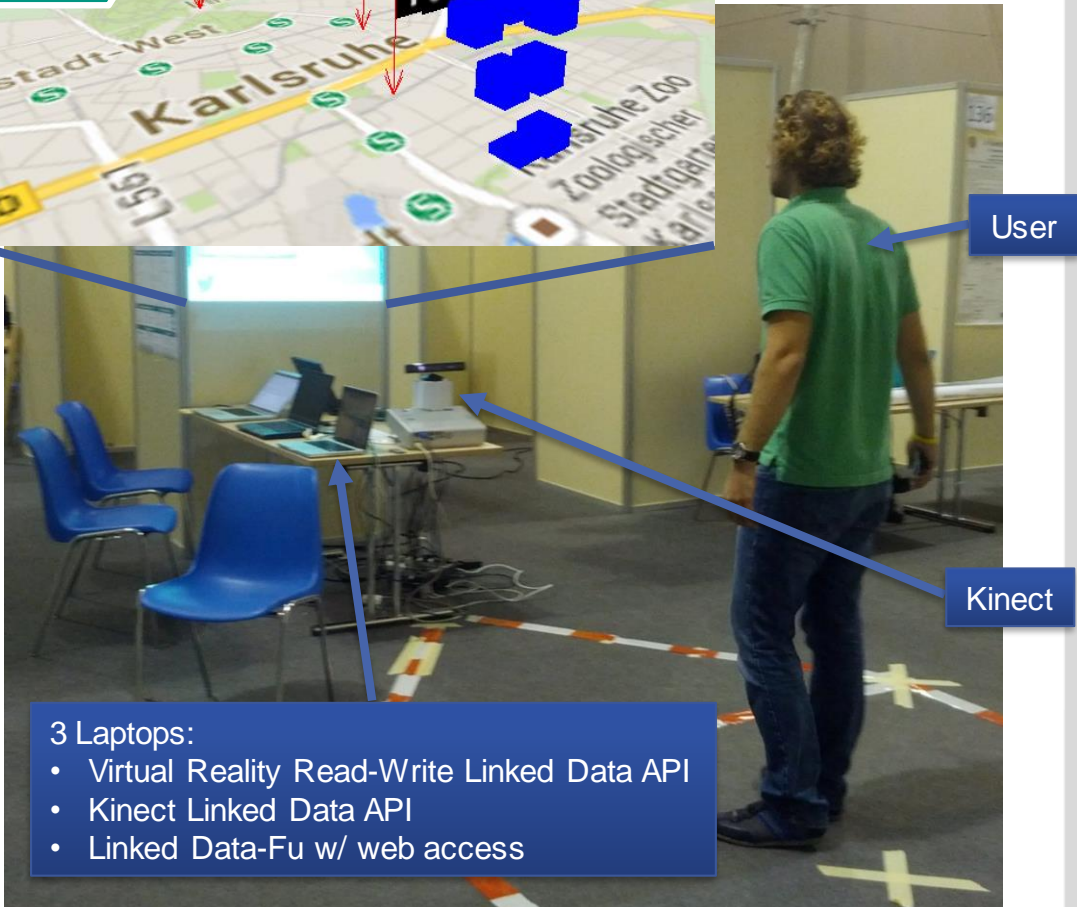
- We can use ASM4LD to give operational semantics to ontologies
- WiLD – Workflows in Linked Data
 - A flow-based workflow language
 - Käfer and Harth: „Specifying, Monitoring, and Executing Workflows in Linked Data“. Proc. ISWC 2018.
- GSM4LD
 - An artifact-centric workflow language
 - Jochum, Nürnberg, Aßfalg, Käfer: „Data-Driven Workflows for Specifying and Executing Agents in an Environment of Reasoning and RESTful Systems“. Proc. WS AI4BPM @ BPM 2019.

Integration of Distributed Systems using Linked Data: Example: a Virtual Reality System

Kinect tracks user → Avatar moves accordingly
Gestures trigger actions



- We encoded in Linked Data-Fu rules:
 - Movement of the avatar according to Kinect data
 - Detection of user gestures
 - Movement of the map according to gestures
 - Loading of concert data from the web
 - Data integration between VR RWLD API, concert LD API, Kinect LD API
- Execution at Kinect sensor refresh rate (30Hz)



Keppmann, Käfer, Stadtmüller, Schubotz, Harth: "High Performance Linked Data Processing for Virtual Reality Environments". P&D ISWC 2014.

Exercise

- Add a `rdfs:seeAlso` link from your document to `<https://ci.mines-stetienne.fr/kg/>`
- Create a rule that follows `rdfs:seeAlso` links
- Run

```
ldfu.sh -i http://where.is/your/turtle.ttl -p rule-to-follow-rdfs-seeAlso.n3 -q myquery.rq -
```
- Add a triple with `rdfs:subClassOf` that relates your `System` class with `ssn:System`
- Change your SPARQL query such that it looks for `ssn:System`
- Run

```
ldfu.sh -i http://where.is/your/turtle.ttl -p rule-to-follow-rdfs-seeAlso.n3 -p rulesets/rdfs.n3 -q myquery.rq -
```


**THANKS FOR YOUR
ATTENTION!**

Creative Commons Licensing

- The slides have been prepared by Tobias Käfer, Andreas Harth, and Lars Heling
- This content is licensed under a Creative Commons Attribution 4.0 International license (CC BY 4.0):
<http://creativecommons.org/licenses/by/4.0/>

